7^{th} International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2010)

Goetz Botterweck, Luís Lamb, João M. Fernandes (eds.)

20 September 2010

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York New York 10121-0701

ACM ISBN: 978-1-4503-0123-7

Preface

Software systems development demands sound methodologies, models, principles and appropriate tools. The MOMPES workshops focus on the theoretical and practical aspects related to the adoption of Model-based Development (MBD) methodologies for supporting the construction of software for pervasive and embedded systems. This International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2010) is the 7th edition of this workshop series. Over the years, the workshops have always contained a mix of industry and academic research papers, fostering productive collaboration amongst the two communities.

Since its first edition in 2004, the workshop has been co-located with prestigious international scientific conferences: ACSD 2004, ACSD 2005, ECBS 2006, ETAPS 2007, ETAPS 2008 and ICSE 2009. In 2010, MOMPES is co-located with the The 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010), one of the leading conferences on Software Engineering.

This book compiles the proceedings of the workshop held on 20 September 2010 in Antwerp, Belgium in conjunction with ASE 2010. Out of 14 submissions, 9 were selected for inclusion in the proceedings and presentation at the workshop. Each submission was reviewed by at least three program committee members.

The papers cover a large spectrum of topics including such model-driven engineering, model-based testing, variability modeling, modeling of avionic systems, design space exploration, domain-specific modeling, application of model transformations, specification of test oracles, wireless sensor networks, and Simulink-based analysis of systems.

We hope that you find this program interesting and thought-provoking and that the workshop provided you with a valuable opportunity to share ideas with other researchers and practitioners from around the world.

September 2010
Luís Lamb, João M. Fernandes, and Goetz Botterweck

Organisation

Program Committee

Ebrahim Bagheri, Athabasca University (CA)
Goetz Botterweck, Lero (IE), co-chair
Jordi Cabot, INRIA (FR)
Gaälle Calvary, U Joseph Fourier (FR)
Dionisio de Niz, CMU (US)
João M. Fernandes, U Minho (PT); co-chair
Robert France, Colorado State University (US)
Artur d'Avila Garcez, City U London (UK)
Luís Gomes, UN Lisboa (PT)
Hans-Gerhard Gross, TU Delft (NL)
Chris Hankin, Imperial College (UK)
Stefan Kowalewski, RWTH Aachen (DE)
Luís Lamb, UFRGS (BR); co-chair
Stephen J. Mellor, Mentor Graphics (US)
Dirk Muthig, Lufthansa Systems (DE)
Isabelle Perseil, TELECOM Paristech (FR)
Iris Reinhartz-Berger, University of Haifa (IL)
Pablo Sánchez, University Cantabria (ES)
João P. Sousa, George Mason University (US)

Reviewers

David Ameller
Eva Beckschulze
Joerg Brauer

Steering Committee

Ricardo J. Machado (chair), Universidade do Minho
Dov Dori, Technion
João M. Fernandes, Universidade do Minho
Mike Hinchey, Lero – The Irish Software Engineering Research Centre
Flávio R. Wagner, UFRGS

Acknowledgments

MOMPES 2010 was co-organised by Universidade Federal do Rio Grande do Sul (UFRGS), Universidade do Minho, and Lero – The Irish Software Engineering Research Centre, Limerick, Ireland.

This work was partly supported by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre, http://www.lero.ie/.

We, the workshop organisers, are grateful to the members of the program committee and the reviewers. We also would like to thank the organisers of ASE 2010, in particular Tom Mens and Tevfik Bultan, the workshop chairs, for their support and the opportunity to hold MOMPES 2010 in conjunction with this well-known conference.

Contents

Support for Variability in Use Case Modeling with Refinement

Sofia Azevedo, Ricardo J. Machado
Universidade do Minho
Dep. de Sistemas de Informação
Guimarães, Portugal
+351 253 510 319
{sofia.azevedo,rmac}@dsi.uminho.pt

Alexandre Bragança
ISEP
Dep. de Eng. Informática
Porto, Portugal
+351 22 834 05 24
alex@dei.isep.ipp.pt

Hugo Ribeiro
Primavera BSS
Rua Cidade do Porto, 79
Braga, Portugal
+351 253 309 900
hugo.ribeiro@primaverabss.com

ABSTRACT
The development of software product lines with model-driven approaches involves dealing with diverse modeling artifacts such as use case diagrams, component diagrams, class diagrams, activity diagrams, sequence diagrams and others. In this paper we focus on use cases for product line development and we analyze them from the perspective of variability. In that context we explore the UML (Unified Modeling Language) *«extend»* relationship. We also explore the functional refinement of use cases with *«extend»* relationships between them. This work allows understanding the activities of use case modeling with support for variability and of use case modeling with functional refinement when variability is present.

Keywords
Use case, software product line, variability, *«extend»*, alternative, option, specialization, refinement.

1. INTRODUCTION
Use case diagrams are one of the modeling artifacts modelers have to deal with when developing product lines with model-driven approaches. This paper envisions use cases according to the perspective of variability. The *«extend»* relationship plays a vital role in variability modeling in the context of use cases and allows for the use case modeling activity to be applicable to the product line software development approach. That is possible by determining the locations in use case diagrams where variation will occur when instantiating the product line. This paper's contribution is on the formalization and understanding of the use case modeling activity with support for variability. We will illustrate our approach with the Fraunhofer IESE's GoPhone case study [1], which presents a series of use cases for a part of a mobile phone product line particularly concerning the interaction between the user and the mobile phone software. We propose an extension to the UML (Unified Modeling Language) metamodel [2] in order to formally provide for both the concrete and abstract syntaxes to represent different types of variability in use case diagrams. We consider use cases in different abstraction levels to elaborate on the (functional) refinement of use cases with *«extend»* relationships between them. In this paper we focus on the variability support as well as on the process point of view with regards to the use case modeling activity.

The paper is structured as follows. Section 2 elaborates on the differences between others' approaches and this paper's approach. Section 3 elaborates on the different types of variability we propose to be used in the context of use case modeling. Section 4 provides for the analysis of the UML *«extend»* relationship in contexts of variability and also for the extension we propose to the UML metamodel to support the different variability types. Section 5 analyzes the process of handling variability in use case diagrams in the context of the functional refinement of use cases. Section 6 illustrates our approach with the GoPhone case study. Finally Section 7 affords some concluding remarks.

MOMPES'10, September 20, 2010, Antwerp, Belgium.

2. RELATED WORK
Despite use cases being sometimes used as drafts during the process of developing software and not as modeling artifacts that actively contribute to the development of software, use cases shall have mechanisms to deal with variability in order for them to have the ability to actively contribute to the process of developing product lines. For instance, modeling variability in use case diagrams is important to later model variability in activity diagrams [3].

This paper's work is inspired on the approach of Bragança and Machado to variability modeling in use case diagrams [4]. Bragança and Machado represent variation points explicitly in use case diagrams through extension points. Their approach consists of commenting *«extend»* relationships with the name of the products from the product line on which the extension point shall be present. Their approach to product line modeling is bottom-up (rather than top-down), which means that all the product line's products are known *a priori*. A top-down approach would consider that the product line would support as many products as possible within the given domain. In [5] John and Muthig refer to required and anticipated variations as well as to a planned set of products for the product line, which indicates that their approach to product line modeling is bottom-up. The approach in this paper adopts the top-down approach for product line modeling, therefore discarding the comments to the *«extend»* relationships.

In [5] John and Muthig refer the benefits of representing variability in use cases. Although we totally agree with the position of these authors towards those benefits, we cannot agree when they state that information on whether certain use cases are optional or alternatives to other use cases shall only be in decision models as it would overload use case diagrams and make them less readable. Our position is that features as well as use cases shall be suited for treating variability in its different types. If a use case is an alternative to another use case, then both use cases shall be modeled in the use case diagram, otherwise the use case diagram will only show a part of the possibilities of the possible products John and Muthig mention in [5].

Gomaa and Shin [6] analyze variability in different modeling views of product lines. They mention that the *«extend»* relationship models a variation of requirements through alternatives. They also model options in use case diagrams by using the stereotype *«optional»* in use cases. We adopt these approaches to alternatives and options but we elaborate on another form of variability (specializations, which we consider to be a special kind of alternatives). Gomaa and Shin refer specialization as a means to express variability in [6]. Besides alternative and optional use cases, Gomaa and Shin consider kernel use cases (use cases common to all product line members). Gomaa models in [7] kernel and optional use cases both with the *«extend»* as well as with the *«include»* relationships (our approach is towards modeling kernel and optional use cases independently of their involvement in either *«extend»* or *«include»* relationships and with a stereotype in the use cases).

Halmans and Pohl propose in [8] use cases as the means to communicate variability relevant to the customer and they also propose extensions to use case diagrams to represent variability relevant to the customer. Halmans and Pohl consider that generalizations between use cases are adequate to represent use cases' variants. This is not our position. We recommend using the *«extend»* relationship instead of the generalization relationship. Halmans and Pohl consider that modeling mandatory and optional use cases with stereotypes in use cases is not adequate because the same use case can be mandatory for one use case and optional for another. Again this is not our position. We also consider that a mandatory use case is not mandatory with regards to another use case, rather it is mandatory for all product line members. We also consider that an optional use case is optional with regards to one or more product line members. Halmans and Pohl end up by introducing additional graphical elements to use case diagrams to represent variation points and variability cardinality explicitly in use case diagrams. We do not agree with this approach since it introduces more complexity to use case diagrams than modeling variability with stereotypes and use case relationships as well as it introduces a reasoning about variability that should be present in decision models (the selection of the variants to be present in the system and the system/product to which that selection applies according to the features).

Maßen and Lichter talk about three types of variability in [9]: optional, alternative and optional alternative (as opposite to alternatives that represent a "1 from n choice", optional alternatives represent a "0 or 1 from n choice"). In this context they propose to extend the UML metamodel to incorporate two new relationships for connecting use cases. Our approach considers options and alternatives as well but we introduce these concepts into the UML metamodel through stereotypes (we consider that the *«extend»* relationship is adequate for modeling alternatives and a stereotype applicable to use cases for modeling options).

According to Gomaa [7], and John and Muthig [5], use cases can be tagged with some stereotypes concerning variability. Table 1 shows the applicability of those stereotypes in our approach.

Table 1. Some use case stereotypes concerned with variability.

Stereotype	Applicability
«kernel»	Use cases in general
«alternative»	*«extend»* relationships
«optional»	Use cases in general
«variant»	Use cases in general

Some examples of approaches to functional decomposition of software systems are the 4SRS (Four Step Rule Set) method [10], KobrA or RSEB (Reuse-Driven Software Engineering Business) [11, 12]. However neither KobrA nor RSEB clearly contemplate a technique for refining use cases like the 4SRS method does.

Greenfield and Short [13] refer to refinement as the inverse of abstraction or the process of turning a description more complex by adding information to it. They refer to the process of developing software through refinement as progressive refinement. The process starts with requirements and ends up with the more concrete description of the software (the executable). They consider refinement as a concatenation of interrelated transformations mapping a problem to a solution. The goal of refinement is to smoothly decrease the abstraction levels that separate the problem from the solution. In general terms, Greenfield and Short talk about refinement as the stepwise decomposition of features' granularity. In the context of use cases, refinement is their detailing. However we defend that use cases can themselves be refined in order to facilitate the transformation of a problem (which can be modeled with use cases) to a solution (which shall be modeled with design artifacts e.g. logical architectures).

Gomaa [7] explored refinement in the context of feature modeling, where a feature can be a refinement of another. But in order to get to the features, use cases have to be modeled and mapped to features. Our approach eliminates this mapping activity. To Gomaa the refinement is expressed through *«extend»* relationships in the context of use cases. To us the refinement shall be expressed through the *«refine»* relationship we proposed in [14].

Cherfi, *et al*. [15] (in their work on quality-based use case modeling with refinement) describe the refinement process as the application of a set of decomposition and restructuring rules to the initial use case diagram. Their approach is iterative and incremental. It consists of decomposing the initial use case diagram into smaller and more cohesive ones to decrease the complexity of the diagram and increase its cohesion. In the approach of Cherfi, *et al*. to refinement, use cases are not actually detailed (like in ours), rather they are decomposed without detail being added to the description of those use cases.

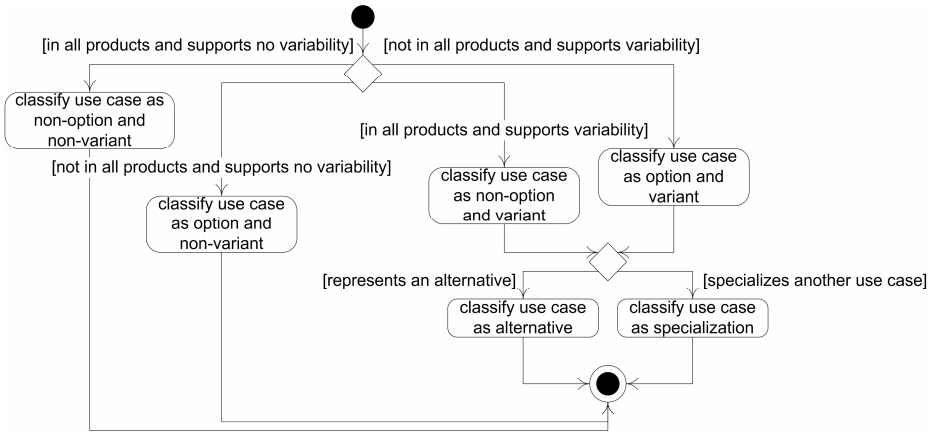


Figure 1. The use case variability types.

3. HANDLING VARIABILITY IN USE CASE MODELING

Figure 1 illustrates the variability types we consider and propose to be applicable in the context of use cases [16]. Use cases can be non-option or option. Non-option use cases are present in all product line members. Option use cases can be present in one product of the product line and not in another. It is not mandatory that option use cases are present in all products of the product line. Non-variant use cases are use cases that do not support variability. Variant use cases are use cases that support variability. This means that different products will support different alternatives for performing the same functionality or that different products will support different specializations of the same functionality. Later on during the modeling activity variant use cases are realized into alternatives or specializations respectively. Alternative use cases represent alternatives for performing the same system's use in mutually exclusive products or sets of products from the product line. Specialization use cases represent a special kind of alternatives. A specialization use case is a specialization of another use case. Specialization use cases that specialize the same use case represent alternatives for performing the same system's use in mutually exclusive products or sets of products from the product line. Option, alternative and specialization use cases are the representation of the three variability types that will be translated into stereotypes to be applicable to use cases. The use cases that do not represent options and are not variant (later alternatives or specializations) are non-option and non-variant, and shall not be marked with any stereotype. Non-option and option use cases are mutually exclusive as well as non-variant and variant use cases. Figure 1 represents the activity of classifying use cases with variability types: either *non-option and non-variant* or *option and non-variant* or *non-option and variant* or *option and variant*. These last two variability types can be realized into the *alternative* or the *specialization* variability types (as already explained). The activity of classifying use cases with the variability types is important for applying the corresponding stereotypes to the use cases (except for the *non-option and non-variant* variability type, which shall not be marked with any stereotype). The conditions of the decision nodes express the semantics of each one of the variability types. We would like to give emphasis to a particular variability type: the *option and variant* variability type. This variability type is applicable to a use case that is not present in all product line members but the different members in which it is present support different alternatives for performing that use case's functionality or different specializations of that use case's functionality. *Option and non-variant* use cases shall be marked as option use cases; *non-option and variant* as variant use cases; and *option and variant* use cases as both option and variant use cases.

4. THE *«extend»* RELATIONSHIP

The *«extend»* relationship allows modeling alternative and specialization use cases in use case diagrams.

Consider that an extending use case is a use case that extends another use case and that an extended use case is a use case that is extended by other use cases. As any other use case, an extending use case represents a given use of the system by a given actor or actors.

In the context of alternatives [16] both extending and extended use cases represent supplementary functionality since both represent alternatives, which are not essential for a product without variability to function. It shall be noted that alternatives are no longer supplementary when product line members are instantiated from the product line. Alternatives can be modeled with the generalization relationship in use case diagrams, but we recommend to model alternatives with the *«extend»* relationship in order to evidence their supplementary character according to the UML semantics.

If the intention is to use differential specification, specializations [16] shall be modeled with the *«extend»* relationship, otherwise they shall be modeled with the generalization relationship. Differential specification of specializations means that specialization use cases represent supplementary functionality regarding the use case they specialize, therefore a product without variability does not require the specialization use cases to function.

Options [16] represent functionality that is only essential for a product with variability to function, therefore options represent supplementary functionality. However we do not recommend modeling options with the *«extend»* relationship because if the stereotype was on the relationship, the relationship itself would be optional and that is not the case (the use case is not optional with regards to any other use case, rather it is optional by itself).

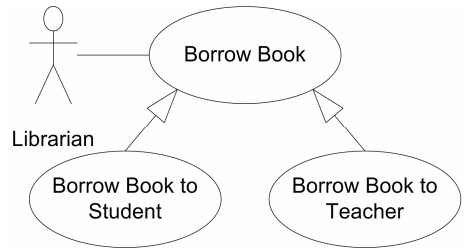


Figure 2. The specialization of the variant use case *Borrow Book* with a single actor.

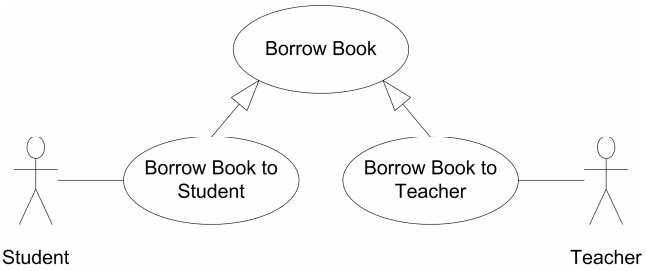


Figure 3. The specialization of the use case *Borrow Book* with two different actors.

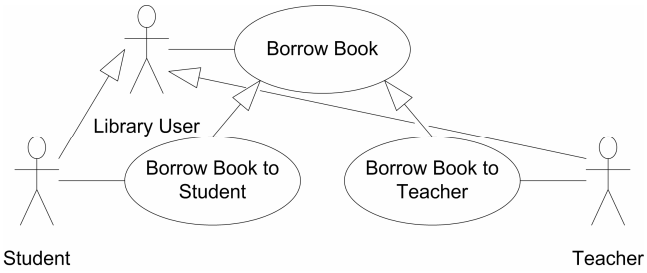


Figure 4. The specialization of the variant use case *Borrow Book* with two different actors.

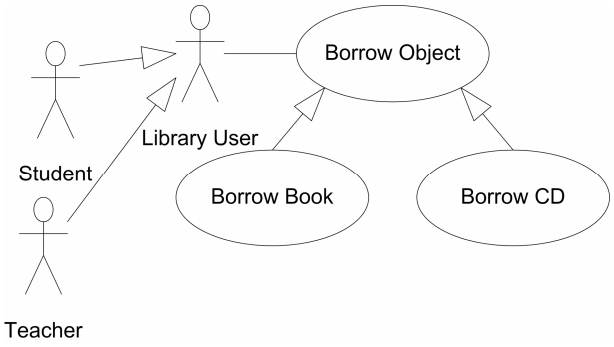


Figure 5. The specialization of the variant use case *Borrow Object*.

Options shall be modeled with a stereotype in use cases. The involvement of an option use case in either *«extend»* or *«include»* relationships, or even in none of those does not imply the presence of that use case in all product line members (which makes of it optional).

In principle an extending use case is a use case that extends another use case both in the case of alternatives and in the case of specializations. In the case of specializations we consider that there is no multiple inheritance, therefore it is impossible for an extending use case to extend more than one use case. If we have more than one alternative use case for the same functionality, one of those use cases shall be the alternative to all the others and extended by them. That use case is the one to be present in the products less robust in terms of functionality. The extended use case is not aware of the functionality described in the extending use case.

As previously mentioned if the intention is not to use differential specification, generalization relationships shall be used because specializations are complementary under those circumstances. However we may argue in a different way that the generalization relationship shall not be used to represent specializations in contexts of variability. Consider the examples depicted in figures 2 through 5 Figure 2. The example is an exception in terms of the (GoPhone) case study we will use further on in this paper. The figure shows that the use case *Borrow Book* can be specialized into *Borrow Book to Student* and *Borrow Book to Teacher*. If the actor is the same (the *Librarian*, who registers the borrowing), then the use cases that specialize the *Borrow Book* use case are alternatives to borrowing a book as both can be performed by the same actor. If the actor is not the same (the *Student* in the case of the *Borrow Book to Student* and the *Teacher* in the case of the *Borrow Book to Teacher*), then the use cases that specialize the *Borrow Book* use case are not alternatives to borrowing a book as both cannot be performed by the same actor (the same actor does not have an alternative way of borrowing a book). In this case in order for the generalization to be considered as variability, the actor of *Borrow Book* has to be the *Library User* (connected to *Borrow Book*) specialized into the *Student* (connected to *Borrow Book to Student*) and into the *Teacher* (connected to *Borrow Book to Teacher*). Another example: the use case *Borrow Object* can be specialized into *Borrow Book* and *Borrow CD*. In this case the actor can be the same for all of the use cases (the *Student* OR the *Teacher*). In order to support all the actors at the same time (the *Student* AND the *Teacher*), the *Library User* has to be specialized into them (the *Student* and the *Teacher*) and connected to the *Borrow Object* use case. This way the same actor (the *Library User*) can borrow an object (a *Book*) or alternatively another (a *CD*).

Figure 6 depicts the extension we propose to the UML metamodel concerning the *«extend»* relationship and use cases. We have added the stereotypes *«alternative»*, *«specialization»* and *«option»* to the standard UML stereotypes in order to distinguish the three variability types that were to be translated into stereotypes to be applicable to use cases. We have also added the stereotype *«variant»* to the standard UML stereotypes in order to mark use cases at higher levels of abstraction before they are realized into alternatives or specializations. We propose the stereotype *«option»* to be applicable to use cases that represent options. We also propose the stereotypes *«alternative»* and *«specialization»* to be applicable to the *«extend»* relationship for modeling alternatives and specializations respectively. Extending use cases involved in *«alternative»* relationships do not need to be marked with the stereotype *«alternative»* to evidence them as alternatives since they do not make sense without being involved

in that kind of relationships (an alternative use case is always alternative to another use case). The same happens with the stereotype *«specialization»* (a use case involved in a specialization relationship always specializes another use case). Regarding Figure 6 and the *Extend* metamodel element, as far as the unidirectional association is concerned, the end named *extendedCase* references the use case that is being extended (the extended use case) and the association means that many (zero or more) *«extend»* relationships refer to one extended use case. Regarding the aggregation, the end named *extend* references the *«extend»* relationships owned by the use case, and the end named *extension* references the use case that represents the extension (the extending use case) and owns the *«extend»* relationship. The metamodel means that one *«extend»* relationship is owned by one extending use case. Summarily a use case can be extended by many use cases and a use case can extend another use case. There can be zero or more alternatives (*«alternative»* relationships) to a use case. There can also be zero or more specializations (*«specialization»* relationships) for a use case. Although it can be argued that specializations are only worth the effort when there are two or more specialization use cases, we do not want to take freedom away from the modeler.

From now on we either use the *«extend»* relationship without stereotypes or with one of the two stereotypes applicable to this relationship from the proposed extension to the UML metamodel (depending on whether we are modeling alternatives or specializations).

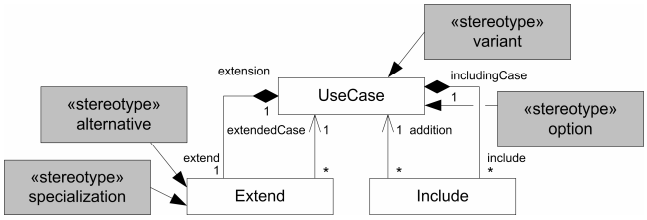


Figure 6. The proposed extension to the UML metamodel for modeling variability in use case diagrams.

It is important to distinguish alternatives from generalizations in contexts of variability. In the case of alternatives the extending use case is an alternative to the extended use case. In the case of specializations the extending use cases are alternatives to each other. Figure 7 shows the specialization of two alternative use cases from the GoPhone case study: *Insert Picture* and *Insert Picture or Draft Text*. It is possible to transform alternatives into specializations and the other way around. Again we are not restrictive on this since we do not want to take freedom away from the modeler.

5. HANDLING VARIABILITY IN USE CASE MODELING WITH REFINEMENT

Use cases can be decomposed with or without detailing their non-stepwise textual descriptions. Without detailing those descriptions we propose to represent the decomposition of use cases in use case diagrams with the *«include»* relationship. This decomposition suits the purpose of e.g. modeling later on an alternative to a part of the decomposed use case or modeling a part of the decomposed use case that is an optional part).

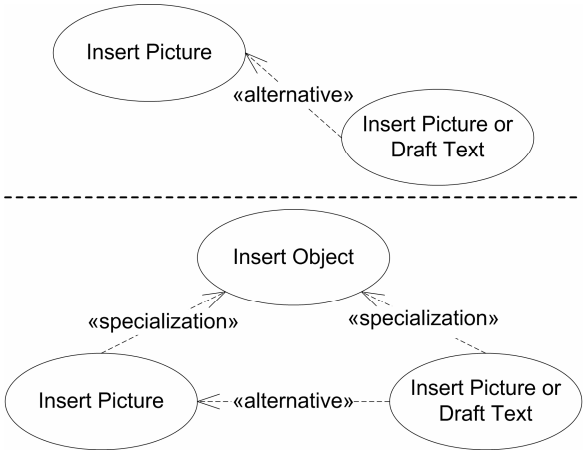


Figure 7. The specialization of *Insert Picture* and *Insert Picture or Draft Text*.

We consider that refining means decomposing and simultaneously detailing use cases. By refining use cases, the artifacts resulting from the refinement process (the refining use cases) are situated in lower abstraction levels comparatively to the refined use cases (the use cases that were submitted to the refinement process). In order to represent in the use case diagram this decrease in the abstraction level when refining use cases, we proposed in [14] to use the *«refine»* relationship (as a sort of traceability between use cases at different levels of detail).

In this section of the paper we depict in Figure 8 use cases according to the perspectives of detail*variability to illustrate in abstract terms our approach to use case modeling with support for variability. The detail perspective is intimately related to the activity of use case refinement. In this sense use cases can be more detailed if they are refined. The variability perspective is associated with the modeling of variability for product line support. The two perspectives (detail and variability) have been converted into axes of the illustrated space: y=detail and z=variability. Each level of the z axis corresponds to a (parallel) plan, which means that we position use cases in variability plans. Thus variability plans are plans that contain use cases representing variability in the three different types that have been translated into stereotypes to be applicable to use cases. The plan z=0 contains none of these use cases that represent variability.

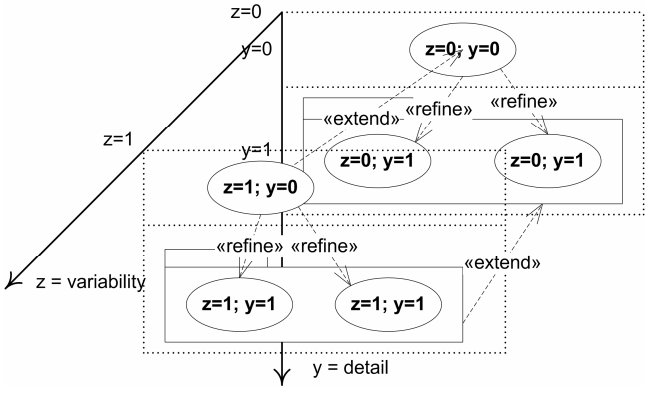


Figure 8. Use cases positioned according to the perspectives of detail*variability.

Use case name: Send Message
Use case description: The mobile user writes the message in a text editor. The GoPhone connects to the network to send the message. In order for the GoPhone to show an acknowledgement to the mobile user (stating that the message was successfully sent), it receives an acknowledgement from the network. Upon request from the GoPhone, the mobile user chooses to save the message into the sent messages folder.
Alternatives: The mobile user sends some different kinds of messages through the GoPhone.
The mobile user inserts objects into a message.
The mobile user attaches objects to a message.
The mobile user chooses the recipient's contact.
Specializations: -
Options: When writing the message, the mobile user activates letter combination (T9).

Use case name: Compose Message
Use case description: The mobile user writes the message in a text editor.
Alternatives: The mobile user sends some different kinds of messages through the GoPhone.
The mobile user inserts objects into a message.
The mobile user attaches objects to a message.
Specializations: -
Options: When writing the message, the mobile user activates letter combination (T9).

Use case name: Archive Message by Request
Use case description: Upon request from the GoPhone, the mobile user chooses to save the message into the sent messages folder.
Alternatives: The GoPhone automatically archives the message
Specializations: -
Options: -

Use case name: Automatically Archive Message
Use case description: The GoPhone saves the message into the sent messages folder and notifies the mobile user on the successful message saving into that folder.
Alternatives: -
Specializations: -
Options: -

Figure 9. Non-stepwise textual descriptions from the GoPhone use case *Send Message* and some of its related use cases.

The figure clarifies that the *«refine»* relationships imply increasing the detail level, whereas the *«extend»* relationships do not imply increasing the detail level but rather changing from one variability plan (z plan) to another. Extending use cases represent alternative or specialization use cases, therefore they must be situated at the same level of detail but in different variability plans (z plans). Variabilities do not imply adding detail to the non-stepwise textual descriptions of the use cases, like refinements do.

The figure shows the general case of the refinement of two use cases connected through an *«extend»* relationship. The refinement of a use case stereotyped as *«option»* is not relevant here, since it is not the case of an *«extend»* relationship connecting two use cases. The figure evidences that the refinement of two use cases connected through an *«extend»* relationship originates more detailed use cases organized in two packages that have also an *«extend»* relationship connecting them. That is not always the case. It is possible to have two use cases connected through a *«specialization»* relationship, which produces *«specialization»* relationships connecting more detailed individual use cases (and not packages) in different variability plans (an example of such case is in the next section of this paper).

6. THE VARIABILITY IN THE GoPhone CASE STUDY

The non-stepwise textual descriptions in Figure 9 were elaborated based on the functional requirements for the GoPhone. We rely on non-stepwise textual descriptions of use cases (the opposite of stepwise textual descriptions of use cases) to model variability in use case diagrams. Stepwise textual descriptions are structured textual descriptions in natural language that provide for a stepwise view of the use case as a sequence of steps, alert for the decisions that have to be made by the user and evidence the notion of use case actions temporarily dependent on each other. Stepwise descriptions shall be treated after modeling the use cases.

The *«include»* relationship involves two types of use cases: the including use case (the use case that includes other use cases) and the included use case (the use case that is included by other use cases). In the context of the *«include»* relationship the UML Superstructure states that the including use case depends on the addition of the included use cases to be complete. Nevertheless in our opinion the functionality of the included use cases shall be described in the including use case. Since we rely on non-stepwise textual descriptions of use cases to determine the *«include»* relationships, the including use case has to contain the description of the included use cases so that the modeler is able to define the parts that compose the including use case in order to decompose that use case (e.g. as can be seen from Figure 9 the functionality of the *Compose Message* use case is described in the *Send Message* use case).

In the context of the *«extend»* relationship the UML Superstructure states that an extending use case consists of one or more behavior fragment descriptions to be inserted into the appropriate spots of the extended use case. This means that the functionality of the extending use case is not described in the

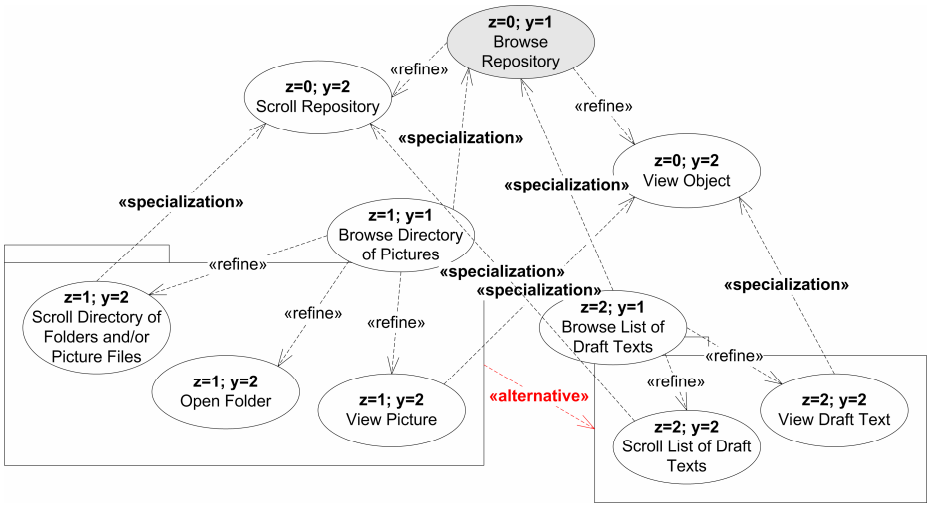


Figure 11. An example of refinement of the specialization type of variability from the GoPhone.

extended use case. The extended use case is not aware of the functionality described in the extending use case (e.g. as can be seen from Figure 9 the functionality of the *Automatically Archive Message* use case is not described in the *Archive Message by Request* use case). As Figure 10 depicts, the use case *Automatically Archive Message* is an alternative to the use case *Archive Message by Request* (they are connected through a kind of *«extend»* relationship, tagged with the stereotype *«alternative»* in order to evidence that the use case *Automatically Archive Message* is an alternative to the use case *Archive Message by Request*). It must be noticed that *Archive Message by Request* is an (included) use case included by the including use case *Send Message*, which means that the functionality of the use case *Archive Message by Request* is described in the *Send Message* use case. For this reason we could have extended the *Send Message* use case with the use case *Automatically Archive Message*, but then we would not be evidencing to which part of the functionality of the *Send Message* use case the use case *Automatically Archive Message* is an alternative to. Figure 10 also depicts that the *Browse Directory of Pictures* use case is a specialization of the use case *Browse Repository* (they are connected through another kind of *«extend»* relationship, tagged with the stereotype *«specialization»* in order to evidence that the use case *Browse Directory of Pictures* is a specialization of the use case *Browse Repository*).

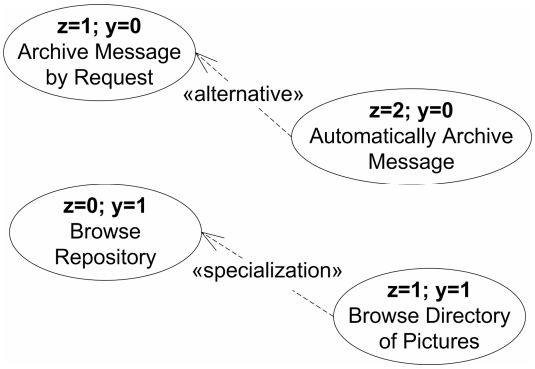


Figure 10. Some examples of variability modeled for the GoPhone use case *Send Message*.

6.1 Refinement of Specializations and Alternatives

Figure 11 shows the refinement of the specialization type of variability. The figure shows that both the use case that has been specialized (the *Browse Repository* use case) and the specialization use cases (the *Browse Directory* and *Browse List* use cases) were refined. Some use cases that refine the specialization use cases are specializations of the use cases that refine the use case that has been specialized (e.g. the *View Picture* use case is a specialization of the *View Object* use case). The use case *Open Folder* represents functionality that is not common to both specialization use cases since it is only applicable to one of the objects the specialization use cases refer to (the *Directory of Pictures*). Having in mind that specializations are a special kind of alternatives, specialization use cases are alternatives to each other. Figure 11 illustrates that the use cases that refine the specialization use cases are alternatives to each other as packages.

Figure 12 depicts that the use cases that refine two use cases connected through an *«alternative»* relationship are alternatives to each other as packages.

7. CONCLUSIONS

This paper has elaborated on the representation of variability in use case diagrams. It began by providing an in depth analysis of the state-of-the-art concerned with this topic. Based on our position towards the related work we proposed an extension to the UML metamodel to represent the three types of variability we have synthesized: alternatives, specializations and options. We concluded that alternatives and specializations shall be adequately modeled with the *«extend»* relationship, and that options shall be adequately modeled with a stereotype on use cases. This conclusion was based on the UML metamodel's semantics associated with the relationships for connecting use cases in use case diagrams: alternatives, specializations and options represent supplementary functionality. Although not being the core of this paper's contribute, we have also introduced the functional refinement of use cases connected through *«extend»* relationships due to its pertinence in large-scale product line contexts.

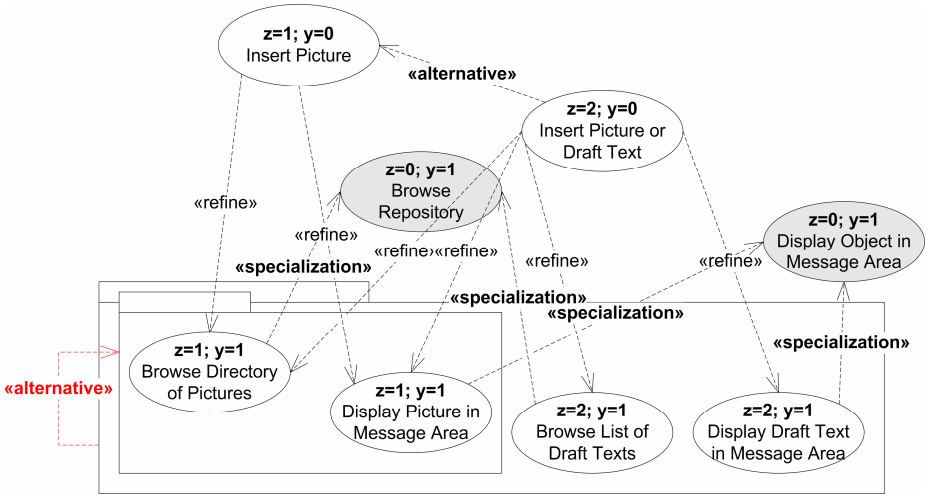


Figure 12. An example of refinement of alternative variability from the GoPhone.

8. REFERENCES

[1] Muthig, D., John, I., Anastasopoulos, M., Forster, T., Dörr, J. and Schmid, K. *GoPhone - A Software Product Line in the Mobile Phone Domain*. IESE-Report No. 025.04/E, Fraunhofer IESE, 2004.

[2] OMG *Unified Modeling Language: Superstructure - version 2.2*. Object Management Group, 2009.

[3] Bragança, A. and Machado, R. J. Extending UML 2.0 Metamodel for Complementary Usages of the «extend» Relationship within Use Case Variability Specification. In *Proceedings of the SPLC 2006* (Baltimore, Maryland, USA, August 21-24, 2006). IEEE Computer Society, 2006.

[4] Bragança, A. and Machado, R. J. Deriving Software Product Line's Architectural Requirements from Use Cases: An Experimental Approach. In *Proceedings of the MOMPES 2005* (Rennes, France, June 6, 2005). TUCS General Publications, 2005.

[5] John, I. and Muthig, D. Product Line Modeling with Generic Use Cases. In *Proceedings of the Workshop on Techniques for Exploiting Commonality Through Variability Management* (San Diego, California, USA, August 19, 2002). Springer-Verlag, 2002.

[6] Gomaa, H. and Shin, M. E. Multiple-View Modelling and Meta-Modelling of Software Product Lines. *Institution of Engineering and Technology Software*, 2, 2 2008), 94-122.

[7] Gomaa, H. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, Upper Saddle River, New Jersey, 2004.

[8] Halmans, G. and Pohl, K. Communicating the Variability of a Software-Product Family to Customers. *Software and Systems Modeling*, 2, 1 2003), 15-36.

[9] Maßen, T. v. d. and Lichter, H. Modeling Variability by UML Use Case Diagrams. In *Proceedings of the REPL 2002* (Essen, Germany, 2002). Avaya Labs, 2002.

[10] Machado, R. J., Fernandes, J. M., Monteiro, P. and Rodrigues, H. Transformation of UML Models for Service-Oriented Software Architectures. In *Proceedings of the ECBS 2005* (Greenbelt, Maryland, USA, April 4-7, 2005). IEEE Computer Society, 2005.

[11] Atkinson, C., Bayer, J. and Muthig, D. Component-Based Product Line Development: The KobrA Approach. In *Proceedings of the SPLC 2000* (Denver, Colorado, USA, August 28-31, 2000). Kluwer Academic Publishers, 2000.

[12] Jacobson, I., Griss, M. and Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, Upper Saddle River, New Jersey, 1997.

[13] Greenfield, J. and Short, K. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Hoboken, New Jersey, 2004.

[14] Azevedo, S., Machado, R. J., Bragança, A. and Ribeiro, H. The UML «include» Relationship and the Functional Refinement of Use Cases. In *Proceedings of the SEAA 2010* (Lille, France, September 1-3, 2010). IEEE Computer Society, 2010 (accepted for publication).

[15] Cherfi, S. S.-s., Akoka, J. and Comyn-Wattiau, I. Use Case Modeling and Refinement: A Quality-Based Approach. In *Proceedings of the ER 2006* (Tucson, Arizona, USA, November 6-9, 2006). Springer-Verlag, 2006.

[16] Azevedo, S., Machado, R. J., Bragança, A. and Ribeiro, H. The UML «extend» Relationship as Support for Software Variability. In *Proceedings of the SPLC 2010* (Jeju Island, South Korea, September 13-17, 2010). Springer-Verlag, 2010 (accepted for publication).

Automating Test Cases Generation: From xtUML System Models to QML Test Models

Federico Ciccozzi
Mälardalen University
Västerås, Sweden
federico.ciccozzi@mdh.se

Antonio Cicchetti
Mälardalen University
Västerås, Sweden
antonio.cicchetti@mdh.se

Toni Siljamäki
Ericsson AB
Stockholm, Sweden
toni.siljamaki@ericsson.com

Jenis Kavadiya
Tata Consultancy Services
Hyderabad, Andhra Pradesh,
India
jenis.kavadiya@tcs.com

ABSTRACT

The scope of Model-Driven Engineering is not limited to Model-Based Development (MBD), i.e. the generation of code from system models, but involves also Model-Based Testing (MBT), which is the automatic generation of efficient test procedures from corresponding test models. Both MBD and MBT include activities such as model creation, model checking, and use of model compilers for the generation of system/test code. By reusing these common activities, the efficiency of MBT can be significantly improved for those organizations which have already adopted MBD, since one of the major efforts in MBT is the creation of test models. In this work, we propose to exploit modeling activity efforts by deriving test models from system models. In this respect, we present a case study in which the Executable and Translatable UML system models are used for automatically generating test models in the QTronic Modeling Language using horizontal model transformations. In turn, the derived artefacts can be used to produce test cases which result to be appropriate for the system under development.

Keywords

Model-Driven Engineering, Model-Based Development, Model-Based Testing, Automatic Test Cases Generation

1. INTRODUCTION

Model Driven Engineering (MDE) aims at facilitating the system development by creating, maintaining and manipulating models. In fact, models provide abstractions of a real phenomena which reduce the complexity of the problem, allow to focus on the aspects that matter in the design of the application, and permit to reason about the scenario in terms of domain-specific concepts [4]. The MDE term was first introduced by Kent [10] and includes all modelling tasks needed for the entire software development process. A system is developed by refining models starting from higher and moving to lower levels of abstraction until code is generated; refinement is implemented by transformations over models. A model transformation converts a source model to a target model preserving their conformance to the respective meta-models. When source and target models are at different levels of abstraction, the transformation is referred to as *vertical* transformation; on the other hand, for *horizontal* is intended the transformation between two models defined at the same level of abstraction [12].

At some point in the transformation chain, a deviation from the standard goal of generating actual code can be taken towards the derivation of test models for model-based testing (MBT) purposes (Fig. 1). Although model-based development (MBD) and MBT

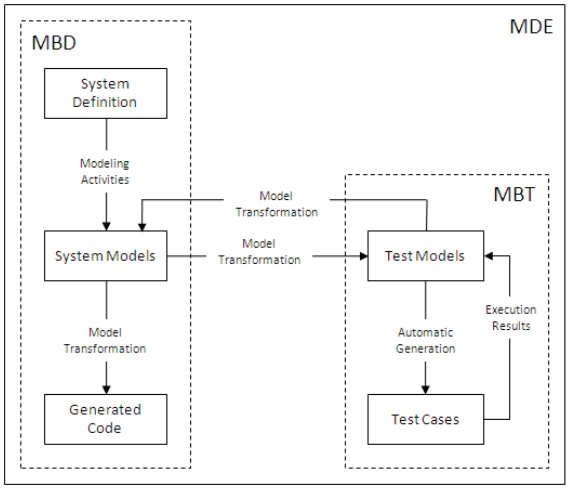


Figure 1: MBD-MBT Integration

are often seen as separated processes, in this work we show how to integrate MBD and MBT in a more complete MDE process. In particular, we propose to automatically derive test models by exploiting the behaviour information carried by the existing system artefacts through horizontal model transformations. The aim is manyfold, notably: *i)* to reduce the efforts devoted to testing tasks by automatically computing test models; *ii)* to ensure the consistency between the behaviour modelled for testing purposes and the one carried by the application design; *iii)* to produce useful feedbacks for the system design whenever generated tests diverge from

MOMPES '10, September 20, 2010, Antwerp, Belgium

the intended system behaviour.

In order to better clarify our proposal and the issues that have to be faced, this work presents a case study dealing with the derivation of QTronic Modeling Language (QML)[1] test models, needed for the generation of efficient test procedures, directly from a set of system models defined in xtUML [11].

The rest of the paper is structured as follows. Section 2 gives an overview on the involved technologies from MDE to model-reuse approaches. The paper culminates with section 3 in which the actual proposed approach is described and fully unwound in its details together with a case study. Moreover, the section contains a discussion of the results observed on the generated artefacts. Section 4 explores the related literature by giving some notions on similar works already proposed in automating model-based test-cases generation. The paper is concluded by section 5.

2. BACKGROUND

As mentioned above, typically model-based development and testing are intended to be distinguished phases; by considering the extent of models reuse for the purpose of a joint process, the most common approaches can be classified into three main categories, each of which entailing specific advantages and drawbacks, as clarified in the following.

Shared Model Approach.

It consists in using the same models for both MBD and MBT purposes. The system is generated from the models from which tests are also derived. Hence the system is tested against itself. The expected output part of the tests or verdicts can not be derived using MBT tools [14]. Testing a system using this approach, without manual derivation of verdicts, may be merely seen as testing of code generators or model compilers. A benefit of this approach is that models have to be created only once and verified twice, i.e. from MBD and MBT perspectives. It also facilitates models maintenance; a change to the generated system or test suite will modify the parent model, which will not only update the system but also help in deriving the updated test suite. However if there is a bug in the parent model, MBT would not detect such bugs as test cases are also derived from the same defected models. Moreover, it could be not possible to enjoy the separation of concerns, that is the usage of distinct languages dealing with domain-specific concepts.

Separate Model Approach.

Different models are used for MBD and MBT purposes. Verdicts can be automatically derived from test models using MBT tools [14]. In many situations testing yields better results when it is independent of the implementation. If compared to the previous solution, this approach can be considered as going in the opposite direction: expected behaviours (i.e. test models) are independent from the actual behaviour (i.e. system models). In this way it is possible to reach a strong separation of concerns between design and testing domain concepts. However, time and effort for creating and maintaining separate models rapidly increase, especially when changes made to the system require test cases to be updated; in fact, synthesis of impacts of design changes to test models could not be a trivial task. A factor which undermines the adoption of this approach is that both architects and testers can be considered as experts for their own respective domain only.

Hybrid Model Approach.

Test models are derived partially or fully from the system models or vice versa [2]. In case of the derivation of partial test models from the system models, a manual enrichment is needed to complete them by covering the aspects to be tested rather than the entire system's test details. By means of this solution the effort of defining models tailored to testing is reduced due to the (partially) automated generation of them. Moreover, the consistency between the concerns can be kept by construction through the transformation itself. As a drawback, adopting this solution demands for the implementation of a mapping between two distinct modelling languages, typically only partially overlapping and with their own semantics, which can result a not obvious task.

In this work we propose to exploit a hybrid model reuse approach for derivation of test models from system models by using horizontal transformations. As mentioned above, it is worth noting that despite the hybrid solution appears as the simplest, it poses several issues to be faced. For instance, test models need to be detailed and formal in order to allow a MBT tool to automatically generate a test infrastructure (i.e. test cases) in a correct way. In turn, this implies that system models have to be formal and detailed as well to provide useful information from which the test models can be derived. In the meantime, the selected languages for design and testing should have a comparable degree of expressiveness in order to enable a fruitful mapping; if not, either too much of the system information would be lost when transforming it for test usage, or a relevant manual input would be required to provide the necessary details not present in the design models, making the necessary effort comparable to the separate model approach.

The remaining part of this paper discusses a possible solution and the related challenges in mapping system design models toward corresponding testing tool artefacts to enable a hybrid approach devoted to application testing. In particular, both the approaches are UML-based and add their own action languages for supporting behaviour descriptions. Therefore, what is required is to map system design concepts to meaningful details from a testing point of view; our case demands for both a syntactical translation of UML state machines represented in different formats and a semantic mapping between the xtUML and QML action languages. The mechanism can be automated through the model compiler provided by the source design language: in essence, the application modelling platform provides a facility able to translate design models toward corresponding artefacts compatible with the testing tool format. Then, minor manual adaptations could be required to fix ambiguities that can not be solved at design time; in this respect, it is of critical relevance to successfully map also the graphical rendering between the tools to aid the testing expert in her/his task. Finally, the models can be input to the test generation engine to produce test cases.

3. PROPOSED APPROACH

3.1 Motivation

The approach we are proposing in this work aims at integrating MBD and MBT for a complete MDE development process. The industry de-facto modelling standard UML is typically too generic and lacks domain-specific features, thus UML profiles, which are extension mechanisms of the basic UML concepts, are used to address the specific needs such as code generation, testing or other domain specific purposes. xtUML is one such UML profiles, developed for executable models and code generation purposes. It has a rich set of constructs that allows model compilers to transform models into, virtually, any programming language. The entire

[1] http://www.conformiq.com/qtronic.php

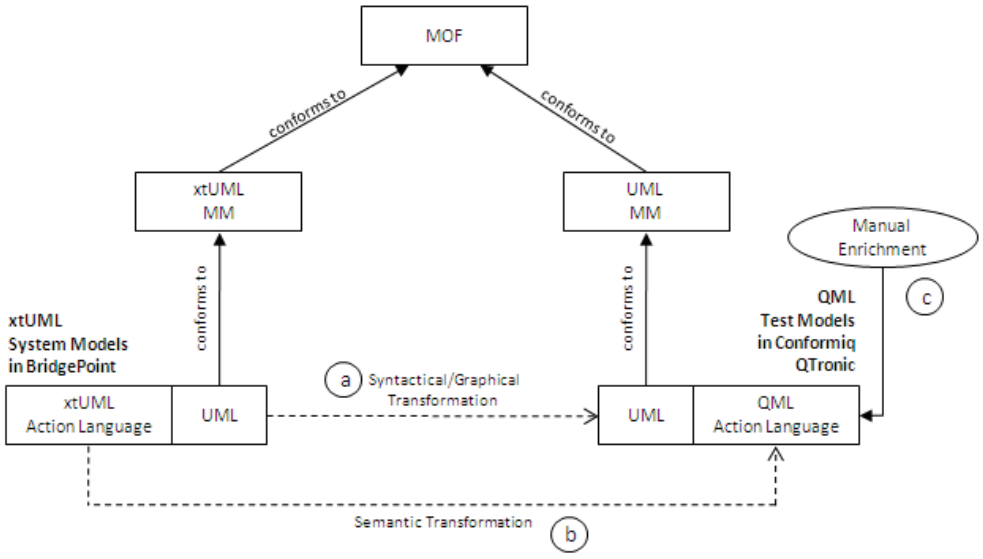


Figure 2: From xtUML System Models to QML Test Models

software application can be automatically generated from source models; however, the designer has to describe a relevant amount of details about the application and its behaviour. Therefore the testing phase could benefit of such modelling efforts by re-using the information specified at the design level for validation purposes. In order to enable such interchange, xtUML models have to be mapped towards an appropriate language from which it could be possible to generate test cases: in our case study we use Mentor Graphics BridgePoint [2] as MBD tool which uses xtUML for the definition of system models and Conformiq QTronic as MBT tool which uses QML for the definition of test models; the final goal is the development of a MDE environment which integrates and enables the cooperation between MBD and MBT tools.

Horizontal transformations between xtUML and QML are not straightforward due to differences in the respective meta-models and action languages. The graphical part of xtUML is in fact conforming to the xtUML meta-model while the UML portion present in QML models does conform to the UML meta-model. Nevertheless, being both xtUML and UML MOF-based, transformations between diagrams from xtUML to QML are reducible to transformations between MOF-based metamodels; xtUML state-machine diagrams can be transformed to QML state-machines with minor syntactical changes since the xtUML meta-model is based on the UML meta-model (Fig. 2.a). The textual part, consisting in the QML action language, is intended to be derived from the xtUML action language, which is platform and language independent (Fig. 2.b). Since QML expresses its UML portion using XMI files, it can be easily exported and imported in QTronic; in the same way, the textual action language part, which is saved in separate files, can be independently created or derived.

In the proposed approach, QML test models can be derived partially using model extraction and horizontal transformation from xtUML system models; completion of test models may still require minor manual enrichments (Fig. 2.c). xtUML's UML portion can account for UML part of the QML models in case of state-machine diagrams. Even though both xtUML and QML support similar types of UML model diagrams (i.e. state machines), it is preferable to derive only the necessary portion of models that allows to satisfy the targeted test criteria. Hence, not all the information present in xtUML models is relevant for testing purposes.

Our approach for MBT benefits from the fact that xtUML system models are detailed enough to be virtually considered as the system itself. Code generators and model compilers only attach language constructs (i.e. syntax and semantics) and optimization to these models to yield a working software application for simulation, analysis or execution purposes. Thus in a sense we are developing a black-box testing tool to perform white-box testing by providing a form of input model which is as 'good' as the system itself. Statement coverage, branch coverage testing techniques, which are supposed to work on models placed at different level of abstraction, can now work on the same virtual system.

3.2 Case Study

A simple microwave oven is used as sample model for the demonstration of the described approach. By adopting a separate model approach (see Sect. 2), from the same requirements specification document two different models have been designed, one for system modelling purposes through xtUML in BridgePoint and a second one for testing purposes through QML in QTronic. On the contrary, a shared model technique would have exploited the same system modelling documents for testing purposes also. By automatically transforming the xtUML system model into the appropriate QML test model, i.e. by implementing a hybrid technique, we aim at showing that the suite of test-cases derived from the generated test model, coming from a system model, is able to cover the whole set of functional requirements as the ones derived from the test model designed in QML in a separate testing task.

The system model designed in xtUML is composed by three components, i.e. oven, door and timer. The approach in xtUML is based on designing structural diagrams of the system, in terms of its components and interactions giving a static view, and then modelling the functional perspective by means of state-machines related to the components (generally one-to-one) and interacting among themselves. In our case, focusing on the functional perspec-

[2] http://www.mentor.com/products/sm/model_development/bridgepoint/

tive, the microwave oven system is composed by three components and therefore three state-machines: oven, door and timer (respectively a, b and c in Fig. 3).

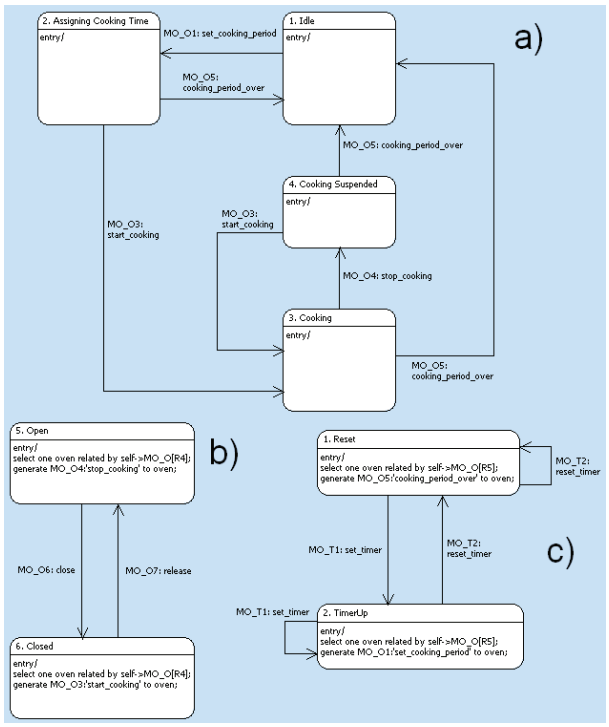


Figure 3: Oven State-Machines modeled in xtUML BridgePoint

Each state-machine is supposed to represent the functional behaviour of the related component and several state-machines interact in order to build-up the overall behaviour of the system. In the xtUML microwave model, *door* and *timer* state-machines (respectively b and c in Fig. 3) status affect the oven's behavior, since events generated by them cause transitions' triggering in the *oven* state-machine (Fig. 3.a).

At the same time, the microwave oven system has been designed in terms of a test model through QML in QTronic (Fig. 4). Looking at the two models, their different nature clearly arises; in fact, while the xtUML state-machines relate directly to the three structural components of the system (oven, door, timer), in the QML state-machines it is quite hard to individuate such a connection of behavioural meanings to structural components, since the model has been designed focusing on testing of the overall behaviour of the system. In the QML microwave model, a main state-machine (Fig. 3.a) is defined to model the behaviour of the oven, and each state contains an inner state-machine (Fig. 3.b) modelling the oven's cooking time. Differently from the xtUML model, no explicit information regarding the structural components of the system can be found in the QML model, even though many similarities can be noticed in terms of behavioural concepts since both models have been designed based on the same functional requirements specification.

3.3 Transformation Mechanism

The transformation mechanism has been developed using the BridgePoint UML Suite Rule Specification Language (RSL), which al-

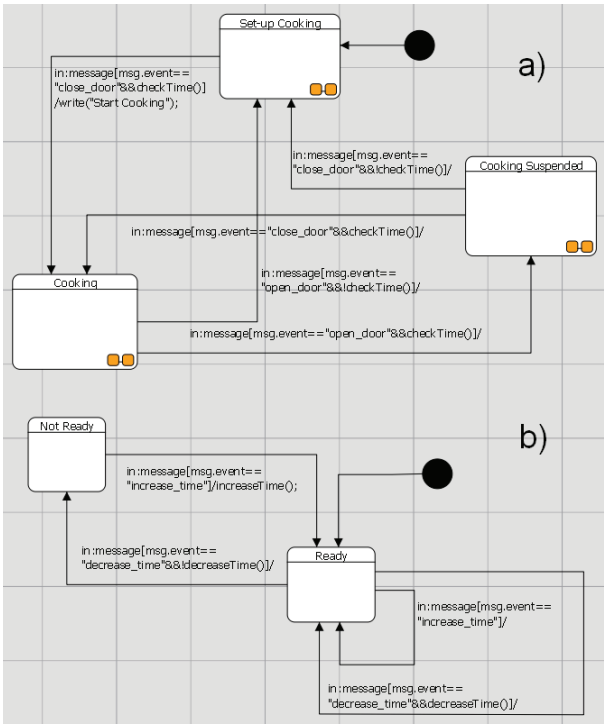


Figure 4: Oven State-Machines modelled in QML QTronic

lows to write translation rules that traverse the xtUML metamodel and create corresponding text that is emitted to files; in essence, it is a model-to-text transformation engine. Since in QML and xtUML a model is composed by a set of state-machines and action language code, the overall transformation functionality is achieved by the integration of two sub-functionalities, i.e. the transformation of the state-machines from BridgePoint to QTronic and the translation of xtUML action language to the QML action language.

The algorithm takes as input a xtUML model which we designed and exported through BridgePoint in a single .xtUML file and produces two files: a .xmi file represents the QML model (function *create_qtronic_state_machine()* in Fig. 5) in terms of state-machines, with eventual action language code placed within them, and their graphical rendering; a .cqa file (function *create_qtronic_actionLang_file()* in Fig. 5) describes the QML action language code needed for QTronic for the dynamic instantiation and automatic test-cases generation.

Syntactical Transformation.

In the xtUML model, both structural and behavioural concepts of the modelled system are described, but our focus in this work is the behavioural perspective, i.e. specification of instance state machines. Each xtUML state-machine gives birth to a correspondent state-machine defined in QML. Transformation issues concerning the translation of state-machine model elements from a UML modeling perspective are purely syntactical.

This statement is enforced by the fact that both languages adopt a subset of the UML definition of state-machines and moreover xtUML has a lower degree of expressiveness than QML in terms of pure state-machines, leaving aside any action language coding. A xtUML state-machine is mapped to a QML state-machine by it-

```
transformation_algorithm(xtUML_file){
  check environment_variables;
  define component_to_translate;
  define code_generation_directories;
  retrieve instance_state_machines from xtUML_file;
  create_qtronic_state_machine(instance_state_machines);
  create_qtronic_actionLang_file(instance_state_machines);
}

create_qtronic_state_machine(instance_state_machines){
  initialize qtronic_SM_file.xmi;
  for each instance_state_machine in instance_state_machines{
    create qtronic_state_machine_initial_state;
    for each state in instance_state_machine{
      navigate instance_state_machine to retrieve state_name;
      navigate instance_state_machine to retrieve state_action;
      if (state_action is not_empty){
        navigate instance_state_machine to retrieve state_actionBody;
        transform state_actionBody to qtronic_action;
        create qtronic_state_machine_state by state_name, qtronic_action;
      }
    }
    create qtronic_state_machine_initial_state_transition;
    for each transition in instance_state_machine{
      navigate instance_state_machine to retrieve transition_name;
      navigate instance_state_machine to retrieve transition_source_state;
      navigate instance_state_machine to retrieve transition_dest_state;
      navigate instance_state_machine to retrieve transition_action;
      if (transition_action is not_empty){
        transform transition_actionBody to qtronic_action
        navigate instance_state_machine to retrieve transition_actionBody;
        create qtronic_state_machine_transition by transition_name,
                                                  transition_source_state,
                                                  transition_dest_state,
                                                  qtronic_action;
      }
    }
    for each state in instance_state_machine{
      navigate instance_state_machine to retrieve state_graphics;
      navigate instance_state_machine to retrieve state_positions;
      navigate instance_state_machine to retrieve state_dimensions;
      create state_graphical_element by state_positions, state_dimensions;
      }
    }
    for each transition in instance_state_machine{
      navigate instance_state_machine to retrieve transition_graphics;
      navigate instance_state_machine to retrieve transition_positions;
      navigate instance_state_machine to retrieve transition_action;
      create transition_graphical_element by transition_positions,
                                             transition_dimensions;
      if (transition is polyline)
        navigate instance_state_machine to retrieve transition_segments;
      for each segment in transition_segments{
        navigate instance_state_machine to retrieve segment_starting_point;
        navigate instance_state_machine to retrieve segment_end_point;
        navigate instance_state_machine to retrieve segment_dimensions;
        create segment_graphical_element by segment_starting_point,
                                            segment_end_point,
                                            segment_dimensions;
      }
    }
  }
  finalize qtronic_SM_file.xmi;
}

create_qtronic_actionLang_file(instance_state_machines){
  initialize qtronic_actLang_file.cqa;
  for each instance_state_machine in instance_state_machines
    retrieve class_names from instance_state_machine;
    for each class in class_names{
      create system_in_port by class;
      create system_out_port;
    }
    for each class in class_names{
      create class_definition by class;
      create class_instance by class_definition;
    }
  }
  create main_method;
  finalize qtronic_actLang_file.cqa;
}
```

Figure 5: Transformation Algorithm in Pseudo-code

erating, for each xtUML element present in the state-machine, the following three-steps approach: (1) *navigating* the xtUML model driven by the xtUML meta-model definition, (2) *retrieving* the needed information to be mapped and (3) eventually *creating* the correspondent QML element. The algorithm performs the syntactical transformation from xtUML to QML state-machine through the function *create_qtronic_state_machine()*, given in pseudo-code in Fig. 5. A snippet of the actual code related to the transformation of a state element is depicted in Fig. 6.

```
.//
<states xsi:type="n1:InitialState"/>
.//
  .for each state in states
.//
.//  CHECK IF THE STATE HAS A STATE ACTION PROCEDURE
.//
    .assign actionBody = ""
    .assign class = ""
    .assign internal = ""
    .select one state_action related by state->
            SM_MOAH[R511]->SM_AH[R513]->SM_ACT[R514]
    .if (not_empty state_action)
.//
.//    CHECK IF THERE ARE ANY STATEMENTS IN THIS ACTION
      .select any statement related by state_action
        ->ACT_SAB[R691]->ACT_ACT[R698]->ACT_BLK[R601]
        ->ACT_SMT[R602]->E_ESS[R603]
      .if (not_empty statement)
        .select any entity related by statement
          ->E_GES[R701]->E_GSME[R703]->SM_EVT[R707]
        .if (not_empty entity)
          .assign actionBody = entity.Mning
          .select any sm related by entity
            ->SM_SM[R502]->SM_ISM[R517]->O_OBJ[R518]
          .if (not_empty sm)
            .assign internal =
          "entry{&#xA;in_${sm.Name} ev;&#xA;ev.event = &quot;
          ${actionBody}&quot;;&#xA;${sm.Name}_SM.send(ev);&#xA;}"
          .end if
        .end if
      .end if
    .end if
.//.assign class =
.//
<states internal="${internal}" name="${state.Numb}.${state.Name}"
                                 xsi:type="n1:BasicState"/>
.//
  .end for
.//
```

Figure 6: Snippet of the Syntactical Transformation

For each state in a xtUML state-machine, the model is navigated in order to reach the information needed for the creation of the XMI code representing the correspondent QML state. An actual initial state element has to be created and added in order to build a correct QML state-machine, since such element is not part of the xtUML meta-model, but on the other hand it is a compulsory element in the QML definition of state-machine. The initial state is then linked by a transition to the state that virtually acted as initial in the xtUML source model.

Concerning the translation of transitions from xtUML to QML, more manipulative transformation rules are needed. A transition in xtUML is defined in terms of *state-machine_ID: event_label*. In order to match to QML, it has to be manipulated to reach the following format:

port_type: in_state_machine_name[msg.val==’event_label’]

where the port_type has to be manually chosen at the end of the transformation process. In fact, they are all created by default as internal transition, i.e. only triggerable by internal events, in order to trace the xtUML concept of interaction among state-machines. Since the test cases generation in QTronic can fail due to deadlocks in state-machines (e.g. an internal transition which is never triggered), manual enrichment is needed in terms of selecting which ports are internal (port_type *this*) and which are triggered by external signals (port_type *in*). This manual step is requested since it is not possible to automate the decision due to the fact that such information is not present in the xtUML source model.

The QML representation of state-machines in the .xmi file does not only contain the syntactical and semantic descriptions of the state-machines, but also their graphical rendering information. Therefore, the transformation mechanism is also in charge of the translation from xtUML to QML graphics. The function *create_qtronic_state_machine()* (Fig. 5) is responsible for such translation and a fragment of actual code performing part of this transformation on rendering of transitions is depicted in Fig. 7. This snippet is only part of the transformation which addresses the rendering of transitions; the actual code performs, for instance, further manipulations in case of polyline transitions for creating and properly rendering their set of segments. Since graphics in xtUML and QML are expressed in different formats, the xtUML model has to be navigated to retrieve the graphical information of each element (states, transitions). This information is then manipulated by the transformation mechanism in order to create the appropriate QML graphical item in the .xmi file.

```
.for each transition in transitions
.//
.//  GET TRANSITION INFORMATION
    .assign transition_id = transition.QML_transition_id
.//
.//  GET THE GRAPHICAL INFORMATION FOR THIS TRANSITION
    .select any GD_GE_instance from instances of GD_GE
        where selected.OOA_ID == transition.Trans_ID
    .select one GD_CON_instance related by
                                GD_GE_instance->GD_CON[R2]
    .select many GD_LS_instances related by
                                GD_CON_instance->GD_LS[R6]
.//
.//  GET THE CENTRE OF THE TRANSITION
    .assign no_of_polyline_points =
                    (cardinality GD_LS_instances) * 2
    .assign centre_positionX = 0
    .assign centre_positionY = 0
    .for each GD_LS_instance in GD_LS_instances
      .select one DIM_WAY_instance related by
                                GD_LS_instance->DIM_WAY[R21]
      .assign centre_positionX = centre_positionX
                    + DIM_WAY_instance.positionX
      .assign centre_positionY = centre_positionY
                    + DIM_WAY_instance.positionY
      .select one DIM_WAY_instance related by
                                GD_LS_instance->DIM_WAY[R22]
      .assign centre_positionX = centre_positionX
                    + DIM_WAY_instance.positionX
      .assign centre_positionY = centre_positionY
                    + DIM_WAY_instance.positionY
    .end for
    .assign centre_positionX =
                    centre_positionX / no_of_polyline_points
    .assign centre_positionY =
                    centre_positionY / no_of_polyline_points
.//
.//  GENERATE INFO FOR THIS TRANSITION
    <items autolayout="false" transition=
                "//@statemachine.${count}/@transitions.${transition_id}"
                                        xsi:type="n1:DiagramTransition">
      <bodylocation align="0" valign="0"
                x="${centre_positionX}" y="${centre_positionY}"/>
```

Figure 7: Snippet of the Graphical Transformation Rule

Semantic Transformation.

The syntactical transformation between xtUML and QML is not enough. Issues concerning the translation between the two action languages is indeed a matter of semantics and therefore a semantic transformation is needed. First of all, the xtUML action language is defined only within the state-machine itself, while in QML, an apposite .cqa file is needed with the description of the state-machines' structure in a Java-based action language, and actions can be placed both in the .cqa file and in the state-machine definition contained by the .xmi file. We decided to map the xtUML actions directly to QML actions to be placed within the state-machines, while delegating to the .cqa file only the minimal amount of code needed to be able to make the QML state-machines operative. The function *create_qtronic_actionLang_file()* in Fig. 5 is in charge of creating the QML action language code file (Fig. 8) in terms of:

- Inbound ports: one for each state-machine created as in_*name*, where *name* is the state-machine's name retrieved from the xtUML model;
- Outbound ports: a common outbound port;
- Records: a structural definition for each defined port; a string field *val* is always created since it is in charge of carrying the string related to the event that would trigger a transition on the state-machine which owns the related port'
- State-machines definition: each state-machine has to be defined has a Java class extending the abstract class *StateMachine*;
- Instantiation of state-machines: in order to be used, defined state-machine classes need to be instantiated;
- Initialization of threads: a *main* method is created and contains the initialization of the state-machines instances in terms of threads.

```
system{
  Inbound in: in_Oven, in_Door, in_Timer;
  Outbound out: out_port;
}
record in_Oven{
  String event;
}
record in_Door{
  String event;
}
record in_Timer{
  String event;
}
record out_port{
  String val;
}
class Oven extends StateMachine {}
class Door extends StateMachine {}
class Timer extends StateMachine {}

Oven Oven_SM;
Door Door_SM;
Timer Timer_SM;

void main(){
  Oven_SM = new Oven();
  Door_SM = new Door();
  Timer_SM = new Timer();
  Oven_SM.start("Oven Thread");
  Door_SM.start("Door Thread");
  Timer_SM.start("Timer Thread");
}
```

Figure 8: QML Action Language Generated File

Both xtUML and QML action languages are complex and powerful in terms of expressiveness, therefore a complete mapping among them goes far beyond the scope of this work, which focus on the motivational proof of translating design models into test models. As a consequence, we take into account only the minimal set of actions needed for the models, i.e. xtUML and QML, to be comparable and the QML models to be operative, i.e. the generation of events for triggering of transitions. In xtUML, the generation of an event triggering a transition is defined by the following action: *generate eventID:'event_name' to state_machine*. In QML, a single code line of action language is not enough for the purpose, since, being a Java-based language, objects have to be first created and instantiated in order to be used. Due to the fact that QML transitions are triggered by messages received through inbound ports, we decided to initially create one of such ports for each state-machine. While the state-machine is statically created and instantiated during the creation of the .cqa file, ports, i.e. records with an obligatory string field for the event, are only defined at that time, while they are instantiated whenever they have to be used. For instance, the QML action language code related to the xtUML *generate MO_O3:'start_cooking' to oven* action (in state *Closed* of Fig. 3.b) would be (in state *Closed* of Fig. 9.b):

in_oven port;
port.event = 'start_cooking';
oven_instance.send(in_oven);
While *oven_instance* was instantiated in the .cqa file and can be used directly, the *in_oven* port has to be instantiated in the action body before it can be used (in the first line); eventually, the event *start_cooking* is sent to the state-machine *oven_instance* through the dedicated *in_oven* port.

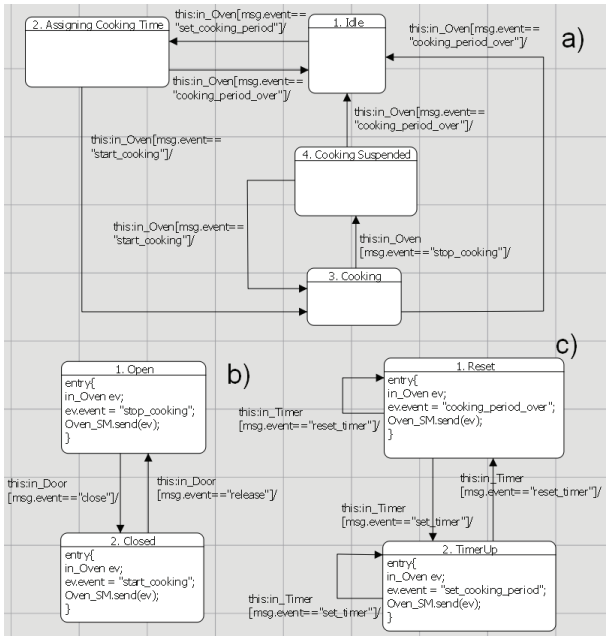


Figure 9: Generated Oven State-Machines in QTronic

3.4 Discussion

In the previous sections we gave our motivations and described in details both the proposed approach and its application in a real case study. In order to prove that our goals have been achieved a conclusive analysis step was performed. As explained previously, the microwave oven system has been modelled in xtUML and QML for different purposes (respectively system modelling and test modelling), but referring to the same functional requirements specification. What we want to demonstrate is the fact the QML test model automatically generated from the xtUML system model applying our approach is capable of fully covering the behavioural requirements of the system as well as the modelled QML test model. Patterns and coverage from automatically generated and manually modelled test models have resulted clearly comparable in terms of functional testing of the system's intended behavioural scenarios. This arises from the comparison of the test-cases generated from the two test models (generated and modelled), and confirms our initial motiviation and goal, since the approach can actually help in reducing the efforts devoted to testing tasks by automatically generating test models from system models. Moreover the consistency between the behaviour modelled for testing purposes and the one modelled at design level is ensured since test models are directly derived from the structural model of the system; this produces a three-fold advantage that would rarely occur in case of designing test models as a separate task in the MBD process:

1. behavioral testing patterns in terms of interaction between system components are produced and tested ;
2. patterns which may be feasible from a pure functional perspective but unfeasible from the system's structural definition (e.g. interactions among components) are not generated;
3. testing patterns that are directly traced to the system structural components gives the possibility for the system designers to get immediate and precise trace of eventual divergences from expected and actual behaviour in terms of involved structural components.

When dealing with more detailed system models, in terms of xtUML action language, the actual approach might not be able to produce test models with same degree of expressiveness due to the fact that the translation between the two action languages has not been fully implemented yet. Therefore we are going to include details coming from xtUML action language in order to generate more refined and powerful test models.

4. RELATED WORK

Automatic test cases generation from design specifications has been widely studied and many approaches have been proposed to address it. In this respect, the closest work to our proposal is presented in [1], where the authors illustrate the mapping of UML and SysML design models toward diagrams in the QTronic tool. Despite that work could appear as overlapping and even extending our technique, there are some relevant differences. First of all, the UML state machines are restricted in their expressive power and copied only with respect to their graphical rendering towards QTronic. The underlying assumption is that the functional representation for testing purposes exactly matches the behavioural description provided at system design level. As we have shown in our work, the design decomposition often does not match the functional view of the application, even for simple case studies; therefore, we exploit the expressive power of xtUML to grasp the functional abstraction of the system behaviour from its design specification. Of course, we try to preserve the graphical information as well, though we consider it necessary but not sufficient. Another important distinction between the techniques is that [1] adopts a text-based translation of tool formats, namely from MagicDraw to QTronic, by means of Python scripts. This entails that whenever the source model streaming format would change the transformation scripts would be corrupted. On the contrary, we propose a model transformation approach based on the xtUML metamodel.

In [13] UML activity diagrams are intended as design specifications and a set of test cases are indirectly generated by refining an initial wider set of randomly generated test cases accordingly to a given activity diagram and coverage criteria. Test suites with a given coverage level can be also generated from UML state charts as described in [6], where tests are derived as solutions to a planning problem solved by a tool that takes as input the state chart elements mapped to the STRIPS planning language. Also in this case, the generation is not directly performed on the design specifications, but rather from behavioural models derived a priori. Another effort in this direction is presented in [15], where the Automated Test Case Generation based on Statecharts (GTSC) environment is presented; its main feature is the automated generation of test sequences from both state chart- and finite state machine- based behavioural models.

Model transformation technology of MDA has also been used to address these issues; in [9] a method for generating unit test cases from platform-independent models is presented and achieved through a set of model-to-model and model-to-text transformations. In order to perform more extensive system testing, behavioural models may not be sufficient. An integration of formalised in-

formation derived from UML use-case and sequence diagrams is presented in [16] and describes the process of generating test cases from a System Testing Graph (STG), which is the result of the integration of the Use-case Diagram Graph (UDG), derived from the UML use-case diagram, and the Sequence Diagram Graph (SDG), derived from the UML sequence diagram.

More theoretical approaches have also been proposed; in [5] the authors aim at generating test cases by combining data abstraction, enumerative test generation and constraint-solving in order to broaden the applicability of model-based test generation tools to a system regardless of its size. Further information is needed for specific testing criteria, such as boundary-based coverage; a combination of UML state machine diagrams, class diagrams and Object Constraint Language (OCL) expressions is used in [18] to automatically generate test data input partitions by correlating OCL pre/post-conditions of operations and guard conditions of the state machines. This approach is supported by the model-based testing tool ParTeG [17].

Another popular approach to solve the test cases generation problem is to use a model-checker. This is done by producing test cases through the formulation of the test generation problem as reachability problem. The selection of test cases is driven by the coverage criterion and such algorithm, proposed in [7], is able to generate a test suite covering all feasible coverage items. Other approaches based on model-checking use coverage criteria derived from control-flow and data-flow of state-charts and, as in [8], formulate the problem of test generation as finding counterexamples during the model checking of such state-charts. In [3] the authors show how an integrated use of state and sequence diagrams can be used for deriving a reference model, which shall be the input for automatic derivation of test cases.

5. CONCLUSIONS

MDE aims at exploiting models in all the phases of the software development lifecycle; in this respect, design models can be (partially) reused for testing purposes. The major advantages are the reduction of efforts for re-modelling the system for testing, the preservation of coherence between design and testing models by construction, and an improved management of application evolution.

This paper presented a hybrid testing technique for re-using design models in an environment able to generate test cases from appropriate artefacts describing a system for testing purposes. Since design decomposition does not necessarily match the functional description of the application, a transformation able to grasp system features from design models and map them toward the corresponding functional representation is required. The feasibility of such a solution has been illustrated together with possible advantages, notably the reduction of efforts for test design and maintenance. Despite the initial results are quite positive, the approach has been validated against small case studies, therefore future investigations will encompass a thorough proof of the technique for industrial sized systems. Furthermore, we are currently investigating the inclusion of additional details coming from xtUML in order to generate more refined test models. Finally, we are planning to map the generated test cases back to xtUML; in fact, in that way we could exploit available facilities for model simulation and generation of test scripts toward different implementation platforms, thus closing the loop of the integration between the domains. In this respect, we are evaluating the adoption of an alternative transformation language, more suitable for such a wider task.

6. ACKNOWLEDGMENTS

We would like to thank Athanasios Karapantelakis for his support and fruitful discussions on QTronic related features.

7. REFERENCES

[1] F. Abbors, T. Pääjärvi, R. Teittinen, D. Truşcan, and J. Lilius. Methodological Issues in Model-Based Testing. In *Model-Based Testing of Reactive Systems*, pages 281–291, 2004.

[2] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer-Verlag New York, Inc., 2007.

[3] A. Bertolino, E. Marchetti, and H. Muccini. Introducing a reasonably complete and coherent approach for model-based testing. *Electron. Notes Theor. Comput. Sci.*, pages 85–97, 2005.

[4] J. Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.

[5] J. R. Calamé, N. Ioustinova, and J. van de Pol. Automatic Model-Based Generation of Parameterized Test Cases Using Data Abstraction. *Electron. Notes Theor. Comput. Sci.*, pages 25–48, 2007.

[6] P. Fröhlich and J. Link. Automated Test Case Generation from Dynamic Models. In *In Proceedings of ECOOP '00*, pages 472–492. Springer-Verlag, 2000.

[7] A. Hessel and P. Pettersson. A Global Algorithm for Model-Based Test Suite Generation. In *In Proceedings of Third Workshop on Model-Based Testing*. Electronic Notes in Theoretical Computer Science 16697, 2007.

[8] H. S. Hong, I. Lee, O. Sokolsky, and S. D. Cha. Automatic test generation from statecharts using model checking. In *In Proceedings of FATES'01*, pages 15–30, 2001.

[9] A. Z. Javed, P. A. Strooper, and G. N. Watson. Automated Generation of Test Cases Using Model-Driven Architecture. In *In Proceedings of AST '07*, page 3. IEEE Computer Society, 2007.

[10] S. Kent. Model Driven Engineering. In *In Proceedings of IFM '02*, pages 286–298. Springer-Verlag, 2002.

[11] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] T. Mens, K. Czarnecki, and P. Van Gorp. A Taxonomy of Model Transformations. In *Language Engineering for Model-Driven Software Development*. IBFI, Schloss Dagstuhl, Germany, 2004.

[13] C. Mingsong, Q. Xiaokang, and L. Xuandong. Automatic test case generation for UML activity diagrams. In *In Proceedings of AST '06*, pages 2–8. ACM, 2006.

[14] A. Pretschner and J. Philipps. Methodological Issues in Model-Based Testing. In *Model-Based Testing of Reactive Systems*, pages 281–291, 2004.

[15] V. Santiago, N. L. Vijaykumar, D. Guimarąes, A. S. Amaral, and E. Ferreira. An Environment for Automated Test Case Generation from Statechart-based and Finite State Machine-based Behavioral Models. In *In Proceedings of ICSTW '08*, pages 63–72. IEEE Computer Society, 2008.

[16] M. Sarma and R. Mall. Automatic Test Case Generation from UML Models. In *In Proceedings of ICIT '07*, pages 196–201. IEEE Computer Society, 2007.

[17] D. Sokenou and S. Weißleder. ParTeG - Integrating Model-based Testing and Model Transformations. In *Software Engineering*, pages 23–24, 2010.

[18] S. Weißleder and D. Sokenou. Automatic Test Case Generation from UML Models and OCL Expressions. In *Software Engineering (Workshops)*, pages 423–426, 2008.

A New Modeling Approach for IMA Platform Early Validation

Michaël Lafaye, David Faura, Marc Gatti, Laurent Pautet

Telecom Paristech, LTCI
46 rue Barrault
75634 Paris Cedex 13, France
{lafaye,pautet}@telecom-paristech.fr

Thales Avionics, ACS/DTEA
18 avenue Maréchal Juin
92366 Meudon-la-Forêt Cedex, France
{david.faura,marc-j.gatti}@fr.thalesgroup.com

ABSTRACT

This past few years, avionics platform conception changed to integrated architecture, permitting one processor to host some applications, in order to reduce weight and space. But this method entails more complexity, especially in safety domain, while time to market tends to decrease, so new development processes are needed. Model-based approaches are now mature enough to design embedded critical systems and perform architecture exploration.

In this paper we present a new modeling approach allowing avionics platform description and dynamic simulation. This method aim at dimensioning the architecture according to the applications it has to process, and to achieve early platform validation.

General Terms

Performance, Design, Verification.

Keywords

modeling, avionics systems, real-time, simulation, AADL, systemC

1. INTRODUCTION

Avionics systems are critical real time systems, i.e. timing constraints have to be strictly respected at the risk of catastrophic issue. They are composed of applications, real-time operating system and hardware modules. Initially, avionics platform (hardware and operating system) were implemented as federated architectures, where one processing unit hosted one function. This relatively simple architecture was however costly in terms of space, weight and power consumption, but offers a simple approach regarding the certification.

In order to reduce these parameters, and also to reduce costs, the Integrated Modular Avionics (IMA) concept was developed in the 2000s. It defines integrated architectures, where one processor can host some applications, and so reduces the number of modules used in avionics platform. Following this evolution, suppliers developed network architectures, in which modules are interconnected and communicate through a deterministic network. However, aggregating applications in a few modules, and gathering communications in a central network entails an increase of complexity in avionics platform design, verification and certification processes. In the same time, time to market tends to decrease. These developments require an early modeling of the system to validate and maximize the use of the future platform, while ensuring the critical level required by current standards in aviation (DO-178B, DO-254, MILS-CC…).

To model this IMA platform and perform early validation, Model-Driven Engineering (MDE) approaches are now suitable to describe system high-level functionalities. Many projects aim at modeling these platform with Architectural Description Languages (ADL) as AADL [1] or MARTE [2], or with synchronous languages such Lustre or Signal [3,4]. However, they often focus on the applications description, model the hardware as connected blackbox components with a few properties, and perform static simulation. Moreover, there actually is no automated process for complete platform modeling and simulation.

We define a new modeling method, that aims at designing a complete avionics platform (hardware, operating system and the applications). It is a component-based approach, relying on two languages and taking advantages of both: AADL and systemC. AADL [5] is, as MARTE, an ADL particularly adapted for software architecture description [6,7], enabling the modeling of ARINC 653 embedded real-time system [8]. In view of the experience of partners who developed the ARINC 653 AADL annex, the ARINC 653 compliant runtime for AADL called POK, the Ocarina tool suite etc., we choose the AADL rather than MARTE. SystemC [9] is an IEEE standard widely used in industry for hardware platform description, and containing a simulation kernel for architecture simulation and exploration.

In this paper we present our new modeling approach. We first introduce IMA concept, then we detail our method, before concluding with perspectives of our work.

2. Integrated Modular Avionics platform

IMA concept introduced integrated architecture, allowing to reduce the number of different modules used for platform design. As illustrated in figure 2, an IMA platform is composed of avionics applications, embedded operating system and the underlying hardware. This latter is composed of several processing modules communicating through a deterministic network, the AFDX (Avionics Full DupleX).

MONPES'10, September 20, 2010, Antwerp, Belgium.

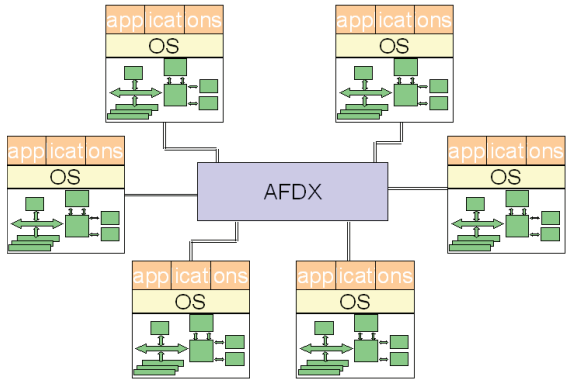


Figure 1. IMA platform

A processing module can host one or some applications at different criticality levels, it is then necessary to respect safety constraints. That's why OS ARINC 653 standard was defined, which specifies space and time partitioning. Figure 2 gives an overview of an IMA module embedding an ARINC 653 operating system. To ensure space partitioning, each application is enclosed in one or some partitions, isolating it from each other. Each partition is bound to a part of memory, so it only access its memory area. This partitioning prevents from failure propagation. Intra and inter-partitions communications are also defined by the standard to prevent failure propagation. To ensure time partitioning, each partition has its own execution time window, during which the application has access to all resources dedicated (processing, memory, dedicated I/O etc.).

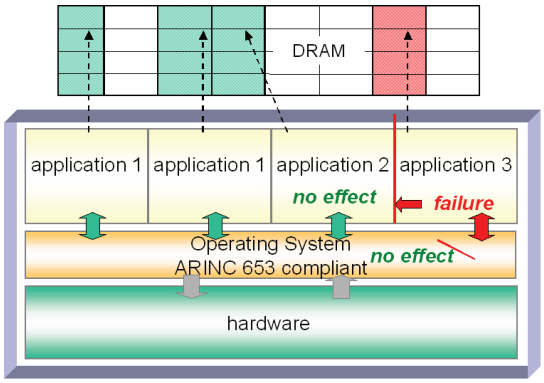


Figure 2. ARINC 653 spatial partitioning

3. IMA platform modeling

3.1 Overview

Model-Driven Engineering approaches are now mature enough to serve as a basis for building embedded systems and perform early validation. They are especially suitable for modeling the high-level architecture, that are the functional architecture (description of the functionalities offered by the system) and logical architecture (description of how the system is structured into logical components cooperating by communications) [10,11]. But at platform architecture level, these approaches describe both hardware and software as static blackbox elements with some properties.

Some projects [1,2,3] aim at building more accurate platform models, but they mainly focus on the software behavior, and model hardware as one or a few blackbox components without behavior information. They after simulate this latter statically. There is so no method to retrieve dynamic performances from the hardware to validate it according to the applications requirements.

Our method develops a new approach for avionics platform modeling. It aims at refining architecture description at platform decomposition level, specially the hardware. In order to refine this latter, our approach models and sizes it according to applications requirements. As we can see in figure 3, this approach consists of different tasks:

- system modeling;
- applications characteristics extraction;
- platform generation according to requirements;
- platform simulation with simulation scenario made with extracted stimuli;
- platform performances analyses.

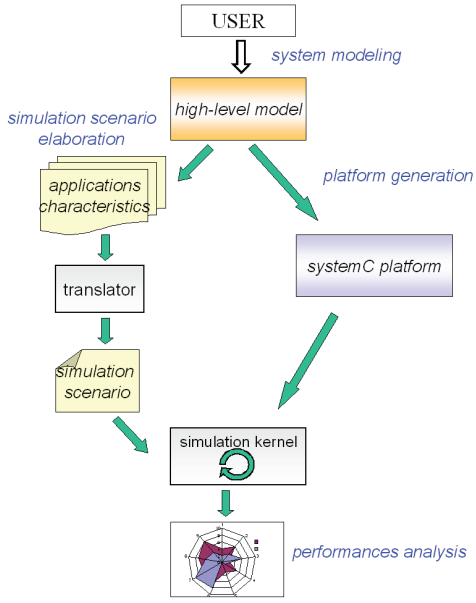


Figure 3. Modeling approach.

3.2 Application modeling

The description of the application consists of a set of characteristics (parallel code percentage, hit cache rate etc.), that will give the future platform requirements, and a set of instructions which will be translated into hardware stimuli. Figure 5 shows the two ways we can use to extract these characteristics from the application, and build a simulation scenario to simulate the platform.

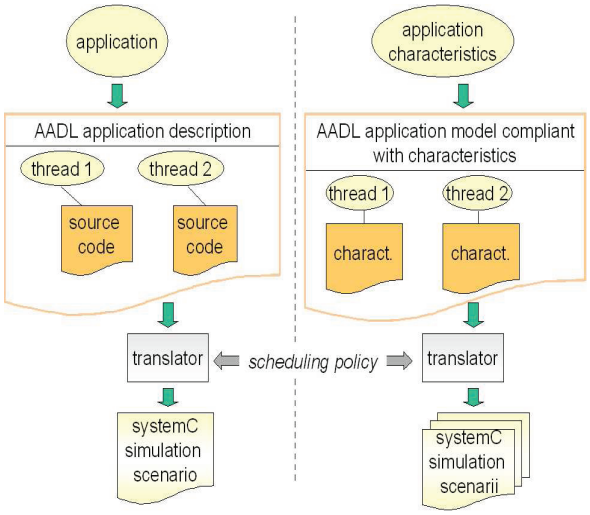


Figure 4. application modeling.

In the first way, -left path in the figure 4-, we have access to the application source code. In this case, the application is modeled with the AADL threads, which represent the ARINC 653 processes of the application. They are configured according to this processes (deadline, priority etc.). Threads are bound to a processor (or partition) and a memory (or part of memory), and ordered according to the scheduling policy defined for the application. Each thread as a reference to a part of the source code, from which we extract characteristics giving the requirements the platform has to match.

To elaborate the simulation scenario, we use a code profiling and application decomposition method. The figure 5 gives an overview of an application decomposition: it is decomposed into a logical function sequence (encoder, decoder…), refined into basic functions instructions (FFT, FIR…). Each basic function is composed of simple instructions (operator and operand(s)), that can be simple operations (add, div etc.) or memory access. The code attached to the AADL thread is parsed and instructions are extracted. We translate these latter into corresponding systemC instructions. For example, a "load" gives a systemC READ_COMMAND instruction.

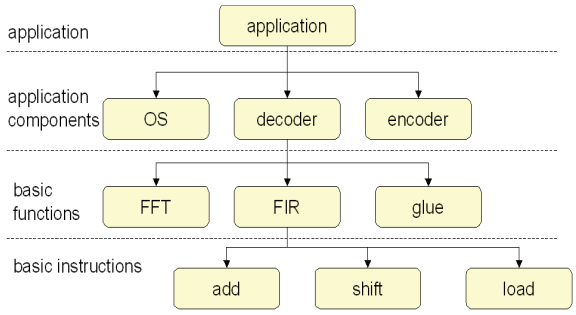


Figure 5. Application decomposition example

On the other way -right path on the figure 4-, we have not the source code of the application, so we can not extract basic instructions. However, some main application characteristics (sequential code percentage, scheduling policy etc) are given, and allow the elaboration of several constraint-random simulation scenario which fulfill application requirements. This method allows a hardware platform early validation without access to the application, but only with representative characteristics. It is less accurate than the first way, but is easier and faster.

3.3 High level system modeling with AADL

As we saw in the previous section, the application is described with AADL threads.
The real-time operating system is defined by some properties dispatched in the different hardware components. For example, scheduling policy is set in the processor module, partition security level is defined in the virtual processor, etc. To model an ARINC 653 operating system, we use the AADL 653 annex, and the method described in this article [8].

Each hardware component is modeled with the AADL corresponding component, or with the device element. Some components more complex, like network, are modeled as a sub-system containing some components. We model hardware component as a pseudo blackbox element, where behavior is not defined. We define interface information (ports, bus required if needed) and a few properties (memory size, bus transfer latency etc.). In order to refine those hardware properties, we created DRAM and cache component (that inherit memory element) to refine their read and write latencies, and we defined or completed some AADL property sets. We introduced behavior and specific properties, like cache hit rate for cache module, or refresh time for DRAM component.

The user models with the AADL the system corresponding to the platform, and choose which viewpoint(s) will be set when analyzing the platform. Viewpoints can be for example timing performance, power consumption or safety, and enable the platform investigation under different angles. We then extract the main characteristics of the application, and parse this AADL platform to retrieve properties, connections and deployment information, that will serve for systemC platform generation.

3.4 Platform integration by generation

In order to generate the systemC platform, we developed a database of configurable systemC components. These component have three parts: behavior, main properties and interface. For each of them, we identified the main characteristics and defined a configurable automata. At each state of this one are attached parameters corresponding to viewpoints and/or global parameters. Figure 6 shows an example of DRAM automata with some timing parameters (tRefresh, tRDC etc.) and one global parameters (burst_cpt).

For each platform element of the AADL model, we retrieve its information (properties, connections etc.) and configure the corresponding systemC behavioral model. Then, we connect all the modules to elaborate the refined systemC platform.

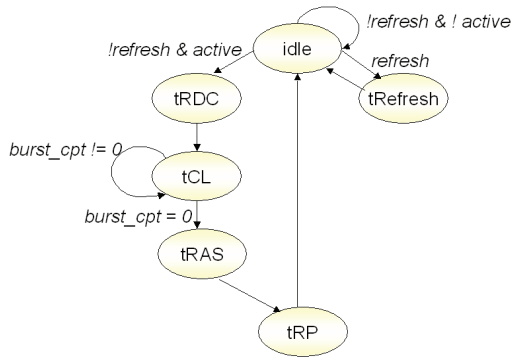


Figure 6. DRAM automata example with timing annotation

3.5 Platform simulation and results

Currently, this is a work in progress. However, we have already encouraging results. The AADL part of the process has been specified and is under development, while systemC main hardware components (processing unit, cache memory, dram and bus) have been developed (behavior, main properties and communication interface). In order to test and refine these elements, we implemented a minimum hardware platform with one or some instance of each of them. We also developed a systemC frame generator that simulates the platform, so we can observe the elements behavior and the platform communications.

Otherwise, to test our future platform, we defined a simulation and performances analysis method: to see if the hardware platform built is compliant with the application requirements, we perform a simulation using systemC kernel. It takes the platform generated, the viewpoint(s) set, and applies the simulation scenario. As we can see in the figure 7, the user can analyze the platform performances by examining performances graphs or simulation traces. Then, we can see if the platform matches the requirements corresponding to the viewpoint(s) set. If the hardware is not compliant with the applications requirements, we can investigate what is the problem, and try to refine or modify one or more components implementation.

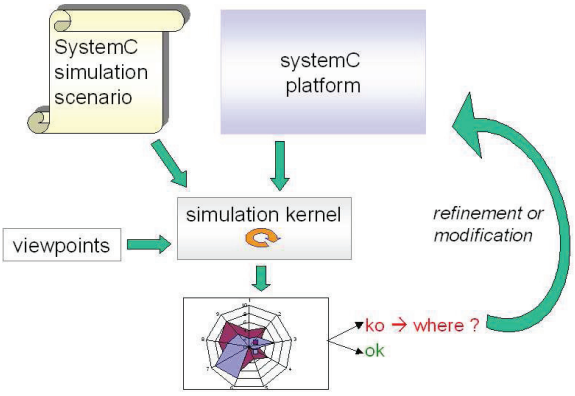


Figure 7. Platform simulation and performances analysis.

4. Conclusion

Current early platform validation methods center on software modeling, regarding the hardware as blackbox components which can't be dynamically simulated.

We have presented a new early validation approach, that aims at modeling a complete avionics platform, software and hardware (i.e. IMA modules and their interconnections as AFDX). Our method automatically generates hardware and simulation scenario to simulate it. It enables a dynamic simulation of the platform, and analyzes its performances according different viewpoints (timing, power consumption or safety). It takes advantages from the AADL, particularly adapted for software architecture modeling, and from systemC, industrial standard for hardware architecture description.

To validate the accuracy of our modelling methodology, we first model electronic evaluation boards. We will afterwards model a complete IMA platform to compare the model performances with the experimental results. Otherwise, we will connect with existing model-driven engineering methods and improve the platform development process.

5. REFERENCES

[1] Support for Predictable Integration of mission Critical Embedded Systems project (SPICES), 2009 http://www.spices-itea.org

[2] Model-Based Approach for Real-Time Embedded Systems development project (MARTES), 2007. http://www.martes-itea.org/

[3] C. Brunette, R. Delamare, A. Gamatié, T. Gautier, J-P. Talpin, "A Modeling Paradigm for Integrated Modular Avionics Design", IRISA report, 2005.

[4] Y. Ma, J-P. Talpin, T. Gautier, "Virtual prototyping AADL architectures in a polychronous model of computation", IRISA research report, 2007.

[5] AADL Portal at Telecom Paristech : http://aadl.telecom-paristech.fr/

[6] J. Hughes, F. Singhoff, "Développement de systèmes à l'aide d'Ocarina et Cheddar" *ETR09*, 2009.

[7] P. Dissaux, F. Signhoff, "the AADL as a Pivot Language for Analyzing Performances of Real Time Architectures", *4th European Congress ERTS Embedded Real Time Software,* 2008.

[8] J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, F. Kordon, "Validate, Simulate and Implement ARINC 653 systems using the AADL", *CM SIGAda Ada Letters*, 2009.

[9] Open SystemC Initiative. IEEE 1666: systemC Language Reference Manual, 2005. www.systemC.org.

[10] J.A. Estefan. "Survey of model-based systems engineering (MBSE) methodologies". Technical report, *INCOSE MBSE Focus Group*, may 2007

[11 Bernhard Schätz, Manfred Broy, Franz Huber, Jan Philipps, Wolfgang Prenninger, Alexander Pretschner, Bernhard Rumpe, "Model-Based Software and Systems Development – a white paper", 2004

Modular Synthesis of Mobile Device Applications from Domain-Specific Models

Raphael Mannadiar
McGill University
3480 University Street
Montreal, Quebec, Canada
rmanna@cs.mcgill.ca

Hans Vangheluwe
McGill University
3480 University Street
Montreal, Quebec, Canada
hv@cs.mcgill.ca

ABSTRACT

Domain-specific modelling enables modelling using constructs familiar to experts of a specific domain. Domain-specific models (DSms) can be automatically transformed to various lower-level artifacts such as configuration files, documentation, executable programs and performance models. Although many researchers have tackled the formalization of various aspects of model-driven development such as model versioning, debugging and transformation, very little attention has been focused on formalizing how artifacts are actually synthesized from DSms. State-of-the-art approaches rely on ad hoc coded generators which essentially use modelling tool APIs to programmatically iterate through model entities and produce the final artifacts. In this work, we propose a more structured approach to artifact generation where layered model transformations are used to modularly isolate, compile and re-combine various aspects of DSms. We demonstrate our technique by detailing the synthesis of running Google Android applications from DSms, and discuss how it may be applied in addressing the characteristic non-functional requirements (e.g. timing constraints, resource utilization) of modern embedded systems.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Software Architectures; D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures , information hiding, languages*

General Terms

Design, Languages, Standardization

Keywords

Multi-paradigm modelling, Model transformations, Language ripping and weaving, Application synthesis, Performance metric synthesis, Google Android

MOMPES '10, September 20, 2010, Antwerp, Belgium

1. INTRODUCTION

Domain-specific languages (DSLs) allow non-programmers to play an active role in the development of applications. This makes obsolete the many error-prone and time consuming translation steps that characterize code-centric development efforts; most notably, the manual mapping between the (often far away) problem and solution domains. Furthermore, due to their tightly constrained nature – as opposed to the general purpose nature of UML models, for instance, which are used to model programs from any domain using object-oriented concepts –, domain-specific models (DSms) can be automatically transformed to complete executable programs. This truly raises the level of abstraction above that of code. Empirical evidence suggests increases in productivity of up to an order of magnitude when using domain-specific modelling (DSM[1]) and automatic program synthesis as opposed to traditional code-driven development approaches [15, 11, 14].

Due to the very central part played by automatic code synthesis in DSM, we argue that structuring how models are transformed into code is both beneficial and necessary. Previous work realizes the said transformation by means of ad hoc hand-coded code generators that manipulate tool APIs, regular expressions and dictionaries [15, 11, 18]. In constrast, our approach employs modular and layered visual graph transformations whose results can readily be interpreted.

Section 2 briefly overviews related work. Section 3 introduces a DSL we have developed for modelling mobile device applications as well as the lower level formalisms and transformations that make up and produce the layers between the DSms and the generated applications. In Section 4, we present a non-trivial instance model of our DSL, every stage of the transformation to code and the synthesized application running on a Google Android [1] device. In Section 5, we compare the traditional approach to artifact synthesis to our own in the context of generating and presenting non-functional requirement related performance information from DSms. Finally, in Section 6, we discuss future work and provide some closing remarks.

[1]Note that we refer to domain-specific modelling as DS*M* and to a domain-specific model as a DS*m*.

2. RELATED WORK

In this section, we briefly discuss related research on three topics: artifact synthesis from DSms, modelling mobile device applications and integrating performance concepts in early development phases.

Most of current research in the general area of model-driven engineering focuses on enabling modellers with development facilities equivalent to those from the programming world. Most notably, these include designing/editing [9, 4, 3], differencing [2, 6, 12], transforming [8, 16], evolving [5] and debugging models [18]. More hands-on research has explored the complete DSM development process starting from the design of DSLs to the synthesis of required artifacts from instance models. In these works, DSms are systematically transformed to lower level artifacts by means of ad hoc hand-coded generators [15, 11, 14]. However, no allusions are made regarding model debugging, simulation, tracing or any other activity where it might be desirable to establish links between model and artifact. Wu et al. [18] recognized this need for the context of DSm debugging and proposed a generic grammar-based technique for generating DSL debuggers that reuse existing integrated-development environment facilities. Unlike previous work, mapping information from model to code is computed and stored during code generation. This information is then used to enable common debugging activities such as setting and clearing breakpoints and stepping into statements at the DSm level. Limitations of this work are that the mapping construction inevitably increases the complexity of the code generator and that the said mapping is not readily presentable to the modeller.

Several academic and non-academic efforts have investigated the modelling and synthesis of mobile device applications. In [14], a meta-model for modelling mobile device applications is introduced where behaviour and user interface elements are intertwined. In [15], a meta-model for home automation device interfaces with provisions for escape semantics[2] is described. The meta-model we introduce in Section 3 is inspired by these two formalisms and is in fact a combination and enhancement.

Finally, in [17], Tawhid and Petriu review past and current research on the benefits of elevating performance concerns to the early stages of development of software product lines (SPLs) as well as propose means to realize the said elevation. Their technique consists in annotating high-level models which are later transformed to performance models that lend themselves to analysis. The reasoning behind integrating low-level non-functional requirement related concepts so early on is that it is best not to realize that these requirements can not be met under the current design once implementation is well under way. The mindset of SPLs (where parameterizing high-level domain-specific concepts enables application synthesis) is of course very similar to that of DSM. Thus, means to reason about performance related concerns such as resource utilization and application response time at the DSm level are desirable.

[2]Means to extend the modelling language's expressiveness are built into the language itself.

3. META-MODELS & TRANSFORMATIONS

3.1 PhoneApps

Mobile device applications often require high levels of user interaction. It can thus be argued that behaviour *and* visual structure make up the domain of such applications. The *PhoneApps* DSL encompasses both of these aspects at an appropriate level of abstraction (see Figure 1a). Timed, conditional and user-prompted transitions describe the flow of control between `Container`s – that can contain other `Container`s and `Widget`s – and `Action`s – mobile device specific features (e.g., sending text messages, dialing numbers) – with each screen in the final application modelled as a *top-level* `Container` (i.e. a `Container` contained in no other). With a series of graph transformations, PhoneApps models are translated to increasingly lower level formalisms until a complete Google Android application is synthesized. Figure 1b gives an overview of the hierarchical relationships between the meta-models in play. The following subsections overview each transformation[3].

3.2 PhoneApps-to-Statecharts

The first step in the synthesis of executable applications from PhoneApps models is the *PhoneApps-to-Statecharts* transformation which extracts the models' behavioural components. Rather than attempt to invent a novel way of transforming and generating code for complex behaviour, we use the extensively proven and studied formalism of Statecharts [10] as our target formalism. The behavioural semantics of PhoneApps models are fully encompassed in the edges between `Container`s and `Action`s and can readily be mapped onto Statecharts. Existing tools for Statechart compilation, simulation, analysis, etc. can be exploited to produce efficient and correct code. Figure 2 shows an example graph transformation rule[4] from a subgraph of a PhoneApps model to its equivalent in the Statechart formalism. When the full PhoneApps-to-Statechart transformation has run its course, every `Container` and `Action` has a corresponding *state*. These are connected via customized *transitions* according to the edges that connect their respective `Container`s and `Action`s. Note that the current range of possible behaviours of PhoneApps models requires only the expressiveness of timed automata. Future work will extend the formalism with notions of hierarchy and concurrency such that more powerful Statechart features (e.g., orthogonality, nesting) become required.

3.3 PhoneApps-to-AndroidAppScreens

After isolating and transforming the behavioural components of PhoneApps models to Statecharts, another trans-

[3]See [13] for more detailed descriptions of the steps that make up these transformations.

[4]Rules are the basic building blocks of rule-based graph transformations. They are parameterized by a left-hand side (LHS), a right-hand side (RHS) and optionally a negative application condition (NAC) pattern, condition code and action code. The LHS and NAC patterns respectively describe what sub-graphs should and shouldn't be present in the source model for the rule to be applicable. The RHS pattern describes how the LHS pattern should be transformed by the application of the rule. Further applicability conditions may be specified within the condition code while actions to carry out after successful application of the rule may be specified within the action code.

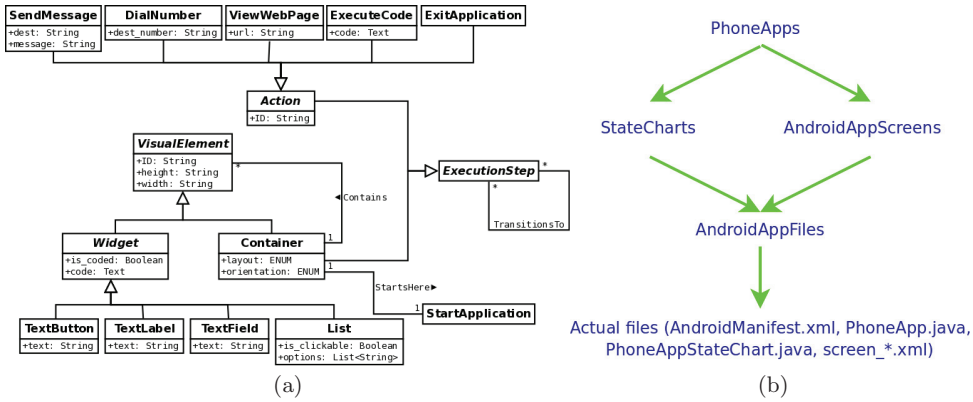


Figure 1: (a) The PhoneApps meta-model (as a Class Diagram); (b) A Formalism Transformation Graph [9] for PhoneApps.

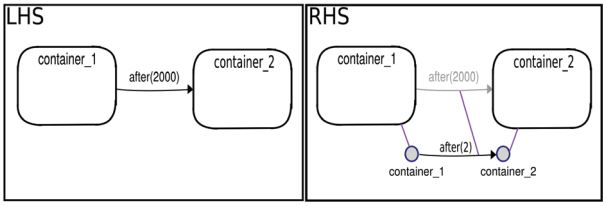


Figure 2: A PhoneApps timeout mapped to a Statechart timeout. The grayed out transition between `Container`s illustrates the marking of that transition as "*visited*".

formation is required to isolate and transform their user-interface and Google Android related components. The formalism we propose to encompass this information is *AndroidAppScreens* (see Figure 3a). When the full *PhoneApps-to-AndroidAppScreens* transformation has run its course, top-level `Container`s and `Action`s each have corresponding `Screen`s and `Act`s respectively. These are appropriately connected to some number of constructs that represent snippets of generated Google Android-specific code (e.g., XML layout code, application requirement manifests, event listener code). Figure 3b shows an example translation rule from a subgraph of a PhoneApps model to its equivalent in the AndroidAppScreens formalism.

The PhoneApps-to-Statecharts/AndroidAppScreens transformations clearly demonstrate the modular and layered nature of our approach to code synthesis. We improve upon the traditional ad-hoc hand-coded generator approach on numerous fronts.

First, the recurringly stated goals of simulation and debugging at the DSm level can be achieved by instrumenting the generated code with appropriate callbacks as in [18]. Unfortunately, in doing so, the code generator is polluted by considerable added complexity. In our approach, the complex task of maintaining backward links between models and synthesized artifacts is accomplished by connecting higher-level entities to their corresponding lower-level entities in transformation rules via *generic edges*[5]. These have minimal impact on the readability of the rules and their specification is amenable to semi-automation. The resulting chains of generic edges can be used to seamlessly animate and update DSms (or any intermediate models) during execution of the synthesized application [6].

Second, the aforementioned generic edges can aid in the debugging of the graph transformations themselves. Advanced DSM tools such as AToM^3 [9] which support step-by-step execution of rule-based transformations provide a limited but free transformation debugging environment where one can very easily observe (and modify) the effect of every single rule in isolation. Complex tasks such as determining what was generated from which model entity become trivial and don't require any further instrumentation. Once again, this is considerably easier, more modular and more elegant that lacing a coded generator with output statements and breakpoints.

Third, although in a finished product the inner workings that convert DSms to artifacts should be hidden from the modeller, it may be useful for educational purposes to see how higher- and lower-level constructs are related (as demonstrated in Figure 5). Both our transformation rules and the cross-formalism links they produce explicit these relationships.

Fourth, the multiple intermediate layers between model and artifact (and the links between them) provide a means to observe models from various "viewpoints". For instance, in the context of PhoneApps, to study only the behavioural aspects of a model, one could observe the generated Statechart in isolation from the DSm and the other generated artifacts.

Finally, the most important advantage of our approach is perhaps that it raises the level of abstraction of the design of "code synthesis engines". Rather than interacting with tool APIs and writing complex code, the task of implementing a code generator is reduced to specifying rela-

[5]Notice the purple, undirected edges between constructs of different formalisms in the figures describing rules.

[6]Future work will explore extending these animation capabilities to more advanced debugging activities such as altering the execution flow.

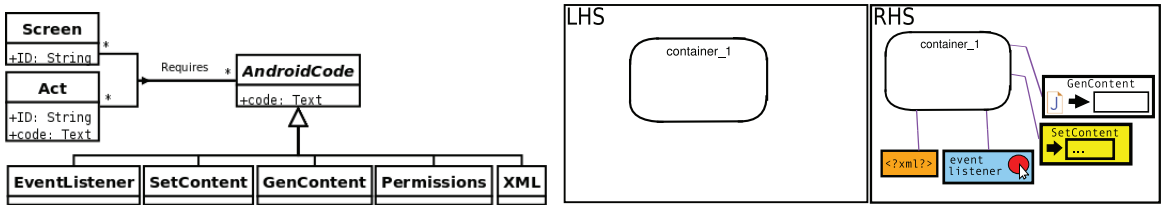


Figure 3: (a) The AndroidAppScreens meta-model; (b) Extracting information from a `Container` into new AndroidAppScreens constructs.

tively simple (graphical) model transformation rules using domain-specific constructs. In a research community that ardently encourages the use of models and modelling and more generally development at a proper level of abstraction, our approach to code synthesis seems like a natural and logical evolution.

3.4 AndroidAppScreens- and Statecharts-to-AbstractFiles

Benefits of keeping links between models and generated artifacts were discussed in the previous subsections. Following the same mindset, we introduce the *AbstractFiles* formalism. This trivially simple formalism serves as an abstraction of the actual generated files i.e., a model element exists for each generated file. Hence, rather than compile and output the previously generated AndroidAppScreens and Statechart models directly to files on disk, their compilation results in an instance model of the AbstractFiles formalism. An added benefit of this design choice is that the generated output for each file can be reviewed from within the modelling environment as part of the debugging process; there is no more need to locate files on disk and open them in a separate editor. Figures 4a and 4b show two example transformation rules from subgraphs of an AndroidAppScreens model to their equivalent in the AbstractFiles formalism. As for the transformation of the Statechart constructs to the AbstractFiles formalism, the output of a Statechart compiler is directed towards an AbstractFiles model element to be later output to a Java file on disk.

4. CASE STUDY: CONFERENCE REGISTRATION IN PHONEAPPS

We now present a hands-on example that demonstrates the successive intermediate representations involved in synthesizing a Google Android conference registration application from a PhoneApps model. Of particular interest is the intuitive mapping between each model and its counterpart(s) in the lower-level formalisms. The following will explicit the above-mentioned advantages of easier debugging and readability of the code and code synthesis engines that result from our approach. Figure 5a shows the modelled conference registration application CR in the PhoneApps formalism. There are 3 main use cases: (1) registering, (2) viewing the program schedule and (3) canceling a registration. The first is explored below.

1. The user sees the *Welcome* screen for 2 seconds and is taken to the *ActionChoice* screen;
2. The user clicks on "Register" on the *ActionChoice* screen and is taken to the *EnterName* screen;
3. The user enters his name, clicks "OK" and is taken to the *PaymentMethodChoice* screen;
4. The user clicks on a payment method. A text message containing the user's name and chosen payment method is sent to a hardcoded phone number after which the user is taken to the *RegistrationDone* screen;
5. The user sees the *RegistrationDone* screen for 2 seconds and the application exits;
6. The mobile device's operating system restores the device to its state prior to the launch of the conference registration application.

The output of the PhoneApps-to-Statecharts transformation is shown in Figure 5b[7]. Essentially, the application behaviour encoded in CR's transitions is isolated and used to produce an equivalent Statechart. Not visible are the *state entry actions* which effect function calls to generated methods that carry out tasks on the mobile device such as loading screens and sending text messages.

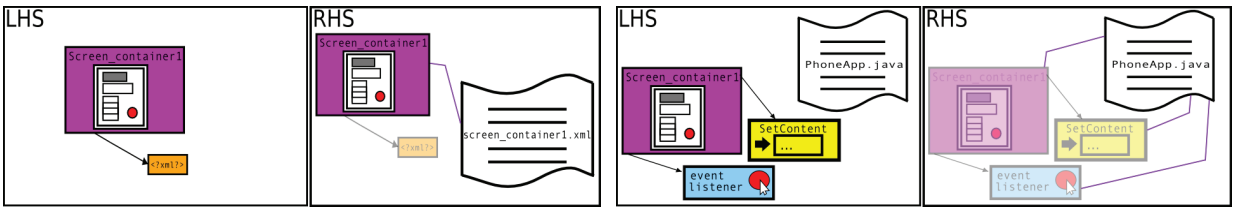
The output of the PhoneApps-to-AndroidAppsScreens is shown in Figure 5c. Essentially, the layout and mobile device specific aspects encoded in CR are translated to appropriate elements of the AndroidAppsScreens formalism.

Figure 5d shows the model after the AndroidAppsScreens-to- and Statecharts-to-AbstractFiles transformations have completed[8]. The two transformations output to a disjoint set of AndroidAppsFiles entities and can thus be run in parallel. Their results are presented together to illustrate the merging of the previously isolated conceptual components into a single, final target formalism.

The final step is the trivial transformation of the `ModelledFile`s to actual files on disk. The end result of this series of transformations is two-fold. First and foremost, a fully functional Google Android application that perfectly reflects the original PhoneApps model is synthesized as shown

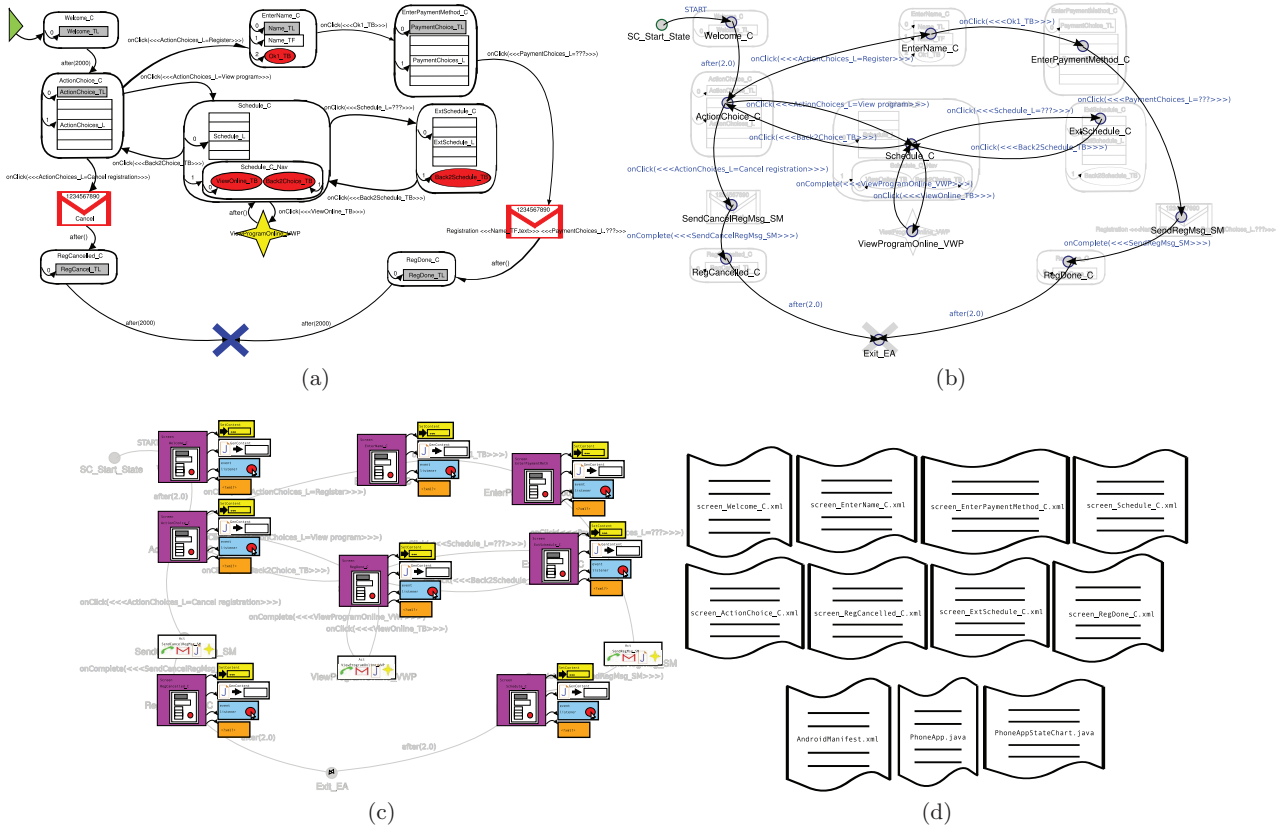
[7]For clarity, we refrain from reproducing the entire CR model and generic edges between it and generated constructs in Figures 5b, 5c and 5d. Instead, we overlay corresponding constructs.

[8]Remember that though they are hidden here, numerous generic edges connect the `ModelledFile`s to Statechart and AndroidAppsScreens constructs



(a) (b)

Figure 4: (a) Creating one `ModelledFile` per `Screen` to hold its XML layout specification; (b) Appending event listener and content initialization code to a `ModelledFile` of the main Java artifact "PhoneApp.java".



(a) (b) (c) (d)

Figure 5: (a) Conference registration in PhoneApps; (b) The CR model after applying the PhoneApps-to-Statecharts transformation; (c) The CR model after applying the PhoneApps-to-AndroidAppsScreens transformation; (d) The CR model after applying the AndroidAppsScreens-to-AbstractFiles and Statecharts-to-AbstractFiles transformations.

in Figure 6. Second, an intricate web of interconnections between model entities at different levels of abstraction is created. This web can be used for explanatory purposes (i.e., as we have used it to relate corresponding constructs in Figures 5b and 5c) or to ease debugging and simulation of DSms.

5. CASE STUDY: PERFORMANCE METRICS FROM PHONEAPPS

Our approach can also be beneficial in the context of modelling embedded systems and more specifically, in the addressing of their characteristic non-functional requirements. Although the mobile devices we target are indeed embedded systems, the Google Android API abstracts away traditional embedded system concerns. However, it is conceivable that information such as expected running time and battery usage may be required by the modeller. Indeed, the designer of a PhoneApps DSm might be faced with non-functional requirements pertaining to resource utilization and application response time. Hence, means to constrain or at least measure such performance related aspects should be provided. In this case study, we compare how such facilities could be implemented using both our approach to artifact synthesis and the traditional coded generator approach.

First, see Table 1 for a set of imaginary performance specifications for all Google Android devices. Second, let us assume that these specifications are stored and formatted such that they can be easily read from a modelling tool or a coded program. Finally, let us also assume that the modeller of a conference registration application is faced with the three non-functional requirements listed below:

- The full execution must require less than x% battery power;
- The full execution must require less than y seconds;
- The waiting time between two `Screen`s should never exceed z seconds.

Such requirements in conjunction with target platform specifications (ala. Table 1) could help a modeller discriminate between design decisions such as preloading a device with data versus downloading it at runtime, or communicating via email versus text messaging.

The task of generating performance models and/or statistics from DSms is analogous to that of generating any other artifact. Thus, the traditional coded generator approach would programmatically iterate over model entities to produce desired output (e.g., estimates of battery usage and running time[9]). One option would be to extend an existing generator (in this context, a generator that would synthesize complete Google Android applications from PhoneApps models) with performance measuring provisions. Another option would be to write a new generator from scratch and have it focus solely on extracting performance related information from models. Both approaches have merits and limitations. Although the latter will be more efficient, it may induce considerable code duplication since model traversal and information extraction will conceivably be carried out in a similar fashion than in existing generators. On the other hand, the former option might introduce undesired complexity and reduce the modularity of an already complex generator. In either case, providing more advanced features (e.g., "tagging" domain-specific constructs with battery usage or running time information at the DSm level, live performance data updates from DSm modifications) that exploit one- or two-way links between model and artifacts will require considerably polluting the generator's code.

[9]These estimates could be parameterized and plotted to illustrate pertinent bounds such as "the application respects requirement r provided SMS messages are restricted to x characters" or "local data as opposed to web data should be used if the said data is larger than y megabytes".

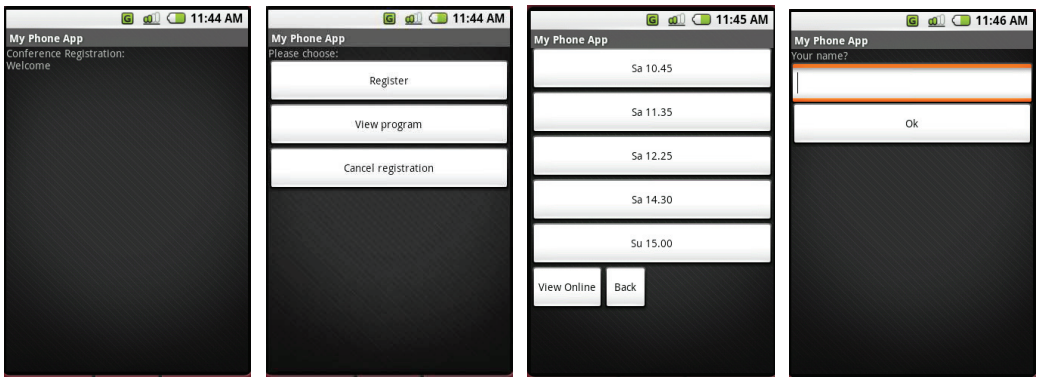
Analogous options for generating performance information using our model transformation-based our approach are fairly obvious: a new orthogonal model transformation could be created or existing model transformations could be refactored. For instance, *PhoneApps-to-* or *AndroidAppScreens-to-Metrics* transformations could be introduced with rules that count such things as the total number of `Widget`s on all `Screen`s along each possible execution path. Both options raise the same concerns as in the traditional approach; namely that PhoneApps-to- and AndroidAppScreens-to-Metrics will conceivably bare numerous similarities to the PhoneApps-to- and AndroidAppScreens-to-AbstractFiles transformations respectively whereas merging everything into a single transformation might result in a complex intermingling of concerns. Nevertheless, our model transformation-based approach will facilitate the creation of links between DSm and synthesized performance related artifacts thereby facilitating the implementation of the aforementioned advanced features.

Thus far, we have focused on "static" performance metrics in the sense that we assume that the desired information can be extracted from static DSms. However, it may be required to execute (or simulate) models to obtain more detailed and precise results (e.g., average and expected measurements as opposed to best and worst case theoretical bounds). Constructing such "dynamic" performance metrics would require that the generator or transformation rules instrument the synthesized executable artifacts (e.g., with code to increment a counter for every displayed widget). Instrumentation to output global information such as total measured running time could just as easily be produced by the traditional approach than by our own. However, to output more localized measurements (ideally on the DSm itself and possibly even during runtime) like the battery usage of a `SendMessage` entity or the time of entry of each `Screen`, links between DSm and artifact become a necessity. As we have repeatedly argued, such links are easier to specify and represent with our approach than with coded generators.

In sum, the numerous benefits of our technique to artifact synthesis not only apply to generating coded applications but also to the production and display of performance data useful in the modelling of embedded system applications.

6. CONCLUSION AND FUTURE WORK

We proposed a structured approach to artifact generation where layered model transformations are used to modularly isolate, compile and re-combine various aspects of DSms, while leaving behind a web of interconnections between cor-



(a) (b) (c) (d)

Figure 6: Screengrabs of the synthesized application running on a Google Android device emulator: (a) The *Welcome* screen; (b) The *ActionChoice* screen; (c) The *ProgramSchedule* screen; (d) The *EnterName* screen.

Function	Execution Time Range (s)	Battery Usage Range (%)
Tap Touch Screen	$[0.001, 0.003]$	$[0.0001, 0.0003]$
Load Screen	$[0.05, 0.07] * nb\_widgets$	$[0.001, 0.003] * nb\_widgets$
Send SMS	$[0.1, 0.3] + [0.1, 0.2] * \lceil sms.length \div 120 \rceil$	$[0.01, 0.03] * \lceil sms.length \div 120 \rceil$
Send Email	$[0.05, 0.1] * \lceil msg.length \div 1024 \rceil$	$[0.05, 0.07] * \lceil msg.length \div 1024 \rceil$
Load Web Data	$data.size \div 200 \frac{Kb}{s}$	$data.size \div 50 \frac{Mb}{\%}$
Load Local Data	$data.size \div 5 \frac{Mb}{s}$	$data.size \div 500 \frac{Mb}{\%}$

Table 1: Imaginary performance specifications for Google Android devices.

responding constructs from formalisms at different levels of abstraction. We argued that our approach improves upon the traditional ad-hoc coded generator approach to synthesizing applications from DSms. Discussed benefits include improving the domain-specificity and easing the development and debugging of code synthesis engines, providing clear pictures of the real and conceptual links between constructs at different levels of abstraction and simplifying the construction of inter-formalism mappings that enable advanced functionalities such as DSm animation, simulation, debugging and on-the-fly tagging.

The DSLs, transformations and case studies we presented provided empirical evidence to back our claims that (graphical) model transformations are a better means of generating artifacts (be they programs or performance metrics) from DSms than coded generators. However, our approach still requires some formalization. Since we essentially *ripped* and *woven* DSLs with our transformations, we believe that the first step towards this formalization is the study of the broader ideas and theory of DSL *weaving* and *ripping*, specifically during language design. For instance, combining some form of explicit DSL concept generalization relationship (e.g., `PhoneApps.ExecutionStep` *is a* `Statechart.State`) with higher-order transformations[10] could enable the automation of much of the above work (e.g., part or all of the PhoneApps-to-Statechart transformation could be generated automatically). As a final benefit of our approach, although (semi-)automatically generating transformation rules seems straight-forward, generating parts of a coded generator would not only require considerable effort but likely produce a complex and incomplete program that would be difficult to understand let alone complete and maintain. Thus, our technique is more amenable to (semi-)automation.

[10]Transformations that take other transformations as input and/or outputs.

7. REFERENCES

[1] Google android. http://code.google.com/android/.

[2] Marcus Alanen and Ivan Porres. Difference and union of models. In *Unified Modeling Language (UML)*, volume LNCS 2863, pages 2–17, 2003.

[3] Jean Bezivin. On the unification power of models. *Software and Systems Modeling (SoSym)*, 4:171–188, 2005.

[4] Alan W. Brown. Model driven architecture: Principles and practice. *Software and Systems Modeling (SoSym)*, 3:314–327, 2004.

[5] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing (EDOC)*, pages 222–231, 2008.

[6] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology (JOT)*, 6:165–185, 2007.

[7] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. 832 pages.

[8] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal (IBMS)*, 45:621–645, 2006.

[9] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and Systems Modeling (SoSym)*, 3:194–209, 2004.

[10] David Harel. Statecharts: A visual formalism for complex systems. *The Science of Computer Programming*, 8:231–274, 1987.

[11] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling : Enabling Full Code Generation*. Wiley-Interscience, 2008. 427 pages.

[12] Yuehua Lin, Jeff Gray, and Frederic Jouault. DSMDiff: A differentiation tool for domain-specific models. *European Journal of Information Systems (EJIS)*, 16:349–361, 2007.

[13] Raphael Mannadiar and Hans Vangheluwe. Modular synthesis of mobile device applications from domain-specific models. Technical report, McGill University, 2010.

[14] MetaCase. Domain-specific modeling with MetaEdit+: 10 times faster than UML. http://www.metacase.com/resources.html; June 2009.

[15] Laurent Safa. The making of user-interface designer a proprietary DSM tool. In *7th OOPSLA Workshop on Domain-Specific Modeling (DSM)*, page 14, http://www.dsmforum.org/events/DSM07/papers.html, 2007.

[16] Yu Sun, Jules Whit, and Jeff Gray. Model transformation by demonstration. In *MODELS*, volume LNCS 5795, pages 712–726, 2009.

[17] Rasha Tawhid and Dorina Petriu. Integrating performance analysis in the model driven development of software product line. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems (MODELS)*, 2008.

[18] Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-driven generation of domain-specific language debuggers. *Software : Practice and Experience*, 38:1073–1103, 2008.

Design Space Abstraction and Metamodeling for Embedded Systems Design Space Exploration

Marcio F. S. Oliveira[1, 2]
marcio@c-lab.de
Francisco A. Nascimento[1]
fanascimento@inf.ufrgs.br
Wolfgang Mueller[2]
wolfgang@c-lab.de
Flávio R. Wagner[1]
flavio@inf.ufrgs.br

[1]Institute of Informatics, Federal University of Rio Grande do Sul
Av. Bento Gonçalves, 9500, Bloco IV, Porto Alegre, Brazil
[2]C-LAB, University of Paderborn
Fürstenallee, 11, Paderborn, Germany

ABSTRACT
In this paper, we present a design space exploration (DSE) method for embedded systems, which represents the design space as a categorical graph product, in order to overcome the challenge of performing multiple DSE activities, such as task mapping, processor allocation, and software binding. Moreover, the method adopts a Model-Driven Engineering (MDE) approach, defining a design space metamodel to represent the categorical graph product and other DSE concepts, such as solutions, costs, and DSE activities. Furthermore, exploiting the MDE approach, we use model-to-model transformation rules to implement the design constraints, which guide and prune the design space. The method is applied to the design of a real-life application, and experiments demonstrate its effectiveness.

Categories and Subject Descriptors
I.6.5 [**Model Development**]: Modeling methodologies; J.6 [Computer-Aided Engineering]: Computer-aided design; C.3 [Special-purpose and application-based systems]: [Real-time and embedded systems]

General Terms
Design, Languages, Performance, Standardization, Verification

Keywords
Design Space Exploration, Model-Driven Engineering, UML.

1. INTRODUCTION
Modern embedded systems have increased their functionality through the composition of a large amount and diversity of hardware and software components, integrating complex Multi-Processor Systems-on-Chip (MPSoC). During the development of MPSoCs, a wide range of design alternatives arises from different design activities. The combination of alternative designs and stringent requirements unveils a complex design space, which the design team must explore under reduced time-to-market. At a system level, a Design Space Exploration (DSE) activity is performed, by which one looks for different solutions for the mapping between an application and an architectural platform, such that each solution corresponds to a different trade-off regarding design requirements and constraints [10].

From a broader point of view, DSE is performed always when an engineer must choose between multiple design alternatives, which arise at different levels. The DSE activity starts in the early development phases, when system requirements are used to define the earliest design artifacts. Distribution of responsibilities between components and definition of the interactions between them, as well as the algorithms to be executed, are some of the first design decisions. Most common design activities, which have automated support by some DSE tools, include definition of schedulable tasks, hardware/software partitioning, mapping of tasks to processors, binding of software to memories, and others. At a lower abstraction level, values for specific configuration parameters of system components, such as cache type and size, pipeline, bus width, and software precision, must be defined in order to tune the system implementation.

All those activities can be performed at different abstraction levels and have different levels of support by a wide range of DSE tools. Each tool requires its own evaluation method, input languages, and output format, so that setting up a tool chain to support all of those design activities is a complex task. Moreover, there is no common understanding about the execution order of those design activities, due to the strong interdependencies between them.

In order to cope with the complexity of the DSE process at earlier design steps, this work defines an abstract design space representation and the support of automated DSE activities by exploiting the Model-Driven Engineering (MDE) [17] approach. First, we define a design space abstraction as categorical graph products, dealing with the interdependence between design activities. We then specify a design space domain metamodel using the

MOMPES '10, September 20, 2010, Antwerp, Belgium

Eclipse EMF [6] project tools, so that it can be used in well-adopted MDE tools, during the development process. Exploiting the MDE approach, we use model-to-model transformation rules to specify the constraints, which guide the exploration process and prune the design space.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the definition of our design space abstraction and its metamodel. Section 4 describes how model-to-model transformation rules are used, in order to guide the design space exploration process. The supporting tools are introduced in Section 5. In Section 6 a case study illustrates the method, applying it to a real-life application design. Section 7 draws main conclusions and introduces future work.

2. RELATED WORK

After more than 20 years, DSE methods still represent an important research topic and many works can be found in the literature, as observed in reviews on DSE [8] and co-design [5].

The design space is the set of candidate designs, from where an engineer should select at least one candidate to implement the system. Each DSE approach has its own way to represent the design space, which depends on supported design activities and optimization algorithms. The representation tries to either expose some alternative properties or facilitate the optimization process.

The most common alternative to represent the design space is the definition of a direct mapping between graphs representing the application and architecture elements, as proposed by Blickle et al. [3]. Following [3], a specification graph consists of compositions of dependence graphs, which represent the application and the architecture models. In the specification graph, each dependence graph is mapped onto another one by mapping edges. The specification graph defines the design space and user constraints for allocation, binding, and scheduling. The term “activations” specifies the set of nodes and edges that are active in the specification graph. A set of active nodes and edges represents an implementation, i.e., a candidate or final solution. After its definition, the specification graph is codified in genes as an internal representation, such that a genetic algorithm may perform an optimized definition of the activation of nodes and vertices. Besides the mapping edges, equations define additional constraints usually related to the design candidate quality, which could be freely defined. However, to the best of our knowledge there is no tool implementing this method, providing either an open API or other mechanism to specify user constraints.

Erbas et al. [7] define the design space by equations, which are composed to form the function to be optimized. The variables are then encoded in genes for a Strength Pareto Evolutionary Algorithm (SPEA). As in [3], the constraints are equations that must be satisfied during the generation of candidates or after their evaluation. However, differently from [3], in [7] the constraints are fixed, according to the problem defined for the design space exploration tool, and restricted to define legal candidates.

A similar approach for direct mapping is implemented by MILAN [1], where the mapping model specifies the available mappings of each dataflow component in the application model onto components at the resource model (architectural model). The mapping is represented by one or more references between components in both the application and architecture models. These multiple references represent the design choices available during the design space exploration. Additionally, values for performance and power can be attached to the references in order to guide the exploration algorithm. The mapping model also determines the channel that implements the communication between dataflow tasks.

The DESERT tool [12], as our approach, also defines a design space metamodel. However, differently from ours, its metamodel represents concepts only for component composition. The design space metamodel consists of references to a library of Simulink components, each reference containing constraints with name and expression. The expressions are defined in OCL and can be freely specified by the user at the moment a component is inserted in the library. The DESERT tool focuses on the exploration of this library, and the constraints mainly impose restrictions on the composition.

The Koski tool [9] uses design constraints specified in the system UML model. Constraints related to real-time properties are defined as tagged values and used to remove unfeasible candidates. The constraints related to the architectural platform are defined as subclasses of a Constraints class, contained by the element to be constrained. Each subclass is a constraint associated to a field of the constrained class. The fields of the constraint class specify the bounds to be satisfied. The internal representation of the design space is not described. Mapping, allocation and scheduling can be directly defined in the UML system model and are used as a start point to the optimization process.

The DaRT project [4] proposes an approach based on MDA for the design of SoCs, where metamodels are defined to specify applications, architectures, and associations between hardware and software. Motel-to-model transformations are used to refine a Platform Independent Model into a Platform Specific Model expressed as a SystemC metamodel. It is possible to refactor an application model in order to better adequate it to a given architecture model. However, in DaRT there is no strategy for design space exploration based on these transformations, and the main focus is just the generation of code for simulation at TLM and RT levels from the models.

What all those methods have in common is the fact that the design space is restricted, according to the activity to be performed. Moreover, the generation of candidate designs is internally implemented, usually as a function that is programmed directly in the tool. As a result, no extension mechanisms are provided, requiring multiple tools to support each design activity. Moreover, except for the Koski and DESERT methods, for most approaches either the constraints set is restricted to previous constraints implemented by the tool or the method supports limited constraints constructs.

The method proposed in this paper overcomes those restrictions by defining a design space abstraction, as a categorical graph product [18]. Besides the automatic construction of the design space, performed by the product of graphs, this abstraction provides a common representation for multiple design activities. Moreover, the specification of a metamodel using a well-adopted technology allows us to exploit the MDE approach, as model-to-model transformation rules are used to implement any user constraints, improving the flexibility of the DSE tool.

3. DESIGN SPACE REPRESENTATION

3.1 Design Space Abstraction

Similar to most DSE and optimization approaches we explicitly define the design space as a mapping of graphs. However, differently from the common approach presented in [3], which is a manual mapping between semantically defined graphs, our approach uses the categorical graph product [18], automatically generating the mapping between graphs. These graphs are free of any specific semantics from the view of the DSE tools. In the following we define the representation of the design space.

Consider $G = <V, E, \partial_0, \partial_1>$ as a graph, where V is the set of all vertices of G; E is the set of all edges of G; $\partial_0: E \rightarrow V$ is the source function of an edge; and $\partial_1: E \rightarrow V$ is the target function of an edge. Let S be the set of graphs, where $G_i = <E_i, T_i, \partial_{0i}, \partial_{1i}> \in S$, $i = \{1..n\}$ and n is the number of graphs in S. This set is formed from graphs that are extracted from the design information, such as a task graph, an architectural graph, the communication structure of buses, and others. How these graphs are extracted from design models will be explained later. However, the specific semantics of each graph is not considered during the generation of the design space, for the purpose of design space abstraction. The specific semantics of these graphs is considered in the exploration rules, as explained in the next section.

The design space is a graph D resulting from the categorical graph product of the sequence of terms, which are all graphs in S. In this fashion,

$$D = G_i \times G_{i+1} ... \times ... G_n = <V_i \times V_{i+1} ... \times ... V_n, E_i \times E_{i+1} ... \times ... E_n, \partial_{0i} \times \partial_{0i+1} ... \times ... \partial_{0n}, \partial_{1i} \times \partial_{1i+1} ... \times ... \partial_{1n}>$$

represents the graph product between G_i, G_{i+1}... *and* G_n, where $\{\partial_{ki} \times \partial_{ki+1} ... \times ... \partial_{kn} \mid k \in \{0, 1\}\}$ are unambiguously induced by the dot product between vertices and edges, considering that any two vertices $(u_i, u_{i+1}, ..., u_n)$ and $(v_i, v_{i+1}, ..., v_n)$ are adjacent in D, if and only if u_i is adjacent with v_i in G_i, u_{i+1} is adjacent with v_{i+1} in G_{i+1}… and u_n is adjacent with v_n in G_n, $i = \{1..n-1\}$, where n is the number of graphs in S.

Each product of the sequence $G_i \times G_{i+1} ... \times ... G_n$ that constitutes D represents a design activity, such as task mapping, processor selection, processor allocation, voltage scaling selection, etc., such that vertices in D are design decisions and edges in D are design alternatives available at a specific vertex of D. The projection function $\pi_i = <\pi_{Vi}, \pi_{Ei}>: G_i \times G_{i+1} \rightarrow G_i$ is defined and returns the graph G_i involved in the product. Using this abstraction, a graph G' is a sub-graph of D and represents a candidate design.

Illustrating our approach, consider as a simple example two graphs T and P. Graph T (Figure 1-a) represents a task graph where the vertices are tasks and edges specify the data dependences between them. Graph P (Figure 1-b) represents the processing units and the allowed communications between them. Graph $T \times P$ in Figure 1-c is the categorical graph product between T and P, representing a design space for the task mapping design activity. The vertices in $T \times P$ represent design decisions (e.g. vertex $<T1, P1>$ specifies that task $T1$ should be mapped into processor $P1$), and edges identify the available design alternatives at a specific vertex (e.g. after selecting the vertex $<T2, P2>$, which maps task $T2$ into processor $P2$, the available design decisions are $<T3, P1>$, $<T3, P2>$, $<T4, P1>$ and $<T4, P2>$).

Using the categorical graph product abstraction, the design space exploration problem consists in searching for sub-graphs, which represent candidate designs, independent of the design exploration activities to be performed. Figure 1-d illustrates a selected sub-graph composed by vertices $<T1, P1>$, $<T2, P2>$, $<T3, P2>$, and $<T4, P2>$. The procedure to select vertices in order to produce the sub-graph must be guided by an optimization algorithm and a set of exploration rules, which search on the design space graph for candidate sub-graphs.

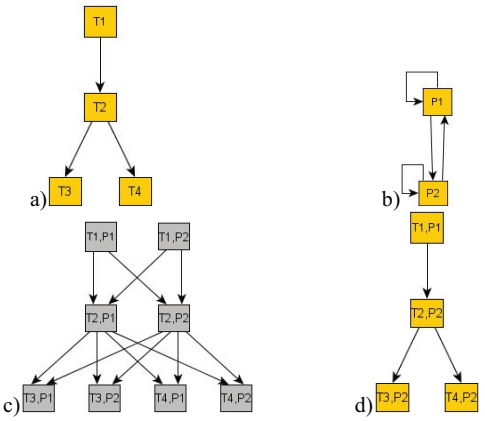


Figure 1. First example of a candidate design sub-graph using the categorical graph product

By applying the product between multiple graphs it is possible to produce a single design space graph for multiple DSE activities. As an example, consider a communication graph C (Figure 2-a), representing the communication structure (e.g. buses or Network-on-Chip) used to integrate the selected processors. Performing the product between the design space graph from Figure 1-c and graph C, the resulting graph (Figure 2-b) represents the mapping between tasks of graph T and selected processors of graph P and simultaneously the possible allocation of the processors selected from graph P (Figure 1-c) in the communication structure represented by graph C. This procedure can continue for multiple products, and the available DSE activities depend on which graphs are produced from the design and on the exploration rules that guide the exploration and prune the design space.

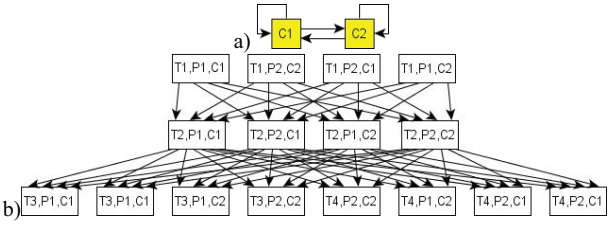


Figure 2. Example of design space for multiple products.

Using the categorical graph product as abstraction, design space exploration can be performed for multiple design activities simultaneously, as each product represents a design activity. Specific properties of this product, such as a restriction on the adjacencies, reduce the number of available alternatives, as the navigation on the design space is performed through the edges. Moreover, this representation overcomes the interdependence between design activities, as one vertex in the design space represents mul-

tiple design decisions at the same time. This abstraction also exposes the communication (dependencies) between elements and is well suited to combine the communication in multiple hierarchies, such as classes, task, processors, and systems.

3.2 Design Space Metamodel

To implement the design space abstraction in a DSE tool and integrate this tool to an MDE environment, we have defined a metamodel to represent the design space abstraction and provide elements to support DSE in a more general approach. The metamodel was defined considering MDE concepts that help an engineer during the DSE phase, such as orthogonalization of concepts, abstraction, and automation. It is important to highlight that this metamodel is proposed to represent a design space in abstract fashion and gauge the DSE process, instead of direct representing a system design. Therefore, it must be used to complement a design language, such as UML, Simulink, or any DSL. This design space metamodel is shown in Figure 3.

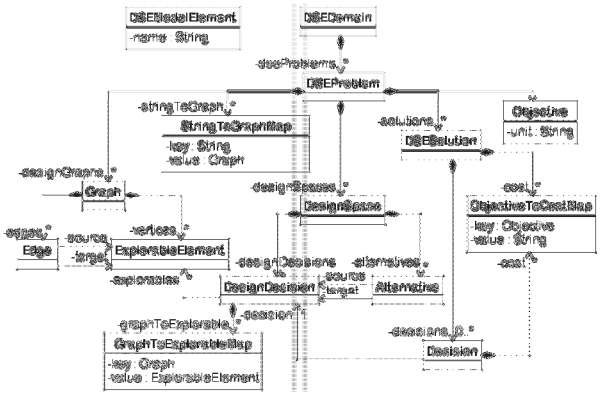


Figure 3. Design space exploration domain metamodel

The root container in this metamodel is *DSEDomain*, which is a container for all elements related to DSE. It inherits properties from *DSEModelElement* as all other elements in this metamodel. The generalization was omitted to keep the diagram clear. *DSEDomain* contains *DSEProblems*, which define a DSE scenario. *DSEProblem* contains a list of *DesignGraphs* extracted from design models. A *DesignGraph* contains vertices and edges, where vertices are *ExplorableElements,* and *Edge* represents the dependences between vertices. *ExplorableElement* is a reference to a design element from which the *DesignGraph* is generated. This reference is important to hook the DSE elements to the design model and allows the metamodel to be attached to multiple models, such as UML, Simulink, and others. Currently, this reference is implemented by holding the name of the design element as a field of *ExplorableElement* and using queries to find the instance of the design element in the design repository (e.g. UML model). This implementation could be improved, but it is important to evaluate factors such as performance, increase of dependence between metamodels, and traceability of design elements. *DSEProblem* also contains a list of *Objectives*, which are the values to be optimized, defined by their name and unit. *DesignSpace* represents the categorical graph product and contains the available *DesignDecisons*. *Alternatives* link the allowed *DesignDecisions*. *DesignDecision* is a tuple of n vertices from the *DesignGraphs*. It contains an instance of *DesignGraph* as a key and an instance of Vertex as a value, so that it can map a design decision to the *ExplorableElements* represented in the *DesignGraphs*. A list of *DSESolutions* represents candidate designs, which are sub-graphs of the design space graph. A *DSESolution* must conform to *ExplorationRules*. A *DSESolution* has its costs defined in the *ObjectiveToCostMap*, acquired from an estimation/simulation process.

The list of *ExplorationRules* defines rules, which guide the DSE tool and prune the design space. These rules are expressed as in any graph grammar, to efficiently handle the design space model. Actually, these rules can be implemented in any model-to-model transformation language, as QVTo, QVTr, Xtend, ATL, and others. As the current implementation of the metamodel uses the Eclipse EMF technology, the restriction is that the transformation engine/language must support ECORE based metamodels. However, EMF is largely supported nowadays. Because there are some mature or standard transformation languages, there is no reason to have a real instance of *ExplorationRules*. However, we are planning to use this element as a facade to configure transformation rules, depending on the *DSEProblem* configuration. Using model-to-model transformation rules, one can write constraints that directly handle the concepts of the design, such as processors, tasks, slots, voltage level, and others. Besides the constraints, these transformation rules are used to guide the DSE process, as they are used directly as a source to generate a candidate solution. This approach allows a generic use of the DSE tool for multiple DSE activities, where the user can freely define the transformations to be applied on the design space, which characterize the semantics of the graphs and the DSE activity.

4. EXPLORATION RULES

In our approach, exploration rules are model-to-model transformation rules, which receive an instance of raw *DesignSpace* (i.e. unconstrained design space) as input and generate a constrained *DesignSpace* instance as output. These rules are constraints to guide and prune the available design space, to reduce the exploration time and ensure the feasibility of a candidate solution. To ease the identification of rules that must be applied during the DSE process, we classify them into three categories:

A) Structural: These rules are applied to avoid illegal designs, which could appear as a sub-graph of the design space. Typical rules avoid double assignments of an element, e.g. different processors assigned to the same slot in a given communication structure or the same task assigned to different processors. Other rules may ensure that all tasks must be mapped, or at least one processor must be allocated. These rules can specify integration issues, such as two components that could not be integrated in the same system because of incompatibility issues.

B) Non-Functional: Even if a design is feasible, it can be invalidated when checked against non-functional requirements, which must be satisfied by the system. This way, these rules avoid the violation of requirements such as task deadlines, maximum delays, and maximum energy consumption. One challenge to the DSE process is to deal with system metrics, which cannot be partially evaluated. As such, the DSE process may postpone the design space pruning to a second step, after system evaluation, when it could filter the candidates from the design space. This procedure avoids selecting the candidate again, by removing design alternatives that may cause requirements violation.

C) *Pre-defined Design Decisions*: Designs usually start with pre-defined design decisions and previously developed components, and the selected platform may impose restrictions, which an engineer must respect. Moreover, engineer's experience may influence how the automated DSE process proceeds. Therefore, these rules are specified in cases where one needs to interfere on the DSE process through specific design decisions. Typical rules define specific task mapping, processor allocation, specific task or processor execution frequency, and others.

Considering this classification, the user of the DSE method is expected to define some rules for each category, which apply to his/her specific *DSEProblem*. The rules can be specified in any model-to-model transformation language that supports ECORE metamodels. A small effort is required to migrate the DSE tool to use a different language. First one needs to configure the DSE tool to load the appropriate file containing the exploration rules in the desired language. Then, it may be required to implement the interface needed to call the transformation engine, for the case that the configured engine does not support the desired language.

To alleviate the user effort, a set of typical rules was implemented and is provided as a library to the user. The current available rules are:

- *Duplicated Processor Assignment*: Avoids assigning different processors to the same slot in a given communication structure.

- *Multiple Assignments of a Processor*: Avoids assigning the same processor to different slots in a given communication structure.

- *Multiple Assignments of a Task*: Avoids assigning the same task to different processors.

- *Map All Tasks*: Ensures that every task in the system is mapped to at least one processor.

- *Lower / Upper Performance / Power/ Memory / Communication Value*: Defines the lower or upper values for performance, power, memory, or communication amount for a task.

- *Maximum Processor Occupation*: Defines an upper limit for occupation of a processor, so that a schedulability test can be satisfied.

- *Task Deadline Violation*: Verifies the timing properties of a task and removes the candidate from the population if there is a deadline violation.

- *Specific Task Mapping*: Defines that a task (active object/thread) must execute in a specific processor.

- *Specific Processor Allocation*: Defines that a processor must be allocated to a specific position /slot in a communication structure.

- *Specific Processor Selection*: Defines the processor type that must be selected to implement the candidate design.

- *Specific Task Execution Frequency*: Defines the frequency at which a processor must execute for a specific task.

- *Specific Processor Execution Frequency*: Defines the frequency at which a processor must execute.

Those rules are parameterized, and the user can pass the instance names, when calling a specific rule. Additional rules can be specified, referring to the instances of the DSE metamodel.

Furthermore, for the rules contained in that library, the DSE tool implements a generator, based on UML/MARTE [15] models, so that the user is not required to manually specify those exploration rules. To implement this generation, some UML/MARTE constructs are mapped to those rules, and the DSE tool automatically configures them. For this purpose, MARTE stereotypes and tags are applied in class, component, and sequence diagrams. Table 1 presents a simplified view of the mapping from UML/MARTE constructs to Exploration Rules. The *<<nfp>>* stereotype must be used to define the data type of the non-functional properties to be evaluated (including objectives to be optimized). *<<nfp>>* is required to define performance, power, memory, and communication data types used in the specification of a *<<nfpConstraint>>*. The tag type of the *<<nfpConstraint>>* stereotype must specify a required value for the property. These constructs can be applied to different elements, such as tasks and processors, and are mapped to the rules *Lower/Upper Performance/Power/Memory/Communication* and *Maximum Processor Occupation*. The *Task Deadline Violation* rule requires a task identified using the *<<rtUnit>>* stereotype, and its behavior identified using the stereotype *<<rtf>>* with its timing information, namely, type of occurrence, period, and relative deadline specified by the tags *occKind*, *period*, and *realDL*, respectively. The rule *Specific Task Mapping* is mapped from the stereotype <<Allocated>>, which must be applied to an association between a task identified by the <<rtUnit>> stereotype and a processor, stereotyped with <<hwProcessor>>. The connection between a communication structure identified as <<hwBus>>, through its ports to a processor, defines the *Specific Processor Allocation* rule. To define a *Specific Processor Selection* rule, the user just needs to instantiate a class stereotyped with <<hwProcessor>>. The stereotype <<hwClock>>, together with the frequency tag, can be associated with a task or a processor, in order to define the *Specific Task Execution Frequency* or the *Specific Processor Execution Frequency.*

Table 1. Mapping of MARTE profile to Exploration Rules

UML/MARTE	*Exploration Rule*
<<nfp>> <<nfpContraint>> {kind=required}	*Lower/Upper Performance/Power/Memory/Communication*
<<nfp>> <<nfpContraint>> {kind=required}	*Maximum Processor Occupation*
<<rtUnit>> <<rtf>> {occKind,period, relDL}	*Task Deadline Violation*
<<Allocated>>	*Specific Task Mapping*
<<hwBuss>> <<hwProcessor>> Port Connection	*Specific Processor Allocation*
<<hwProcessorr>>	*Specific Processor Selection*
<<hwClock>> {frequence}	*Specific Task Execution Frequency*
<<hwClock>> {frequence}	*Specific Processor Execution Frequency*

5. SUPPORTING TOOLS

The method presented in this paper extends the High-level Design Space Exploration tool (H-SPEX) [13], which implements the design space abstraction method as the DSE metamodel. The DSE metamodel was defined using the Eclipse EMF technology and is an ECORE model. The inputs to the DSE tool can be specified using the DSE metamodel editor generated using the EMF tools, or extracted from UML models and configuration files. The design information is extracted from UML models through model-to-model transformations, which extract the required information. Currently, this is performed by the MODES framework [11], which works as a design repository and provides the design information extracted from UML models.

The library of exploration rules is an extension in the Xtend language, from the openArchitectureWare [16]. The exploration rules can be specified using the Eclipse Modeling distribution, which contains the editor for Xtend.

The DSE tool also requires an optimization algorithm and an evaluation tool. We implemented two algorithms, in order to provide an optimization step during the candidates` generation: Crowding Population-based Ant Colony Optimization for Multi-Objective (CPACO-MO) [2] and Random. Actually, the DSE tool is not limited to these algorithms, and we are planning to integrate this tool to some optimization library to improve the optimization support with analysis and graphical features. The optimization is observed as a black-box transformation, which uses an API to communicate information between the transformation engine and the optimization algorithm. As evaluation tool, we use SPEU [14], a static analysis tool based on UML models, as it provides a fast evaluation step, which is the bottleneck of the DSE process. However, any other evaluation tool could be used, since the evaluation and DSE tools can exchange data. Currently, this is done by assigning the costs for a *DSESolution*, which can be performed by model-to-model transformations or using the API generated by the EMF tool.

6. CASE STUDY

6.1 A Design Space Exploration Scenario

In order to illustrate the proposed DSE method, this section presents a DSE scenario for a real application, concerning the automated control of a wheelchair. The application model consists of one use case diagram, one class diagram, 18 interaction diagrams, and one deployment diagram. Figure 4 illustrates the wheelchair system and its functional components.

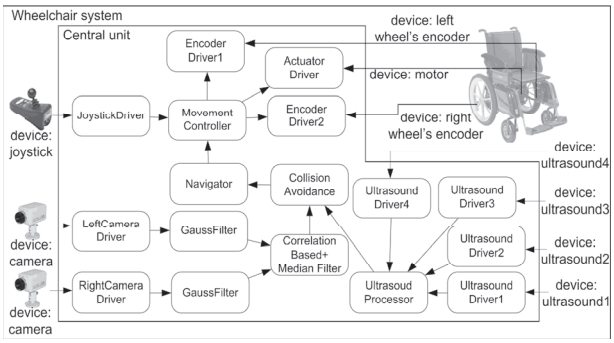


Figure 4. Wheelchair control system overview

In the automatic DSE process performed in this scenario, the DSE tool was configured to perform following design tasks: (i) definition of which objects are active or passive (runnables), among the 17 behaviors defined in the Interaction Graphs; (ii) deployment of the active objects to selected processors (up to 6 processors); (iii) allocation of the selected processors into a hierarchical bus with two segments; and (iv) processor voltage scaling with 4 distinct voltage levels. Exploring all these activities simultaneously, the DSE tool was configured to optimize the system in terms of performance (cycles), power (μWatt), energy (μJoules), total memory (bytes), and communication volume (bytes, transmitted in the bus).

The candidate population was found after 5,000 evaluations and represents the non-dominated set of candidate designs. Figure 5 illustrates these results. The best overall candidate must be selected after a trade-off analysis between the obtained estimations and based on some criteria, such as weights for the optimized objectives, or any other design feature.

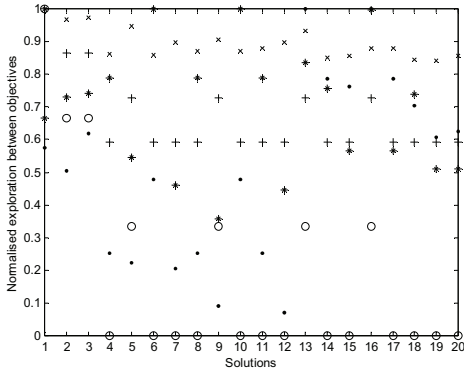
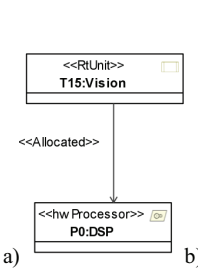


Figure 5. Normalized design space exploration results with five objectives: performance (+), power (.), total memory (x), energy (*), and communication (o).

The design space in this case study contains 2,064 alternative design decisions (vertices) and 334,080 edges, from where a set up to 17 (maximum active task distribution) vertices must be selected to define a candidate design solution (sub-graph). The unveiled design space presents more than 5.89×10^{41} alternative designs, considering an unrestricted design space (fully connected graph). However, in this proposal, edges guide the available alternatives, and constraints, specified as model-to-model transformation rules, are locally applied between the current vertex and its neighbors, thus pruning the design space and speeding up the DSE process.

Let a task drawn from the wheelchair case study be identified as T15, which implements a stereovision function (in Figure 4, T15 corresponds to the "Correlation-based + Median Filters" vertex), presenting heavy image processing algorithms. Figure 6-A shows a diagram specifying that the DSE tool must map Task 15 into the DSP processor P0, benefiting from the DSP processor architecture. Figure 6-B shows the Specific Mapping constraint, which is automatically applied when a deployment diagram is specified.



a)

```
DesignSpace specificTaskMapping(
    DesignDecision v1,
    DesignDecision v2,
    DesignSpace inDesignSpace
    String task,
    String processor):
    let t2 = v2.get("TASK"):
    let p2 = v2.get("PROCESSOR")
   ((t2 == getTask(task) ) &&
   (p2 != getProcessor (processor)) ?
  inDesignSpace.removeEdge (v1,v2):
null -> this;
```

b)

Figure 6. Sample of design constraints

Let consider a vertex from design space graph be the tuple <T13, P1, C1, V2>, which specifies that task T13 must be mapped to processor P1, while P1 must be allocated to communication bus C1 and execute T13 with voltage level V2. There are 48 alternatives at this vertex. Figure 7 illustrates a partial graph, representing the design space at this vertex, which is located at the center. The shadowed vertices around the vertex <T13, P1, C1, V2> in the centre are pruned nodes, and the white nodes are the alternative designs that satisfy all constraints.

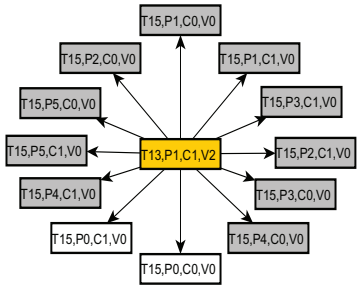


Figure 7. Sample of a partial design space graph

Applying the structural constraints presented in Section 4 and the sample design constraint here defined, the pruning process has reduced the design space by 83 % on the specific vertex, avoiding wasting time with unnecessary evaluations and unfeasible designs, thus focusing the search for an adequate solution on the most relevant design points.

6.2 Design Space Exploration Evaluation

Experiments were obtained with an Intel® Core™ 2 Quad 2.4 GHz processor with 2 GB RAM, running Microsoft Windows XP Professional with Service Pack 2 operating system. We performed 100 iterations (i) of each experiment. The number of generated candidate designs (m) was set to 50, thus resulting in 5,000 evaluations for each experiment, and the population size (p) was set to 20 (only the non-dominant candidates).

Figure 8 presents the total execution time for 100 iterations of the optimization algorithm, obtained when the number of objectives being explored varies from 1 to 7. The number of activities (a) was fixed at 2. The activities are task mapping and processor allocation. The objectives to be optimized are: performance, energy, power, program memory, data memory, total memory, and communication volume.

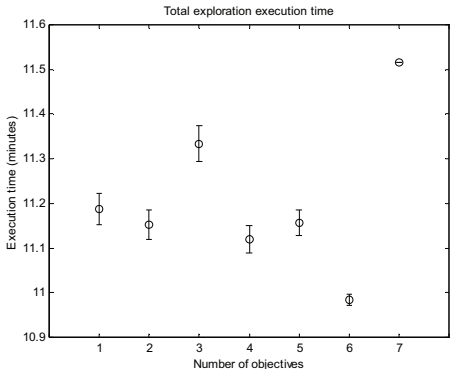


Figure 8. Exploration execution time for 1 to 7 objectives.

For one single objective, the circle represents the average value of the execution time for the seven distinct objectives, while the top and bottom bars represent the maximum and minimum execution time obtained in the exploration of these seven objectives, respectively. For two to six objectives, the circle represents the average value of the execution time of three randomly taken combinations of n objectives, where n is the number of objectives to explore (e.g. if n = 2, one possible combination is <performance, energy>), and the top and bottom bars represent the maximum and minimum execution time obtained for the exploration, respectively. In the case of seven objectives, the indicated value represents the execution time of the only possible single exploration combination for those seven objectives.

The results, varying from 11.0 to 11.5 minutes for the execution time, show an almost constant value for the exploration execution time, no matter how many objectives are being explored. It can be concluded that the proposed algorithm is robust regarding the increase of objectives being explored. Actually, for each additional objective, the exploration algorithm requires only four additional if-then-else commands and one variable assignment.

Table 2 presents results obtained when we vary the number of activities being explored with 100 optimization iterations. The number of objectives (k) was fixed at 2. The activities explored are definition of runnables, deployment, allocation, and voltage scaling, with their parameters equal to the previous DSE scenario presented in Section 6.1. The fixed objectives are performance and power.

Table 2. Exploration with 2 to 4 activities (*a*)

Activities	*Exec.time*	*# vertices*	*# edges*
Runnable and deployment	0.1478	172	2320
Runnable, deployment, and allocation	0.2175	344	9280
Runnable, deployment, allocation, and voltage scaling	1.0	1376	148480

In the second column, execution times were normalized to the longest one, i.e. when the DSE tool is executed with four activities. The numbers of vertices and edges account for the size of the graph product. Results in Table 2 show that design space exploration is highly dependent on the number of exploration activities taken into account.

7. CONCLUSION

This work has proposed a new design space abstraction based on the categorical graph product. This abstraction overcomes the challenge to deal with interdependences between design activities and provides a flexible representation for multiple design activities. Moreover, this representation decouples the exploration algorithm from the design space, always keeping the same exploration problem view regardless of the optimization algorithm, namely to find a sub-graph that optimizes a set of objectives. This abstraction is well suited for automatic exploration tools, which can take advantage from the MDE approach, as the graphs are easily handled by model-to-model transformation rules. Considering this fact, a DSE metamodel was defined, so that the design space could be easy handled by MDE transformation engines using their transformation rules. These rules are used to implement design constraints that prune the design space and generate the candidate design, thus improving DSE results. Moreover, the non-functional requirements are used to generate the additional transformation rules, which remove unfeasible designs from the design space, thus saving time with unnecessary evaluations.

The proposed method integrates the core of the DSE steps, namely candidate solution generation and design space pruning, into an MDE methodology and opens new opportunities to improve the DSE process through the exploitation of MDE. In the future, we are planning to extend the DSE metamodel to be more integrated to some OMG metamodels, such as OCL, QVT, and/or MARTE, so that a user could use OCL expressions and a standard type system, improving the specification of the exploration rules. Additionally, improvements in the metamodel may ensure its application in unplanned DSE scenarios, and additional experiments could be performed to quantitatively evaluate this proposal against others.

ACKNOWLEDGEMENTS

The work described herein was partly funded by the German Ministry of Education and Research through the SANITAS project(01M3088).

REFERENCES

[1] A.Agrawal et al. “MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems”. In: Proceedings of LCTES, 2001.

[2] D.Angus, “Crowding Population-based Ant Colony Optimisation for the Multi-objective Travelling Salesman Problem”. In: Proc. MCDM 2007, pp. 333-340.

[3] T.Blickle, J.Teich, L.Thiele. “System-Level Synthesis Using Evolutionary Algorithms”. In: Design Automation for Embedded Systems, v. 3, 1998. pp 23-58

[4] L.Bondé, C.Dumoulin, J.-L.Dekeyser. “Metamodels and MDA Transformations for Embedded Systems”. In: Proceedings of the Forum on Design Languages (FDL), Lille, France, September 2004.

[5] D.Densmore , R.Passerone , A.Sangiovanni-Vincentelli. “A Platform-Based Taxonomy for ESL Design”. IEEE Design & Test, v.23 n.5, p.359-374, September 2006

[6] EMF. Eclipse Modeling Framework. Available at: http://www.eclipse.org/emf. Accessed in May, 20010.

[7] C.Erbas, S.E.Erbas, A.D.Pimentel. “A Multi objective Optimization Model for Exploring Multiprocessor Mappings of Process Networks”. In: Proc. CODES+ISSS 2003, ACM Press, pp. 182-187.

[8] M.Gries. “Method for Evaluating and Covering the Design Space During Early Design Development”. Integration, the VLSI Journal, [S.l.], v. 38, n. 2, p.131-183, 2004.

[9] T.Kangas et al. “UML-based Multi-Processor SoC Design Framework”. ACM Transactions on Embedded Computing Systems, v. 5, n. 2, 2006.

[10] K.Keutzer, A.R.Newton, J.M.Rabaey, A.Sangiovanni-Vincentelli. “System-Level Design: Orthogonalization of Concerns and Platform-Based Design”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 19(12), 1523-1543, 2000.

[11] F.A.Nascimento, M.F.S.Oliveira, F.R.Wagner. “ModES - Embedded Systems Design Methodology and Tools based on MDE”. In: Proc. of 4th Int. Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES). Braga, Portugal, 2007.

[12] S.Neema, J.Sztipanovits, G.Karsai. “Constraint-Based Design-Space Exploration and Model Synthesis”. In: Proceedings of the 3rd International Conference on Embedded Software (EMSOFT), Philadelphia, USA, October 2003. LNCS 2855. pp 290-305

[13] M.F.S.Oliveira, E.W.Briao, F.A.Nascimento, F.R.Wagner. “Model Driven Engineering for MPSoC Design Space Exploration”. Journal of Integrated Circuits and Systems,v. 3, n.1, 2008.

[14] M. F. S. Oliveira, L.B.Brisolara, L.Carro, F.R.Wagner. “Early Embedded Software Design Space Exploration Using UML-Based Estimation”. In: Proc. of RSP’06 - 17th IEEE Int. Workshop on Rapid System Prototyping, Chania, Greece, 2006.

[15] OMG. UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE). 2007, available at <http://www.omgmarte.org>.

[16] openArchitectureWare framework. http://www. openarchitectureware.org

[17] D.C.Schmidt. “Guest Editor’s Introduction: Model-Driven Engineering”. Computer, Vol. 39, No. 2, Feb. 2006, pp. 25-31.

[18] P.M.Weichsel. “The Kronecker Product of Graphs”. Proceedings of the American Mathematical Society, vol. 13, n. 1, 1962.

View-Supported Rollout and Evolution of Model-Based ECU Applications

Andreas Polzer
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
polzer@cs.rwth-aachen.de

Daniel Merschen
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
merschen@cs.rwth-aachen.de

Jacques Thomas
Daimler AG
Group Research & Advanced Engineering
Böblingen, Germany
jacques.thomas@daimler.com

Bernd Hedenetz
Daimler AG
Group Research & Advanced Engineering
Böblingen, Germany
bernd.hedenetz@daimler.com

Goetz Botterweck
Lero – The Irish Software Engineering Research Centre
Limerick, Ireland
goetz.botterweck@lero.ie

Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
kowalewski@cs.rwth-aachen.de

ABSTRACT

When applying model-based techniques to the engineering of embedded application software, a typical challenge is the complexity of dependencies between application elements. In many situations, e.g., during rollout of products or in the evolution of product lines, the understanding of these dependencies is a key capability. In this paper, we discuss how model-based techniques, in particular, model transformations can help to reduce the complexity of such analysis tasks. To this end, we realised a representation of Simulink models based on the Eclipse Modeling Framework (EMF). The resulting integration allows us to apply various model-based frameworks from the Eclipse ecosystem. On this basis we developed a view that increases the visibility of functional dependencies, which otherwise would have been hidden due to a lack of abstraction in the native Simulink representation. The provided analysis framework comes in handy, when such a model has to be modified. Consequently, the developer is supported in reusing existing models and avoiding errors. The concepts and techniques are illustrated with a running example, which is derived from a real industry model from Automotive Software Engineering.

Keywords

Model-based development, variability, Matlab Simulink, automotive software, model transformation, ATLAS Transformation Language (ATL), Epsilon Translation Language (ETL)

MOMPES '10, September 20, 2010, Antwerp, Belgium

1. INTRODUCTION

Model-based development (MBD) of ECU applications has become an established methodology for automotive electronics. The behaviour of the application (e.g., exterior light control, 12V power network management) is first modelled using, e.g., Target Link, then ECU code is automatically generated. Finally, the code is integrated into the runtime environment of the ECU. Nowadays, the challenge is no longer to introduce such methods and techniques, but to manage the application including legacy versions and current product variants over many years. In doing so, one has to consider many different types of artefacts, for instance, the Simulink model, the requirements specification and the corresponding tests.

In the automotive industry MBD is often used in connection with a rollout of an application in different car lines. Although the reuse level across car lines is high, some differences cannot be avoided nevertheless. Moreover, the application is continuously developed, because of enhancements or newly introduced features. Consequently, there is the major challenge of managing consistent artefacts over time and car lines. With each change or new variant, the application developer is confronted with questions like: "Which functions are there already in the application?", "What are the dependencies between them?", "What effect will a change in requirements / a Simulink subsystem have?", "Are there some other requirements/Simulink subsystems concerned?", "Which consequences will result for the corresponding tests?", or "How can we ensure that the artefacts stay consistent despite the change?". Answering such questions today can only be done by an experienced application expert and involves complex searches in a large number of potentially involved artefacts.

To address these challenges, we propose to support development and variability handling using model-driven technologies. In particular, we apply technologies from the Eclipse ecosystem, such as the Eclipse Modelling Framework (EMF) and corresponding model transformation languages. The

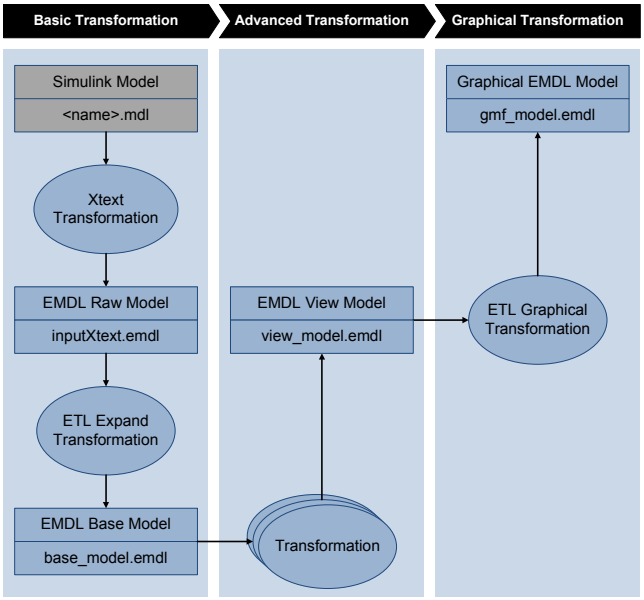


Figure 1: Overview of the transformation concept

main idea is to import original artefacts (as used in MBD industry practices) into an EMF-based representation and subsequently apply techniques, which are available in Eclipse-based frameworks. For instance, with such techniques an artefact can be analysed directly or combined with other artefacts for cross-artefacts analyses.

Out of the many involved artefacts, in this paper we focus on the Simulink model. We will first describe the model transformation concept, then we provide an analysis example producing another view of the model applying this concept. Finally we evaluate the concept in the context of benefit for industries.

2. TRANSFORMATION CONCEPT

To answer the questions raised in Section 1 we have developed a transformation concept, which is visualised in Figure 1.

The transformation concept consists of three main steps each of which is dedicated to a specific task in the process. First of all the Simulink model is imported into the EMF world during the basic transformation step and enriched with additional logical information. This step is described in more detail in Section 2.1. The result is a model that we call *EMDL Base Model* as we use it as input for further advanced transformations (cf. Section 2.2). These advanced transformations lead to models that represent special task-specific views of the original Simulink model. Such a "view model" can subsequently be transformed to a "GMF model" during the graphical transformation process described in Section 2.3 to be visualised.

2.1 Basic transformation

To explain the benefits of the basic transformation step, we will focus on lines that are connecting blocks in a Simulink model as they are the central element to identify those blocks which are transporting a signal and are potentially affected by the signal. For instance, in the very simple Simulink model visualised in Figure 2 we might be interested in which blocks are affected if the constant signal of block "Constant 2" changes. To automate this task we have to analyse the Simulink model text file, which includes all information about the different blocks, their connections, signal flows, graphical positions and so on.

If we want to perform such analysis, the original format of the text file is problematic as an input, because some of the necessary information mentioned above is available only in an implicit form. That is the reason why the task of identifying lines which are connected to block `Constant 2` in Figure 2 becomes quite complicated as illustrated in the following. Each line can be connected either directly to a block or to a port of a block. Lines and blocks are independent objects which are represented in the model file with a keyword ("Line" / "Block") followed by a body surrounded by curly braces "{}". All entities are just listed in a linear order without a logical or hierarchical structure. Listing 1 illustrates how a line between the Simulink blocks `Constant 2`, `Module A`, `Module C` and `Output Interface` of the Simulink model in Figure 2 is represented in the original Simulink model file.

```
1  Line {
2    [...]
3    SrcBlock            "Constant 2"
4    SrcPort             1
5    [...]
6    Branch {
7      [...]
8      Branch {
9        DstBlock        "Module C"
10       DstPort         1
11     }
12     Branch {
13       [...]
14       DstBlock        "Output Interface"
15       DstPort         4
16     }
17   }
18   Branch {
19     DstBlock      "Module A"
20     DstPort       1
21   }
22 }
```

Listing 1: A line as represented in the original Simulink file

Answering the previously mentioned question is problematic with this structure. For example, if we would like to retrieve all blocks that are linked by lines to the block `Constant 2` directly or indirectly via further blocks (which is generally a frequent task when identifying blocks affected by a signal) we first have to pick this block in the model file and retrieve its name (here `Constant 2`). Then, we have to iterate through all lines beginning at this block (i.e., which have an attribute "SrcBlock" with this name as value).

One of these lines is displayed in Listing 1. The names of the blocks which are directly connected to `Constant 2` will then be all *DstBlock* entries which appear in the line body of these lines. In order to identify blocks connected with block `Constant 2` indirectly we would have to look for the names of the destination blocks (DstBlock) of all lines which the condition `SrcBlock="Constant 2"` holds for. After this we have to repeat the whole process recursively on the resulting destination blocks.

In summary, the flat structure and textual storage format of original Simulink files, impedes efficient traversal and analysis of the model. This is not a suitable basis when

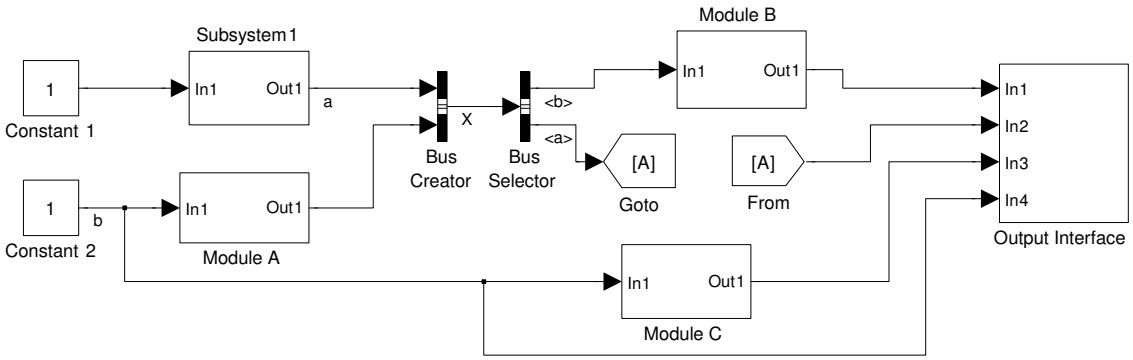


Figure 2: A Simulink model to be analysed

aiming to understand and analyse an ECU application, even if such analysis is performed with tool support.

To overcome this obstacle, we decided to explore techniques that would turn implicit information into explicit objects, such that queries on the Simulink model can be implemented and performed more elegantly and efficiently. In order to do so, we created a parser within the Xtext framework [10]. This parser turns a given Simulink model file in the original `.mdl` format into an EMF-based representation, which is stored in XMI (XML Metadata Interchange), a common interchange format for EMF models. This document conforms to an EMDL metamodel, which we defined to capture all information given by the Simulink model. Furthermore, we extended this metamodel by additional logical and hierarchical information, which facilitates analyses. Connections between blocks and lines, for example, will then be easier to address.

The Xtext parser is started via an Xtext workflow script.For the given Simulink model line in Listing 1 it generates the output shown in Listing 2.

```
<mdlLines name="b">
  [...]
  <branches>
    <branches>
      <destinationPointer = "Module C"/>
    </branches>
    <branches>
      <destinationPointer = "Output Interface"/>
      [...]
    </branches>
    [...]
  </branches>
  <branches>
    <destinationPointer = "Module A"/>
  </branches>
  <sourcePointer = "Constant 2">
</mdlLines>
```

Listing 2: A line as represented after Xtext transformation

Up to this point we have just translated the original Simulink model file *one-to-one* into the new EMDL Raw Model file, which conforms to the defined meta model. As we mentioned before some important information is implicit in the Simulink model and cannot be addressed directly. This information is still implicit in this EMDL Raw Model. For analyses purposes it is desirable to abstract from technical details to make the required information explicit. For

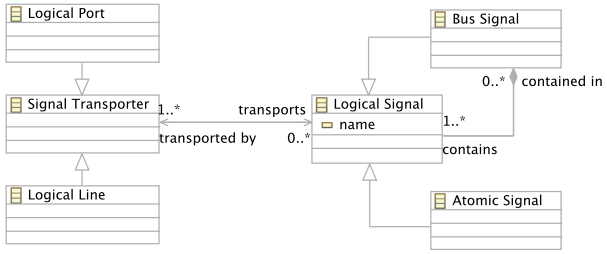


Figure 3: The structure of logical entities which facilitate the transformation process

the line displayed in Listing 2 an abstraction would not use the quite technical construct of branches. Instead it would just describe a *logical* line with one source and many targets (which would be references to the corresponding target blocks reached by this signal). Therefore we identified the following logical entities which become relevant in this context and which are created during a further transformation (ETL Expand Transformation):

1. Logical ports
2. Logical lines
3. Logical signals
4. Signal transporters

Figure 3 illustrates a part of the metamodel describing the relationships between these logical entities.

2.1.1 Logical ports

Each line in a Simulink model is connected either directly to blocks or to ports of blocks. In our enhanced model we facilitate this by introducing logical ports which belong to a block. Logical lines can then only be connected to logical ports. So in this context we do not have to distinguish between blocks and ports as endpoints of lines in a Simulink model. A logical port can either be a logical inport or a logical outport depending on whether a port is a line's target or source.

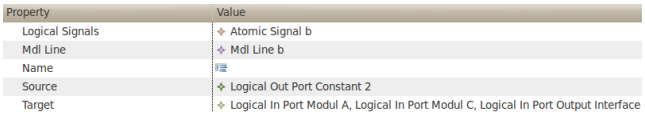


Figure 4: A logical line that abstracts from details of a Simulink model line

2.1.2 *Logical lines*

As mentioned before, in the EMDL Raw Model connecting lines between blocks are still represented in a quite technical manner. In other words, each line has exactly one source and one explicit target but it can contain many branches where further lines branch off (see Listing 2). This construct of branches is not necessary as it does not introduce more semantical information into the model useful for analysing signal flow.

For analysis purposes we are mostly interested in what block (or port respectively) is connected with which other blocks and which signals are transported. Hence, we introduced the logical lines which abstract from the details of *how* a line is connected with different blocks. Instead a line has exactly one logical source port and at least one logical target port but no branches as they do not introduce relevant information about signal flow. Consequently, there are no branching lines left which facilitates the process of analysing connections and signal flow between blocks significantly.

Figure 4 illustrates the logical representation of the Simulink line of Listing 1 (and MdlLine of the EMDL Raw Model of Listing 2). Furthermore, even visually separated blocks like `from` and `goto` blocks in the Simulink model are connected by a logical line.

2.1.3 *Logical signals*

While information about the signal flow is contained implicitly in the original Simulink model as described above (and hence difficult to extract) we create logical signals, which have a name and are related to all ports and lines transporting the signal. For example, in order to identify the signal flow of a given signal in the Simulink model we have to follow the lines beginning at the source block. But a line may also transport more than one single signal, i.e., it transports a bus signal that contains all transported signals.

To this end, we also introduce logical signals that can either be bus signals, which contain one or many further logical signals or atomic signals that may be contained in a bus signal but cannot transport other signals. Furthermore, each atomic signal is linked to all bus signals, which transport it. By this means, it becomes apparent for a given signal which lines transport this signal (either included in a bus signal or directly). For a given atomic signal we can now directly lookup which line is associated (either to this atomic signal or to a bus signal that contains this signal). Now we can perform queries on the model to identify parts of a model which are potentially influenced when a signal changes.

2.1.4 *Signal transporters*

In a Simulink model signals are transported between blocks. In order to facilitate identifying signal flows throughout a Simulink model both logical lines and logical ports are abstracted as “signal transporters”. By having such an abstraction at hand the problem of finding out, which parts of a model are influenced by a special signal, becomes solvable with a reasonable effort since the information about signal flow now becomes explicit. That means that we no longer need to follow a given signal through a Simulink model. Instead, we just have to query our transformed model for relevant signal transporters for the signal under consideration.

2.2 Advanced transformations

By enriching the original Simulink model with abstract logical information we created a useful base model, which now allows us to answer questions about the ECU application and the corresponding Simulink model with reasonable effort (e.g., ”Identify all lines and blocks of the model which are influenced by a change of signal ’a’.”). The second step of the concept is then to extract some information out of the EMDL Base Model and process it further. For that step we use transformations that we call *Advanced Transformations*. One of these is shown in Section 3. In future extensions these advanced transformations may also consider further input as exemplified in Section 6. The result of this step is called the *view model*, which represents those aspects that are relevant to a particular stakeholder or for a particular task.

2.3 Graphical transformation

To further support the engineer in his work, one objective is to provide tools, which provide interactive access to the task-relevant information. Hence, we aimed to produce a graphical representation of the view model to make the result of the analysis more comprehensible and intuitive. To this end, we have developed a graphical editor using Eclipse GMF (Graphical Modeling Framework).

The editor enables the user to recursively descend into subsystems by double-clicking them – similar to the behaviour known from Simulink. The graphical representation uses available positioning information as given by the Simulink model to arrange the blocks on the canvas.

Each model to be visualised in the editor has to be structured such that the editor can display it. Therefore, our transformed model has to be enhanced such that it provides the necessary information. This is the task of the graphical transformation. Especially lines and – if desired – branches of lines are introduced within this transformation. Now the created view can be visualised. The result of the graphical transformation is called a *GMF model*. Section 3 applies the described methodology to create such a graphical editor view.

2.4 Transformation languages

For all transformations we applied two state-of-the-art transformation languages, selecting one of them depending on the adequacy for a given problem. While the ATLAS Transformation Language (ATL) [8] succeeds in transforming huge models very efficiently, the Epsilon Translation Language (ETL) [9] seems to be more convenient to apply for a developer used to programming with Java or C++.

This is why we mainly use ETL whenever concrete modifications on a model have to be performed (e.g., to create special view in the advanced transformations) whereas we apply ATL to add minor additional information on possibly large models (e.g., to create logical signals and to distinguish bus and atomic signals). With transformation languages it is difficult to directly manipulate a given model in place (similar to accessing and modify objects in an object-oriented

programming language). Hence, we created what we call "identical transformations" both for ETL and for ATL. These transformations create an output model that is identical to the input model. That way, we can build new advanced transformation just by modifying these identical transformations as necessary.

It should be noted that the applied transformation languages (both ATL and ETL) provide some similar support for copy-and-modify transformations, e.g., in ATL's refine mechanism or in ETL's capability to generate a copy transformation. However, we had mixed results with these and are still exploring our options to find the best solution for use in everyday practice.

3. ANALYSIS OF SIMULINK MODELS

The previous section explained the transformation concept. Such transformation of a Simulink model into the Eclipse world enables to use all techniques available from the Eclipse Modeling Project, e.g., corresponding transformation languages. In this section we apply these concepts to support the development of Simulink models. In particular, we are aiming to support a development strategy, where a model is built out functional, compatible units taken from a library. We will refer to this approach as "module-based".

A Simulink model is often used to generate code, for instance with Target Link. As models are used by different partners many standardisation techniques have to be adopted (e.g., to comply with AUTOSAR [4]). For example abstraction techniques have to be adopted to follow special design patterns. Parts of the whole system might be grouped into subsystems, which in turn themselves may contain subsystems. Furthermore, signals may be integrated into buses. Consequently, the structure of a Simulink model often becomes very complex.

Especially, as we are dealing with families of products (e.g., similar implementations for various types cars) the implementation model contains the functionality for a whole family of variants. These variants are used to adopt the models for different vehicles where slightly different changes are necessary. Therefore different patterns are used to enable variant management of the functionality. In earlier work, we presented some approaches to manage and configure variants [12, 11]. One of these patterns is a subsystem called `Module`, which encapsulates the functionality of one feature.

Figure 5 shows the pattern of a module. There are different activation states. The *Disabled* state is set when the feature/module is never active in the variant (e.g., due to an functional option that is not available in this particular car type). When the module is *Enabled* it is either active or inactive. *Active* means that the module is running while an *Inactive* module is available but not running. The different activation states are controlled by the subsystem **A**.

With this structure the set of all modules can be divided into different groups of running modules. Again we have to deal with the complexity of the system, which is hard to handle for the human engineer. The modules running at a specific car state are not easy to grasp and understand by a developer, neither on the requirement specification level nor on the level of the Simulink model implementing these requirements. To analyse the model the developer needs to "see" which module is active at which state and which dependencies exist between modules.

As an example for the above mentioned issue we assume

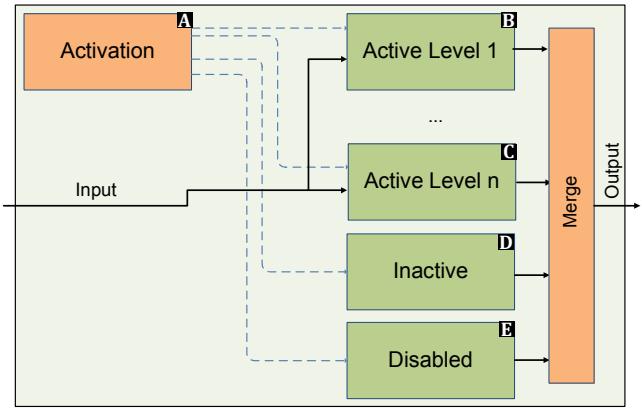


Figure 5: Structure of the module pattern

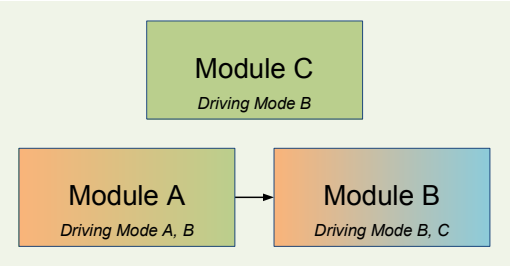


Figure 6: Example for a module structure which might cause problems

the Simulink model given in Section 2 shown in Figure 6. The `Module A` is activated in `DrivingMode_a` and `DrivingMode_b` and providing signals to `Module B`. There is no problem with `Module C` which is activated in `DrivingMode_b` does not depend on signals from `Module A`. But `Module C` is activated in `DrivingMode_B` and `DrivingMode_C`. So it expects to have input from `Module A` in both modi. But `Module A` does not provide any signals in `DrivingMode_C`. This might cause an error.

One important task when developing the Simulink model is the analysis of dependencies of modules within the models. Hence, our work aims at providing a view which shows the dependencies of modules and the states where the modules are activated. The view should depict all modules on one pane with arrows indicating the dependencies and the structure which shows the activation.

To do so, we first import the Simulink model into EMF as described in Section 2.1, then in a second step we have to transform the EMDL Base Model to the desired "view model", showing the dependencies between modules. In the next chapter we describe this advanced transformation.

3.1 Transformation of the Simulink Model

We have written two advanced model transformations (see the structure shown in Figure 7), which identify the modules at a first stage ❶ and transform ❷ the modules to a structure according to the requirements described above.

The transformation identifying modules in the Simulink model is written in the ATLAS Transformation Language

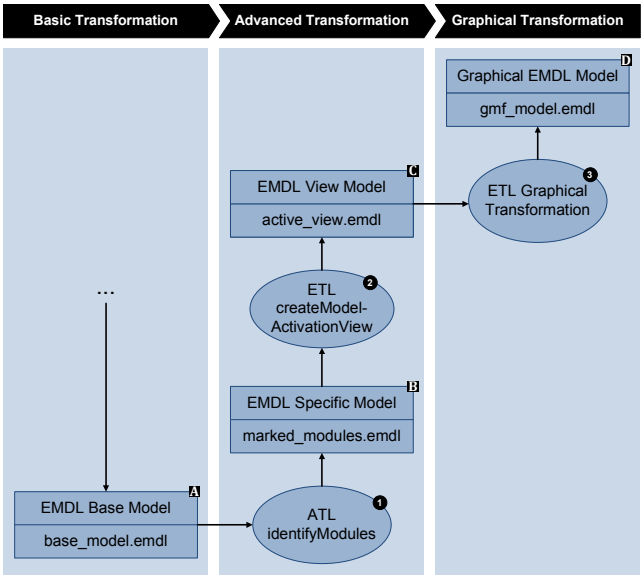


Figure 7: Overview of the model transformation to create a flattened module structure

(ATL) [8]. Modules can be identified via a tag added to the Simulink model. Each found module is transformed into an instance of "Module" (a metaclass introduced in the metamodel). By these means it is possible to identify the modules in later processing steps.

The second transformation creates a new model consisting only of modules and dependencies. To do this, we used transformation rules written in Epsilon Transformation Language (ETL) [9]. Listing 3 describes the essential rule for a root system and the computation for new targets.

The rule `CreateRootSystem` converts a normal `System` (input model) to a new special system (output model) where only blocks of the type `Module` are available. The transformation of these new blocks can be found in lines 13-18, where a loop transforms all instances of Module in line 15 and adds them to the target model in line 17.

Within this rule we also adapt the lines such that they should now indicate, that there is a connection between modules. These connections are created by the operation `createNewLine` which is called in line 19. Due to space constraints we do not depict this function. Its main contribution is to create a new line with the source module m from the loop and the targets which are computed by the operation `getLinkedModules`.

```
1  [...]
2  rule CreateRootSystem
3    transform    s:MdlIn!System
4    to           t:MdlOut!System
5    extends NamedElement
6    {
7
8      var allModules :new Set(MdlIn!Module);
9
10     for (m in MdlIn!Module.allInstances()){
11       allModules.add(m);
12     }
13
14     // add only module blocks
15     for (m in allModules)
16     {
17       var outModule : MdlOut!Module = m.equivalent
                ("CreateModule");
18       t.blocks.add(outModule);
19
20       //create the structural connection
21       t.logicalLines.add(createNewLine(m, m.
             getLinkedModules(allModules));
22     }
23   }
24   [...]
```

Listing 3: Transformation rule for System given in ETL

The operation mentioned above, which searches for linked modules is an important operation, which is depicted in Listing 4. This operation is realized as a recursive object-oriented function. For a given block and a set of `Subsystems` given as parameter it returns a subset of `Subsystems` which are connected to the calling block. The result will be found in the variable `linkedModules`.

As illustrated in line 3 the operation therefore iterates through all outports of the block which the operation is invoked for. If the output is linked to a logical line we retrieve all targets of the line (line 5-7) and decide what to do with them based on the block type of the target block of the line.

If the target is contained in the set of our modules we have found a connection and add it to our result (lines 8-10). If we have found another subsystem we have to search for modules within this subsystem. So we are calling this function again recursively (line 12) and add all results we obtain from this call. If we have reached the end of a subsystem we leave it and continue searching in the containing system (line 14). For all other types of blocks we call the function again recursively (line 16).

```
1  operation MdlIn!Block getLinkedModules(modules :
       Set(MdlIn!SubSystem)) : Set (MdlIn!SubSystem)
       {
2    var linkedModules : new Set(MdlIn!SubSystem);
3
4    for (outPort in self.outPorts){
5
6      //  is there a Logical Line
7      if (outPort.linkingLine.isDefined()){
8
9      // Search in all Targets
10       for (target in outPort.linkingLine.target){
11
12         if (modules.includes(target.parent)){
13           linkedModules.add(target.parent);
14         }
15         else if(target.parent.isTypeOf(MdlIn!
               SubSystem)){
16       linkedModules.addAll(target.accordingBlock.
             getLinkedModules(modules));
17         }
18         else if (target.parent.isTypeOf(MdlIn!
               BlockOutPort)){
19           linkedModules.addAll(target.
                accordingPort.parent.
                getLinkedModules(modules));
20         }
21          else {
22           linkedModules.addAll(target.parent.
                getLinkedModules(modules));
23         }
24       }
25     }
26   }
27   return linkedModules;
28 }
```

Listing 4: Operation which determines the targets for a given module

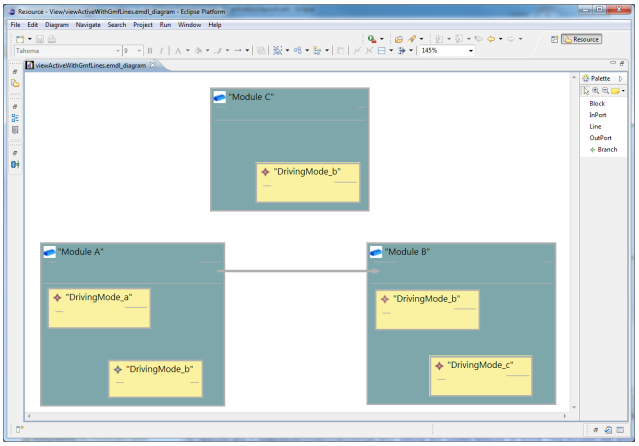


Figure 8: Graphical editor for the transformed Simulink Model

With the algorithm in Listing 4 we are able to scan the whole model structure no matter if there are `subsystems`, `bus structures`, `goto` or `from` blocks. Hence we are able to find the modules that are connected with a special module very easily due to the fact, that we did a lot of structural work before.

The resulting model now contains the structure we are searching for. All modules have been lifted to the same layer and connected if there is a connection in the source model.

3.2 Graphical Editor

In Figure 8 we show how the transformed model appears in the editor.

We used the example from Figure 2 in Section 2. After the transformations the modules A, B and C are on one layer and if modules are interconnected this it depicted by lines, e. g., like between module A and module B. Within the modules the developer can see the structural content. Therefore he is able to decide in which levels the module is activated.

4. EXPERIENCES AND EVALUATION OF VIEWS

We applied the presented transformation concept on real examples from automotive industry. These examples are Simulink models which have up to ten different *Driving Modes* similar to the presented example in Section 3. With this concept module dependencies and their activation can now automatically be extracted out of the Simulink model. The application developer no longer has to do the exhausting job of looking for this information in the Simulink model.

The views allow to save time and to avoid losing track during search in the model. Additionally the graphical representation facilitates the analysis of the information. Adding a new module in the Simulink model or changing an existing one is now better supported.

During early testing we noticed that the presented transformations work correctly and efficiently when applied on less complex Simulink models. However, there seem to be enormous differences between the used transformation languages ETL and ATL. For instance, by (manual) translation of an ETL transformation into an equivalent ATL transformation we could increase the performance of the transformation by factor 60 related to one and the same Simulink model. It is too early to give detailed report on the reasons for this discrepancy, but it surely seems to be worth further investigations.

Furthermore, we discovered that independent of the transformation language very complex models caused heap space errors on a normal PC (AMD Dualcore with 3 GB RAM). This currently stops us from performing the presented transformation with reasonable effort for very complex models. We plan to tackle this challenge by (1) translating current ETL transformations into equivalent ATL transformations and (2) shrinking the Eclipse representation of the Simulink models.

The latter means for example to drop out information about the Simulink model which is not necessary for a developer to visualise in a view, e.g., options associated to blocks, technical lines (cf. Section 2.1) and graphical positioning information. However, this means that a transformation of the Eclipse representation back into a Simulink model will no longer be possible. But as we primarily aim at generating views this will be acceptable for the current usage scenario.

5. RELATED WORK

Similar to our approach of analysing Simulink models there is the MOFLON framework [3], which provides tools to access Simulink models and other process artefacts. The authors present a method of keeping requirements and Simulink models consistent with the help of model transformations given in the MOFLON framework.

Alhawash et al. [2] provide another framework to develop and analyse automotive software systems. The framework supports the development in an early design phase with different views based on a common model provided by this framework. The concrete specification and implementation is done with Matlab/Simulink.

Agrawal et al. [1] focus on model transformation of Matlab Simulink and Stateflow models in a verification context. Therefore they adopted the Graph Rewriting and Transformation language (GReAT) to build hybrid automata from given models specified in Hybrid Systems Interchange Format (HSIF). In contrast of our work the transformation focussed on the semantics of the Simulink model.

A further important focus on model transformation is presented by Biehl et al. [5]. They use the above mentioned ATLAS Transformation Language (ATL) to automate translations from the automotive architecture description language EAST-ADL2 to a safety analysis tool called HiP-HOPS in the context of model-based development of safety-related embedded systems.

This paper primarily aims to support the developer to manage variability and changes of products or product lines respectively based on the analysis of Simulink models via Eclipse frameworks. Another approach presented in earlier work [12] is managing variability via improving the Rapid Control Prototyping engineering process with the help of formal feature models.

[7] attempts to integrate product configuration and variability resolving into domain specific languages with special focus on dependencies between elements and features. To this end, the authors adopt higher-order transformation languages like the ATLAS transformation language (ATL).

[6] deals with possibilities to translate domain specific languages into configurable models with formal semantics. The presented framework enables a tool-supported configuration process by visualising mapping between features and implementation including explanations about constraining dependencies (e.g., excludes and requires) in a given feature selection. Furthermore they show how negative variability combined with subsequent pruning can be used to derive a product-specific model for a given configuration. Finally they evaluate the presented approach on a parking assistant.

[11] elaborates on challenges which result from combining embedded software products to a product line in model-based engineering and how the upcoming problems can be tackled. These challenges range from complexity handling to tool integration. Therefore they create a Simulink model from a feature model and transform this model into a domain model in the Eclipse world using the Xtext framework. A product can then be derived by selecting the desired features in the feature model and mapping this selection to the previously created domain model.

6. CONCLUSIONS

In this paper we presented an approach to analyse the structure of a Simulink model to facilitate the engineering process, in particular with respect to change and variability management. We exemplified the methodology by a model-based developed ECU application inspired by real projects in automotive industry. Hence, our work is based on Simulink models where we identified so-called modules (i.e., subsystems that represent a special feature) and restructured them to enable dependency analyses.

To this end, we created a view using model transformations to determine the interesting blocks and interrelations between them. The visualisation is implemented by a model-based developed editor. Finally, we evaluated our approach on a real industry example.

Generating this particular view was only a first step in evaluating EMF to support the further development of ECU applications. We plan to implement and evaluate diverse further views with respect to their impact on industry development. Moreover, we intend to develop a method or a language suited to define arbitrary special view.

Additionally, we expect benefits and possibilities from integrating further input into such transformations. Integrating artefacts of different engineering phases, e.g., requirements and test cases will enable analyses that support a traceable engineering process of ECU applications. Even formal feature models could be used as further input. For instance, an analysis tools could identify and visualise which parts of the Simulink model will be affected by feature selection.

7. REFERENCES

[1] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science*, 109:43–56, 2004. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).

[2] K. Alhawash, T. Ceylan, T. Eckardt, M. Fazal-Baqaie, J. Greenyer, C. Heinzemann, S. Henkler, R. Ristov, D. Travkin, and C. Yalcin. The fujaba automotive tool suite. In *Proc. of the 6th International Fujaba Days 2008, Dresden, Germany*, 2008.

[3] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag, Springer Verlag.

[4] AUTOSAR. Autosar - automotive open system architecture. http://www.autosar.org.

[5] M. Biehl, C. DeJiu, and M. Törngren. Integrating safety analysis into the model-based development toolchain of automotive embedded systems. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 125–132, New York, NY, USA, 2010. ACM.

[6] G. Botterweck, A. Polzer, and S. Kowalewski. Interactive configuration of embedded systems product lines. In *Proceedings of the 1st International Workshop on Model-driven Approaches in Software Product Line Engineering(MAPLE 2009), collocated with the 13th International Software Product Line Conference (SPLC 2009)*, volume 557, pages 29 – 35, San Francisco, California, USA, August 2009. CEUR Workshop Proceedings. ISSN 1613-0073.

[7] G. Botterweck, A. Polzer, and S. Kowalewski. Using higher-order transformations to derive variability mechanism for embedded systems. In *2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACESMB 2009), Workshop at the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009)*, Denver, Colorado, USA, September 2009.

[8] Eclipse-Foundation. Atl (ATLAS Transformation Language). http://www.eclipse.org/m2m/atl/.

[9] Eclipse-Foundation. Epsilon. http://www.eclipse.org/gmt/epsilon/.

[10] Eclipse-Foundation. Xtext - a programming language framework. http://www.eclipse.org/Xtext/.

[11] A. Polzer, G. Botterweck, I. Wangerin, and S. Kowalewski. Variabilität im modellbasierten Engineering von eingebetteten Systemen. In *7. Workshop Automotive Software Engineering*, volume P-154 of *Lecture Notes in Informatics (LNI)*, pages 2702 – 2719. Gesellschaft für Informatik (GI), 2009.

[12] A. Polzer, S. Kowalewski, and G. Botterweck. Applying software product line techniques in model-based embedded systems engineering. In *Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2009), Workshop at the 31st International Conference on Software Engineering (ICSE 2009)*, pages 2–10. IEEE Computer Societ, May 2009.

Assertion-Based Test Oracles for Home Automation Systems*

Ajitha Rajan, Lydie du Bousquet, Yves Ledru, German Vega, Jean-Luc Richier
Laboratoire d'Informatique de Grenoble (LIG), Grenoble, France
{ajitha.rajan, lydie.du-bousquet, yves.ledru, german.vega, jean-luc.richier}@imag.fr

ABSTRACT

The Home Automation System (HAS) is a service-oriented application that facilitates the automation of a private home to improve the comfort and security of its residents. HAS is implemented using a service-oriented architecture. Many of the services in the HAS dynamically change their configuration during run-time. This occurs due to change in availability and bindings between services. Dynamic reconfigurations of services in the HAS presents several testing challenges, one being the specification of test oracles. In this paper, we give an approach for specifying test oracles for services in the HAS. We formally specify test oracles in the JML specification language. To verify service behavior in the presence of dynamic reconfigurations, we use mechanisms in the service architecture that notify dynamic changes along with run-time evaluation of JML specifications. We illustrate our approach using an example service in the H-Omega HAS developed on the OSGi^{TM}and iPOJO service platform. To evaluate our approach, we developed a testing framework that allows for generation of tests with dynamic service reconfigurations. In addition, we seeded faults into the example service, and evaluated the effectiveness of the test oracles in revealing the faults using the generated tests.

1. INTRODUCTION

Modern day homes are being revolutionized with the advent of devices and technologies that can network and communicate with each other. A Home Automation System (HAS) facilitates the automation of a private home to improve the comfort and security of its residents. It integrates different home appliances via a network to provide services for entertainment, safety, and comfort. For instance, integrating a TV, a DVD player, surround speakers, lights, curtains and an air-conditioner allows to provide an integrated service, that we call *Theater integrated service*, where a user can watch movies in a theater-like atmosphere. HAS is an application in the domain of Service-Oriented Computing (SOC) and is implemented using service-oriented architecture. The HAS, like any other SOC application, utilizes services as the basic units to support development of the distributed application.

Most of the research in SOC has focused on the architecture and framework for developing SOC applications. Research in verification of SOC applications is still in its infancy. The main challenge in verifying SOC applications like the HAS lies in the dynamic reconfigurations that often occur in these applications. In the HAS, new services may appear or existing services may disappear as the application is running. Not only is there a dynamic change in availability of services but the bindings between them also change during run-time. As a result, the architecture and configuration of the HAS and its services evolve dynamically. In the rest of this paper, we refer to this phenomenon in the HAS as its dynamic nature/behavior/reconfigurations. To exemplify, consider the *theater integrated service* mentioned earlier. The service may be required to connect to a mobile video player (like an iPad) if it is available in the room and play videos from it. Thus, if a mobile video player appears in (or disappears from) the room when the theater integrated service is running, the service is required to dynamically bind to (or unbind from) the player service at run-time. Such dynamic changes in service configuration may affect the correctness and quality levels of these applications.

Verification of HAS and other applications with dynamic reconfigurations can be viewed as two testing problems, (1) the need for test oracles that observe and check behavior during dynamic reconfigurations, and (2) the need to generate tests that involve dynamic service reconfigurations. In this paper, we primarily focus on addressing the first testing concern–specifying test oracles for HAS. Nevertheless, to evaluate our approach for test oracles, we also addressed the second testing concern with regard to test generation, albeit in a preliminary manner.

Traditional test oracles that examine outputs at the end of the test execution are not adequate for the HAS since the configurations and context of the service can change dramatically as the service is running. We need test oracles that are run-time monitors, continuously monitoring the behavior of the service, particularly during dynamic reconfigurations. Test oracles that monitor the run-time behavior of a system for consistency with requirements have been proposed in the past [20, 16, 11]. These approaches, however, cannot be directly applied to the HAS since they are not tailored towards monitoring dynamic reconfigurations in the service

*Partially supported by the iPOTest Project of the Université Joseph Fourier, and by the ISLE cluster of the Région Rhône-Alpes .

MOMPES '10, September 20, 2010, Antwerp, Belgium

composition and bindings.

To address this issue, we propose test oracles in the form of formal specifications that act as run-time monitors of the services in the HAS. The test oracles monitor whether the services deliver the functions expected from them in the presence of dynamic reconfigurations. Our approach relies on utilizing mechanisms in the service architecture to notify run-time monitors of dynamic reconfigurations. We use the Java Modeling Language (JML) [17] specification language to formally specify test oracles. We illustrate our approach on an example service in the HAS. The HAS we use in this paper is simulated using the H-Omega [6, 9] framework implemented on top of OSGi [2].

To evaluate our approach for test oracles, we generated tests with dynamic service reconfigurations for the HAS. We adapted our existing combinatorial testing tool, TOBIAS [18], to support test generation with dynamic reconfigurations that can be executed on a service-oriented platform. We evaluated the fault revealing capability of our test oracles by seeding faults into the example service and running the TOBIAS test suite against them. Automatically generating tests with dynamic reconfigurations for SOC applications like the HAS has not been explored extensively in the past. We believe our effort at test generation is a useful, although preliminary, step in this direction.

2. BACKGROUND

2.1 Framework for Home Automation Systems

The Adele team at the Laboratoire d'Informatique de Grenoble (LIG) developed a platform, called H-Omega [6, 9], for building home automation systems. The H-Omega gateway eases the creation and deployment of new services by transparently managing service bindings, heterogeneity, and dynamism. The gateway is implemented on top of OSGi^{TM} [2] and iPOJO [10]. The OSGi framework is a system for Java that implements a dynamic component model that can be remotely managed. The service-oriented component model, iPOJO (injected POJO), aims to simplify service-oriented programming on OSGi frameworks by transparently managing service dynamics.

The iPOJO framework allows developers to distinctly separate functional code (i.e., the POJO - acronym for *Plain Old Java Object*) from the non-functional code (for dependency management, service provision, configuration, etc.). All non functional concerns are externalized and managed by the container through handlers (see Fig. 1). The component is the central concept in iPOJO. The description of the component — information on service dependencies, provided services, and callbacks — is recorded in the component's metadata. Using the component metadata, the iPOJO runtime manages the component, i.e., manage its life cycle, inject required services, publish provided services, discover needed services.

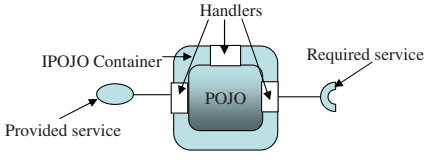


Figure 1: Component in iPOJO

2.2 Example Service in HAS

To help illustrate the principles in iPOJO, consider the following example of an integrated service, termed *Temperature Control*, in the HAS. Note that the HAS and temperature control service are simulations using the H-Omega framework. Real home automation systems and their services are not easily available. We chose to use the temperature control service example simply to illustrate dynamic reconfigurations in services and the need for their continuous monitoring. Other services like the theater integrated service, mentioned in Section 1, may also be used in its place. Regardless of the example service used, the monitoring challenges encountered are similar. The temperature control service controls the temperature of the room, so that the target temperature desired by the user is reached. The service requires heaters, and a display device (termed LCD in the service) that displays the number of heaters active and running. At least one heater and LCD are mandatory requirements for the temperature control service, implying the service will be invalid if either of these devices are unavailable. The service also ensures that the heaters are used economically. The number of heaters that ought to be running for economical usage is controlled based on the difference in temperature between the desired target temperature and current room temperature. The service uses the following conditions for economical usage of heaters:

Temperature Difference	< 10	Turn on 1 heater
Temperature Difference	10 *to* 20	Turn on <= 3 heaters
Temperature Difference	> 20	Turn on All heaters

The service continuously monitors the temperature difference, and controls the available heaters in the room (turning heaters on/off) based on this difference. The *dynamic aspect* in the service is introduced by two factors:

1. Heaters may appear/disappear from the room. (Assuming the heaters are portable heaters)
2. Depending on the temperature difference, the number of active heaters in the room keeps changing. The LCD should display the number of active heaters and update the display as the number of active heaters changes.

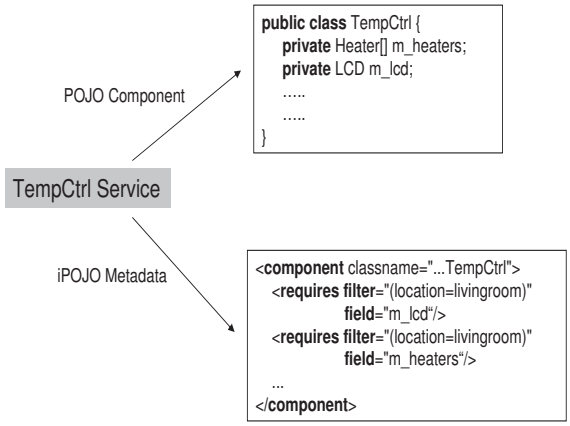


Figure 2: Temperature Control Service - POJO component and metadata

The POJO component of the temperature control service contains the Java class defining functionality of the service. The devices (or services) required by temperature control are simply used as fields in the component class. Figure 2

shows a portion of the POJO component for the temperature control service with field declarations for the required devices. For instance, *m_heaters* in the POJO component in Figure 2 is an array of heaters whose length will vary depending on the number of available heaters at run-time. To enable iPOJO to manage this component, we describe the component in the metadata file. In the metadata, we ask iPOJO to create the temperature control component and an instance, indicate the service provided by temperature control, and indicate fields in the component that represent required services. Figure 2 shows a portion of the iPOJO metadata for the temperature control service with the required services. As seen in the figure, fields for heaters, and LCD in the temperature control component are indicated as required services (using the *requires* tag) to be injected by iPOJO at run-time. If required services are unavailable, the temperature control component instance becomes invalid. When services fulfilling the requirement appear, the instance becomes valid.

2.3 JML: Java Modeling Language

JML is an annotation language used to specify Java programs by expressing formal properties and requirements on the classes and their methods [17]. Our proposed approach for test oracles in the HAS uses JML specifications. We chose to use JML as the formal specification language for the following reasons, (1) Wide range of tools already exist for JML, supporting, runtime assertion checking, static checking, program verification, generation of annotations, specification browsing [17], and (2) The HAS application was implemented in the Java programming language. JML is a natural choice as a formal specification language for Java.

JML specifications appear within special Java comments, /*@ and @*/, or starting with //@. The specifications of each method precede the method declaration. We ask the reader to refer to [17] for a discussion on syntax and usage of JML specifications. An example JML postcondition in the temperature control service is presented below.

```
//@ ensures
    ((isrunning && (m_heaters.length >= 3) &&
        (tempdiff >= 10) && (tempdiff < 20))
      ==> (num_running == 3));
```

The postcondition states that if the service is running and the number of available heaters is greater than or equal to 3 and the temperature difference between current and desired room temperature lies between 10 and 20 degrees, then the number of heaters active in the room should be 3.

Our approach uses JML specifications as run-time monitors. The JML Runtime Assertion Checker (RAC) [7] provides this capability. It translates JML specifications into runtime checking bytecode, and verifies that specifications are satisfied during program execution.

3. TEST ORACLES USING JML SPECIFICATIONS

Test oracles that monitor the run-time behavior of a system for consistency with requirements have been proposed in the past [20, 16, 11]. Additionally, run-time monitoring of JML specifications for use as test oracles in unit testing of programs has been proposed previously [8]. Nevertheless, these existing approaches for run-time monitoring have never been used for applications like the HAS where the architecture is dynamically evolving, i.e., where bindings among components in the application change dynamically, and components available for composition also change. To adopt existing run-time monitoring techniques for the HAS, we need to enhance them with the capability of monitoring service behavior during dynamic service reconfigurations. In this section, we present our approach for doing this using JML specifications along with capabilities in the service architecture.

3.1 Validating Service dynamism using JML

As with any run-time monitoring technique, one of the difficult and important aspects lies in ensuring the assertions are placed and checked at the right points in the execution of the program. This aspect is more challenging in the HAS due to the presence of dynamic reconfigurations in the application and execution context. Broadly, in our approach, we tackle this issue by first identifying potential sources of dynamic behavior in the service, i.e. fields in the service that may change dynamically. We then place probes in the service architecture, so that dynamic changes at the identified sources are communicated to a *listener method* in the service component. The listener method is associated with a set of JML assertions that check the correctness of the service during dynamic reconfigurations. If a JML assertion is violated, a run-time exception is raised to notify the user.

To better understand our approach, we briefly describe how dynamic nature in services is managed by the H-Omega framework. We then discuss the mechanisms we use in the architecture to alert a listener method. The main source of dynamic behavior in the HAS lies in the dynamism in service availability which in turn affects other services depending on or requiring them. When a service (or component instance in the vocabulary of iPOJO) requires another service, the iPOJO framework chooses a suitable service satisfying the requirements and directly injects the required service object inside a field in the component, or invokes a method when the required service appears (or disappears). The dependency handler in iPOJO manages service dependencies/requirements. As stated in [1], the handler manages two types of service injection in the component to handle dependencies:

1. Field injection: a field in the component contains the service object. As soon as the field is used, a consistent service object is injected. This injection type fully hides the dynamism.
2. Method invocation: when a required service appears, or disappears a method in the component is invoked. For each dependency, the component can declare bind and unbind methods that get invoked when the service appears or disappears, respectively.

In the second injection mechanism, method invocation, the dynamics can be managed directly by the developer. Each dependency can declare two callback methods: A bind method, called when a service appears, and an unbind method, called when a service disappears. The two injection mechanisms, field injection and method invocation, can also be used together. In this combined injection mechanism, the field receives the value before the bind method invocation. So, if the field is used in the bind method, the returned value will be up to date. Table 1 presents a portion of the iPOJO metadata for the Temperature Control service

Field Injection Mechanism

```
<component classname="...TempCtrl">
  <requires filter="(location=livingroom)" field="m_heaters">
  </requires>
  ...
</component>
```

Method Invocation Mechanism

```
<component classname="...TempCtrl">
  <requires>
      <callback type="bind" method="bindHeater"/>
      <callback type="unbind" method="unbindHeater"/>
  </requires>
  ...
</component>
```

Combined Injection Mechanism

```
<component classname="...TempCtrl">
  <requires filter="(location=livingroom)" field="m_heaters">
      <callback type="bind" method="bindHeater"/>
      <callback type="unbind" method="unbindHeater"/>
  </requires>
  ...
</component>
```

Table 1: Injection Mechanisms in iPOJO Metadata for TempCtrl Component

component (introduced earlier in Section 2.2) that shows the dependency of the service on heaters. The three mechanisms for injecting the heater service–field injection, method invocation, and combined injection–for this dependency are illustrated in Table 1. In the field injection mechanism, we simply mention the field name *m_heaters* in the *requires* tag for the component, and iPOJO takes care of injecting and updating the field when heaters appear or disappear from the living room. In the method invocation mechanism, we define the callback types *bind* and *unbind* along with their associated methods in the component in the *requires* tag. When a heater appears, the *bindHeater* method defined in the temperature control component is called. In a similar fashion the *unbindHeater* method in the component gets called when a heater disappears[1]. The bind and unbind callback methods are responsible for updating the *m_heaters* field. Finally, in the combined injection mechanism, we use both the field name and callback types in the *requires* tag. When a heater appears or disappears, iPOJO takes care of injecting and updating the *m_heaters* field. After the field update, the callback methods in the component get called.

Our approach uses the combined service injection mechanism, since it allows the component to be notified of the dynamic event, while taking care of the burden of updating the field with a consistent service object automatically. The callback methods in the component that get invoked when the dynamic change in the field occurs are referred to as the *listener methods*. The listener methods usually define actions or updates to be executed in the service after the reconfiguration. We attach JML assertions to these listener methods. Thus, every time the service is dynamically reconfigured, the listener methods are invoked and consequently, JML assertions associated with the listener method are evaluated and checked for violations. Our approach thus validates service behavior during dynamic reconfigurations while the service is running.

Our approach for monitoring services is targeted at helping the creators of integrated services like the temperature control or theater integrated services with testing and monitoring their behavior. Our approach manually inserts JML assertions and tags listener methods in the service implementation. We believe it is reasonable to assume that the creators and testers of the service have access to its implementation when testing the service. Note that our approach views other devices and services used by the service of interest as a black box. We do not intrude into the implementation of these services. The completeness and correctness of the JML assertions would have to be manually ensured by the testers of the service.

3.2 Test oracles for Temperature Control Service

In this section, we illustrate our approach for test oracles using the temperature control service introduced earlier in Section 2.2. The test oracles monitor the dynamic nature of the service in the H-Omega architecture. The dynamic aspect in the service is introduced by the appearance or disappearance of heaters, or due to temperature change resulting in heaters being dynamically switched on/off.

We begin by briefly describing the methods implementing the functionality of the temperature control component. The *execute()* method in the component is responsible for the core functionality of the service. When the temperature control service is activated, the *execute()* method in the service component is called. The method computes and monitors the temperature difference every 2 seconds[2]. Based on the temperature difference and the availability of heaters, the *execute()* method switches on/off the heaters, sets the target temperature, and displays the number of active heaters on the display device (LCD). The *bindHeater()* and *unbindHeater()* methods serve as the listener methods in the service component that are notified of dynamic changes in the heaters. The listener methods are responsible for making the necessary updates to the LCD display when dynamic reconfigurations in heaters occur.

We now proceed to describe the test oracles for this service. For ease in understanding, we split the JML specifications for the service into two: (1) JML specifications that monitor service behavior during normal (no dynamic reconfiguration) service operation, (2) JML specifications that monitor service behavior during dynamic reconfigurations. The JML specifications in (1) are associated with the *execute()* method. The JML specifications in (2) for monitoring dynamic behavior are associated with the listener methods, *bindHeater()* and *unbindHeater()*. To better understand the JML specifications for the service, we give a brief description of the variables used in the specifications.

- *num_running* reflects the number of heaters that ought to be active and running based on the temperature difference and number of available heaters.
- *isrunning* is a boolean variable that reflects whether the temperature control service is running.
- *m_heaters* is an array of available heaters in the room.

[1]The callback methods, *bindHeater* and *unbindHeater*, use the heater service object appearing or disappearing as a parameter in the method definition. iPOJO infers the service object type for the requirement using this parameter.

[2]We found that monitoring every 2 seconds was adequate to detect change in room temperature. Other values that also ensure frequent monitoring can be chosen. Choice of this value is only a service implementation concern, it does not affect the specification of test oracles.

m_heaters.length gives the number of available heaters.

- *m_lcd* represents the LCD device available in the room.
- *tempdiff* represents the temperature difference between the desired and current room temperature.

iPOJO takes care of injecting the fields, *m_heaters* and *m_lcd*, at run-time with available heaters and LCD, respectively. Recall that our approach for handling dynamic reconfigurations in services uses the combined service injection mechanism mentioned in Section 3.1. As a result, when heaters used in the service get dynamically reconfigured, iPOJO automatically updates the *m_heaters* field with the change while also notifying the change to the listener methods. All three methods in the service component, *execute()*, *bindHeater()*, and *unbindHeater()* update the variables *num_running*, the *m_lcd* display, and *tempdiff*. The *isrunning* variable is updated by the *execute()* method.

The first set of JML specifications, invariants and post conditions, to check service behavior during normal operation, are shown in Table 2. The post conditions (using the *ensures* clause) in Table 2 are specified on the *execute()* method and should hold after the *execute()* method call. The invariants (using the *invariant* clause), on the other hand, are checked *before* and *after* every method execution in the service component. The post conditions $N1, N2, N3, N4, N5$ ensure that the number of heaters active in the room correspond to the number of available heaters and the temperature difference conditions described earlier in Section 2.2. The post conditions for the heater ($H1$, $H2$, $H3$) ensure that only *num_running* heaters as per the economic usage conditions are *on*. They also ensure that all of those heaters are set at the desired target temperature. The invariants ($L1, L2$) for the LCD ensure that when the temperature control service is running, the LCD is *on* and displays the number of active heaters. The invariants $L1, L2$ should hold at the beginning and end of the *execute(), bindHeater(), unbindHeater()* method executions. Note that since $L1, L2$ are specified as invariants, they aid in monitoring service behavior during dynamic reconfigurations in addition to normal service operation. It is also worth noting that properties involving the heaters are specified as post conditions, rather than invariants, since dynamic changes in heater availability would cause such invariants to be violated *before* calls to *bindHeater()* and *unbindHeater()* that take care of the necessary service updates during dynamic reconfigurations.

The second set of JML specifications is to monitor the dynamic nature of the service. In the service component, the listener methods *bindHeater()* or *unbindHeater()* respectively get called when heaters satisfying the temperature control service requirements appear or disappear. To monitor dynamic reconfigurations, our approach associates JML specifications to these listener methods. We present the bind/unbind methods along with their JML specifications in Table 3. When a heater appears, the *bindHeater()* method in Table 3 is called by the service. Within the method, if the conditions for economic usage are not violated, then this newly available heater is switched on and set to the target temperature. The LCD display is updated to reflect the change in the number of running heaters. The JML post conditions associated with the *bindHeater()* method check whether the heater is turned on according to the conditions for economic usage. Note that post conditions from the *execute()* method $N1, N2, N3, N4, N5, H1, H2, H3$ are repeated here since they represent the economic usage conditions for the heaters. The JML invariants, $L1$ and $L2$, for the LCD, mentioned earlier in Table 2, also get evaluated to ensure the LCD display is updated correctly.

When a heater disappears, the *unbindHeater()* method in Table 3 is called by the service. To compensate for the unbound heater, the method switches on another heater, if available, in compliance with the economic usage conditions. The number of active heaters in the room and the LCD display are updated. The JML post conditions check whether the heater being unbound is switched off and if the economic usage conditions are obeyed. The JML invariants, $L1$ and $L2$, check whether the LCD display is updated with the correct number of active heaters.

4. TEST GENERATION AND EVALUATION

We evaluated our proposed approach for test oracles in the HAS by testing several dynamic reconfigurations in services. To enable us to evaluate and test our approach thoroughly, we developed a testing framework that generates tests with dynamic service reconfigurations for the HAS from a test pattern. We tailored our existing combinatorial testing tool, TOBIAS [18], to help achieve this. We monitored the JML specifications as the test cases were run to check for violations in service behavior. Additionally, we created several mutated services by seeding faults into the service so that service behavior is altered during dynamic changes. Each mutated service has a single seeded fault. We ran the test suite generated by TOBIAS against the set of mutated services, and checked whether the test oracles in the service could reveal the mutations. We say that the test oracles revealed the service mutation for the given test suite if at least one of the test cases in the test suite violated at least one of the JML specifications in the mutated service.

The tests generated by TOBIAS are sequences of method calls with different combinations of input parameter values for the methods. The input to TOBIAS is a test pattern (also called test schema) that defines the set of test cases to be generated. A test pattern is a bounded regular expression involving the Java methods in the service. TOBIAS unfolds the test pattern into a set of sequences, and then computes all combinations of the input parameters for all the methods in the pattern. The resulting test suite is converted into a JUnit ([14, 4]) file for testing services on the OSGi platform. Note that TOBIAS was previously used as a combinatorial test generation tool for traditional JAVA applications rather than service-oriented applications such as the HAS. We adapted TOBIAS to generate test suites that are executable on the OSGi service-oriented platform. Additionally, we created test patterns that exercised different dynamic reconfigurations and behavior changes in the service by placing calls to methods that made required services appear/disappear or by changing the configuration of the environment during service run-time.

4.1 Temperature Control Service: Test Oracle Evaluation

Due to space limitations, we only briefly illustrate test generation and oracle evaluation using the temperature control service in this Section. The test pattern we used to automatically generate test cases using TOBIAS for the temperature control service is described in Table 4. In the test

```
// Properties for number of active heaters in the room
// (labeled N1, N2, N3, N4, N5)
N1: //@ ensures (isrunning ==> (num_running <= m_heaters.length));
N2: //@ ensures ((isrunning && (m_heaters.length > 0) && (tempdiff < 10))
                    ==> (num_running == 1));
N3: //@ ensures ((isrunning && (m_heaters.length >= 3) && (tempdiff >= 10) && (tempdiff < 20))
                    ==> (num_running == 3));
N4: //@ ensures ((isrunning && (m_heaters.length < 3) && (tempdiff >= 10) && (tempdiff < 20))
                    ==> (num_running == m_heaters.length));
N5: //@ ensures ((isrunning && (tempdiff >= 20)) ==> (num_running == m_heaters.length));

// Heater Properties (labeled H1, H2, H3)
H1: //@ ensures isrunning ==> (\forall int i; 0<=i && i<num_running; m_heaters[i].isOn());
H2: //@ ensures isrunning ==> (\forall int i; num_running<=i && i<m_heaters.length;
                    !(m_heaters[i].isOn()));
H3: //@ ensures isrunning ==> (\forall int i; 0<=i && i<num_running;
                    m_heaters[i].getTargetedTemperature() == targetTemp);

// LCD properties (labeled L1, L2)
L1: //@ invariant (isrunning ==> m_lcd.isOn());
L2: //@ invariant (isrunning ==> m_lcd.getDisplay().equals( "Number of heaters active is " +
                    Integer.toString(num_running)));
```

Table 2: JML assertions to monitor temperature control service behavior

```
// BIND method and specifications
/*@ ensures ((isrunning && (((tempdiff < 10) && (\old(num_running) < 1))
            ||((tempdiff >= 10) && (tempdiff < 20) && (\old(num_running) < 3))
            ||(tempdiff > 20))) <==> (h.isOn() && (num_running == \old(num_running) + 1)));
@*/
//@ Repeat post conditions N1, N2, N3, N4, N5 given earlier
//@ Repeat post conditions H1, H2, H3 given earlier

private synchronized void bindHeater(Heater h) {
    if (isrunning) {
        tempdiff = tempDiff();
        if (((tempdiff < 10) && (num_running < 1))
           ||((tempdiff >= 10) && (tempdiff < 20) && (num_running < 3)) || (tempdiff > 20)){
                System.out.println("Binding Heater: " + h.getFriendlyName());
                h.turnOn();
                h.setTargetedTemperature(targetTemp);
                num_running++ ;
                m_lcd.display("Number of heaters active is " + Integer.toString(num_running));
        }
    }
  // if isrunning is false it means the execute method is not running,
  // so no updates necessary
}

// UNBIND method and specifications
//@ ensures (isrunning ==> (h.isOn() == false));
//@ Repeat post conditions N1, N2, N3, N4, N5 given earlier
//@ Repeat post conditions H1, H2, H3 given earlier

private void unbindHeater(Heater h){
    if (isrunning && h.isOn()) {
        System.out.println("Unbinding Heater: " + h.getFriendlyName());
        h.turnOff();
        num_running--;
        // Turn on another heater, if available, according to
        // temp diff and economic usage conditions
        if ((num_running < m_heaters.length) && !m_heaters[num_running].isOn() &&
           (((tempdiff < 10) && (num_running < 1))
           ||((tempdiff >= 10) && (tempdiff < 20) && (num_running < 3)) || (tempdiff > 20))){
                m_heaters[num_running].turnOn();
                m_heaters[num_running].setTargetedTemperature(targetTemp);
                num_running++ ;
        }
        m_lcd.display("Number of heaters active is " + Integer.toString(num_running));
    }
}
```

Table 3: Listener methods and JML assertions to monitor dynamic reconfigurations in temperature control service

Initial Configuration
Introduce 3 to 5 heaters Set environment temperature to 5, 20, or 80 Set desired room target temperature to 20, 40 or 100 Activate Temperature Control Service Wait for a fixed time
Dynamic Changes
Add/Remove heater Change environment temperature Wait for a fixed time Deactivate Temperature Control Service

Table 4: Informal description of the TOBIAS test pattern for Temperature Control service

pattern in Table 4, the wait times were configured to allow the service to run for a sufficiently long time so that changes in service behavior could be observed. The test pattern illustrated was unfolded into 135 test cases by TOBIAS with different combinations of input parameters. Note that it is possible to create many other test patterns, different from the one in Table 4, for the temperature control service; with different sequences of method calls, different input parameters for heater configurations and temperature settings, different numbers and combinations of dynamic changes. We chose the test pattern in Table 4 to simply illustrate our test generation and evaluation approach. We do not place any claims on the thoroughness and effectiveness of the generated test suite.

We evaluated the effectiveness of our oracles by seeding faults into the service and checking if the oracles were capable of revealing the faults. We created 25 mutated services, by *manually* seeding faults to alter behavior of the service during dynamic reconfigurations. Each mutated service had a single seeded fault. We ran the test suite generated for the test pattern in Table 4 over each of the 25 mutated services and checked if any of the JML specifications in the mutated service were violated. We found that for 23 of the 25 mutated services, JML specifications were violated revealing the mutations in the service. Thus, for the given test suite of 135 test cases, our approach for test oracles was effective in revealing 23 of the 25 seeded faults. On closer examination of the two undetected faulty scenarios, we found that the test suite did not exercise the scenarios involving the two seeded faults. To overcome this weakness in the test suite, we manually created test cases that exercised the two faulty scenarios. The newly created test cases violated the JML specifications for the bindHeater() listener method. We could thus reveal all 25 mutations with our test oracles and the test suite augmented with the newly created test cases. The evaluation clearly showed that the JML specifications associated with the bind/unbind listener methods were effective in revealing erroneous behaviors during reconfigurations.In our future work, we plan to explore test generation for SOC applications like the HAS in more depth.

4.2 Threats to Validity

We face two threats to the validity of our evaluation. The first one is with regard to the properties that can be expressed with JML. In our example, the reconfiguration properties specified with JML were either static properties expressed as invariants, or properties only valid in the initial or final states expressed as pre or postconditions. We also specified dynamic properties involving current and previous states in the example (using the `\old` clause). Nevertheless, we did not explore properties where the current behavior is dependent on behavior that occurred past the previous state, i.e. the system has some memory of the behavior history and reacts differently to an event based on the history. For example in the temperature control service, when a heater appears, the service may be required to react differently based on whether that heater was already seen before or not. The JML specification language does not support operators to express such temporal properties. This issue, however, can be overcome by using the approach proposed by Bellegarde et al. [5] that translates such temporal properties into an equivalent set of JML annotations. The second external threat is that the example service, test suite, and number of mutations used in our evaluation are relatively small when compared to an evaluation over an industrial system. Nevertheless, this is only a preliminary evaluation that helped show that our approach for test oracles holds promise. We plan to pursue a more extensive evaluation on real world examples in the future.

5. RELATED WORK

The run-time monitoring challenges encountered in systems composed of web services are closely related to the monitoring challenges in the HAS since both applications encounter dynamic reconfigurations in services. Run-time monitoring of properties in web services has been explored in the past.

Spanoudakis et al. [21] proposed a framework for run-time monitoring of requirements, expressed in event calculus, for web-service compositions. Baresi et al. [3] also proposed an approach for run-time monitoring web service compositions. They monitor whether the external service selected by the composition process conforms to the behavior expected from it. Ghezzi et al. [12] proposed an approach, Dynamo, to specify constraints and monitor collaborations with external services. Dynamo monitors whether the external services that it collaborates with deliver what is expected of them.

The related work in the web services domain primarily focuses on monitoring web service compositions. Web service compositions are managed by a composition process specified in languages like BPEL [13] or WSCDL [15] (depending on the collaboration model chosen). A composition process, as defined by Mahbub et al. [19], is one that coordinates external web services that get deployed in a service-oriented system. The composition process provides the required system functionality by calling operations in the external web services, receiving and processing the results that these services return, and accepting and/or responding to requests from them. All the proposed monitoring approaches in the web services domain rely heavily on the composition process, and monitor whether the external web service selected by the composition process adheres to the behavior expected from it. Our proposed approach for monitoring the HAS differs from these existing approaches in two fundamental ways. One, our approach aims at monitoring the behavior of the integrated service that uses other external services. Unlike existing approaches, our monitoring approach is not concerned with the selection mechanism and behavior of external services. For instance in the temperature control service, we monitored behavior of the service in the presence of changes to the external heater services. We do not

monitor whether the heater and LCD services that it uses are selected according to requirements and function as expected. Two, the H-Omega gateway that we use to deploy and provide services, transparently manages interaction between services. There is no explicit composition process. As a result, our approach for run-time monitoring is independent of a composition process, and instead employs listener methods interacting with the service architecture to help verify dynamic service behavior.

6. CONCLUSION

In this paper, we proposed an approach to address one of the challenges in testing home automation systems—specifying test oracles that monitor service behavior in the presence of dynamic service reconfigurations. We formally specify test oracles for the HAS using the JML specification language. We use JML specifications as run-time monitors of the service behavior. The main challenge in run-time monitoring is in identifying "visible" states for evaluating the specifications during program execution. We provide the visible states for specification evaluation using *listener methods* that are associated to dynamic events in the service architecture. We combine this capability with JML run-time assertion evaluation to monitor service behavior during dynamic reconfigurations. We illustrated our approach with an example service in the HAS.

We conducted an initial evaluation of our proposed approach for test oracles by testing several dynamic service reconfigurations in the example service. We adapted our existing combinatorial testing tool, TOBIAS, over JAVA applications to generate tests with dynamic service reconfigurations for the HAS. We ran the test suite from TOBIAS against 25 mutated versions of the example service to evaluate the fault finding capability of the test oracles. We found that the test oracles could reveal all 25 mutations. From our preliminary evaluation, we believe our proposed approach provides a useful and effective means for defining test oracles that monitor service behavior in the presence of dynamic reconfigurations for the HAS. We plan to conduct a more extensive evaluation of our test oracle approach on real world example systems in our future work.

In this paper, we have only explored the applicability of our approach to the HAS implemented using the H-Omega service architecture. Nevertheless, it is straightforward to see that our approach can be used in a like manner for other service-oriented applications implemented using the H-Omega service architecture. To apply our approach to other service architectures, we would need to utilize the appropriate mechanisms in the underlying architecture for notification of dynamic changes. We plan to extend our approach to include other architectures in our future work.

7. REFERENCES

[1] Apache felix iPOJO website. http://felix.apache.org/site/apache-felix-ipojo.html.

[2] OSGi Alliance. *OSGi Service Platform: Release 3, March 2003*. IOS Press, 2003.

[3] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOCŠ04*, pages 193–202, 2004.

[4] K. Beck and E. Gamma. Test infected: Programmers love writing tests. Java Report 3(7), July 1998.

[5] F. Bellegarde, J. Groslambert, M. Huisman, J. Julliand, and O. Kouchnarenko. Verification of liveness properties with JML. Technical report, INRIA.

[6] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin. A dynamic-soa home control gateway. In *IEEE International Conference on Services Computing (SCC 2006)*, 2006.

[7] Y. Cheon and G. T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328, Las Vegas, Nevada, June 2002. CSREA Press.

[8] Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *16th European Conference on Object-Oriented Programming (ECOOP'02)*, number 2374 in LNCS, pages 231–255. Springer, June 2002.

[9] C. Escoffier, J. Bourcier, P. Lalanda, and Jianqi Yu. Towards a home application server. In *5th IEEE Consumer Communications and Networking Conference*, pages 321–325, January 2008.

[10] C. Escoffier, R.S. Hall, and P. Lalanda. iPOJO: an extensible service-oriented component framework. In *IEEE International Conference on Services Computing (SCC 2007)*, pages 474–481, July 2007.

[11] S. Fickas and M.S. Feather. Requirements monitoring in dynamic environments. In *Proc. of the Second IEEE International Symposium on Requirements Engineering*, pages 140–147, March 1995.

[12] C. Ghezzi and S. Guinea. *Run-time monitoring in service-oriented architectures*. Test and Analysis of Web Services, 2007.

[13] IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. Business Process Execution Language for Web Services 1.1, 2005.

[14] JUnit. http://www.junit.org.

[15] N. Kavantzas, D. Burdett, and G. Ritzinger. Web Services Choreography Description Language version 1.0, 2004.

[16] M. Kim, S. Kannan, I.Lee, O. Sokolosky, and M. Viswanathan. JAVA-MAC: a runtime assurance tool for java programs. *Electronic Notes in Theoretical Computer Science*, 55, 2001.

[17] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Muller, J. R. Kiniry, and P. Chalin. *JML Reference Manual*. Iowa State University, Jan 2006.

[18] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering TOBIAS combinatorial test suites. In *7th Int. Conf. FASE, Held as Part of ETAPS*, volume 2984 of *LNCS*, pages 281–294, Barcelona, Spain, 2004.

[19] K. Mahbub and G. Spanoudakis. *Monitoring WS-Agreements: An Event Calculus-Based Approach*. Test and Analysis of Web Services, 2007.

[20] D.K. Peters and D.L. Parnas. Requirements-based monitors for real-time systems. *IEEE Trans. Softw. Eng.*, 28(2):146–158, 2002.

[21] G. Spanoudakis and K. Mahbub. Requirements monitoring for service-based systems:towards a framework based on event calculus. In *Proceedings of the 19th International Conference on Automated Software Engineering*, 2004.

PicOS Tuples: Easing Event Based Programming in Tiny Pervasive Systems

Benny Shimony, Ioanis Nikolaidis, Pawel Gburzynski, Eleni Stroulia
Department of Computing Science
University of Alberta, Edmonton, Alberta
Canada T6G 2E8
{shimony, yannis, pawel, stroulia}@cs.ualberta.ca

ABSTRACT

The task of programming sensor-based systems comes with severe constraints on the resources, typically memory, CPU power, and energy. The challenge is usually addressed with techniques that result in poor code understandability and maintainability. In this paper, we report on a data centric language extension based on a tuple-space abstraction, akin to Linda [2], applied to PicOS [5], a programming environment for wireless sensor networks (WSN's). The extension improves state and context management in a multi-tasking environment suffering from severe memory limitations. The solution integrates tuple operations into the model – for networking, event handling, and thread contexts. We demonstrate how tuple constructs improve coding and reduce code overhead. We also show how thread's context-tuples can be used as interface arguments for extension by modular aspect constructs.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*data abstraction,domain-specific architectures, languages*; D.1.3 [**Concurrent Programming**]: Distributed programming

General Terms

Design

1. INTRODUCTION

To be practically viable, a typical wireless sensor network must be characterized by a low cost of its components combined with minimalistic energy requirements [7]. These constraints translate into some physical properties of hardware painfully perceptible by the programmer: small amount of memory (RAM), code size limitations (ROM), limited CPU power, and so on, which drastically affect the programming characteristics of building WSN applications.

From the functional point of view, the environment of a WSN node is reactive (rather than computationally-intensive), which is good news. However, it can still be quite complex, *e.g.,* requiring the node to respond in a timely fashion to complicated configurations of events, with multiple events (possibly of different types) occurring at (almost) the same time. This calls for multitasking capabilities of the node's program.

In this setting, the event-based programming paradigm has been the preferred choice of the most popular programming platforms for WSN nodes [7]. This paradigm has been demonstrated to be efficient in terms of energy usage, memory footprint, and concurrent task management. However, these advantages come at a serious detriment to system design qualities, such as code understandability, maintainability, and extensibility [13]. In particular, the tight RAM budget renders classical multi-threading methods impractical, as they tend to squander memory for multiple and largely fragmented stacks. As a result, alternative coding structures are used to handle multi-tasking. Typically, conceptual tasks are broken into *de facto* independent event-handling procedures that share a single (global) stack with no possibility of preserving there a task-specific context. Hacks mitigating this problem are known under the collective name of *manual stack management* (MSM [1]). Common techniques rely on global variables or dynamic heap memory, which are both inefficient in a RAM-tight environment. Moreover, they are bad practice from the viewpoints of encapsulation, modularity, and code readability, since they incur code overhead to co-ordinate the shared data while also being counter-intuitive and prone to errors [13].

To set the stage for our work, we review and compare two programming styles: 1) a typical event-based system akin to TinyOs [12] vs. 2) PicOS [5], a locally-developed, low-overhead operating system for sensor nodes. In both environments, limited memory imposes a non-persistent stack space. TinyOS, the most popular environment for WSN programming today, accommodates a single thread (termed task), which executes at a low priority, while interrupt handlers account for most of the dynamics. In PicOS, a lean mechanism of *cooperative threading* is used to support a finite state machine (FSM) task management abstraction.

Our solution put forward in this paper adopts a data-centric methodology relying on a tuple-space abstraction. That abstraction makes it possible to define named data structures that can be easily handled and shared (a) across multiple

local tasks and (b) across a network neighborhood using mirrored tuple spaces. Tasks can coordinate their activities by accessing the (shared) tuples via associative memory operations for retrieving, removing, and adding data. The effectiveness and elegance of this paradigm have been demonstrated in Linda's distributed programs [2], and it adapts well to concurrent task programming, especially in a well-coupled (local) setting. In this paper, we show how the basic idea of Linda's context matching operations fits naturally the reactive environment of WSNs. Moreover, by implementing a local tuple repository with additional support for thread *context tuples*, we eliminate the need for global variables and (explicit) heap space, as detailed in sections 3, 4. Finally, a flexible mechanism for matching and setting the thread *context* and its event triggering conditions plays a central role in extensions/modifications of programs. We view this mechanism as an interaction point for modular *rule* refinement along the line of *aspect* based constructs. The approach is related to *cooperative aspect-oriented programming* (Co-AOP [8]), as will be detailed in sections 5 and 6.

This work makes two important contributions. First, it integrates tuple-space operations into the existing PicOS-threads programming model – by adding language constructs and services to the event-based semantics of the system. Second, it introduces thread *context tuples* as a standard interface to manage thread state context (mitigating limitations on stack use), while providing an interface for modular extensions.

2. BACKGROUND AND EXAMPLE

PicOS [5] provides a lean multi-tasking mechanism derived from an FSM abstraction, around which elaborate rule-driven routing protocols can be developed [4]. In this section, we discuss the PicOS programming paradigm in comparison to TinyOS.

The two coding approaches are exemplified in Figures 1 and 2. The original code has been borrowed from [9]; its goal is to collect temperature samples from neighboring nodes and compare their average to a local temperature reading. Conceptually, the program at the collecting node operates in three phases: 1) sending a request, 2) waiting for replies (temperature readings), and 3) producing the result after a timeout.

2.1 The original code

As listed in Figure 1, the code includes a setup function **init_remote_compare** (lines 4-8) and two event-handling functions **message_hdl** (lines 9-13) and **timeout_hdl** (lines 14-18). In the first phase, the node broadcasts a request message to its neighboring nodes prompting them for reports of their temperature readings (line 6). A report received in response to that request triggers an invocation of **message_hdl**. The function is responsible for aggregating the received data (lines 9-13). When the timeout expires, the handler (**timeout_hdl**) compares the average of the remote temperature readings collected in the meantime to the local temperature (lines 16-18).

Note that variables **num** and **sum**, manipulated by **message_hdl** and **timeout_hdl**, realize a variant of MSM. Since the local stack is unrolled after every action, the local state is not persistent, and the programmer must store the relevant data explicitly: either globally and statically (as in the above example), or dynamically (on the heap). Both solutions are flawed. With the first one, the memory is statically and permanently locked at one task. The latter approach is ridden with various overheads and susceptible to fragmentation, which affects all heap-based allocation techniques and is particularly painful in a non-preemptive (thread-less) environment with tight RAM. This is because such a system must be prepared to deal with a (temporary) unavailability of dynamic memory. That, in turn, assumes that the requesting entity is able to block half-way through its action (no memory) and resume later (on a *memory available* event) as if "nothing happened", which gets us back to the problem of automatic and safe context preservation (for which a per-task stack appears as the most natural solution). The primary reason why TinyOS implements no heap is that its activities cannot block half-way through and thus cannot sensibly wait for memory events.

Another generic way to mitigate the complexity of detailed context preservation, known as *manual flow control* (MFC), is illustrated by the usage of variable **sampling_active**. That variable acts as a coordinating flag between the handlers **message_hdl** and **timeout_hdl** (lines 3, 5, 10, 15). Note that by setting the flag to *false*, **timeout_hdl** disables the other handler (line 10).

2.2 The PicOS version

PicOS defines a flavor of threads. A single thread can describe a number of different event-response actions selected by its dynamic *state*. Following the model of coroutines, the states tag the thread's *entry points* (the **entry** statements in Figure 2), which are the only points where the thread's execution may commence. Cooperative multi-threading takes place as threads explicitly yield the CPU, usually at the end of a state code section, using the **release** command. In order to resume a thread's flow of control, each yield is typically preceded by one or more event wait operations (called *wait requests*). A single wait request specifies an event (that the thread wants to wait for) together with the state to be assumed (entered) by the thread when the event occurs. For example the state INIT in Figure 2 includes two wait requests: one with **when** and the other with **delay** (lines 7-8) operations. With the former, the thread declares that it wants to be resumed in state COLLECT upon the occurrence of an event represented by the address of a message object (buffer). Such events can be signaled by explicit trigger operations (not shown here). The delay operation sets up an alarm clock for the specified number of milliseconds (1000). The event waking the process at state STOP will be triggered when the alarm clock goes off. The action of initiating a new data collection is captured in the invocation of **request_remote_temp** (line 6). While at first sight that statement might stand for something as simple as broadcasting a single *request message* into the neighborhood, we prefer to make the operation more elaborate. In a realistic implementation, the request message will be broadcast several times (at some intervals), and (most conveniently) by a separate thread, in order to maximize the likelihood that all neighbors have perceived the request.

```
1   int sum = 0;
2   int num = 0;
3   bool sampling active=FALSE;
4   void init_remote_compare() {
5       sampling active=TRUE;
6       request_remote_temp();
7       register_timeout( 1000 );
8   }
9   void message_hdl( MSG msg ) {
10      if(sampling active==FALSE) return;
11      sum = sum + msg.value;
12      num++;
13  }
14  void timeout_hdl() {
15      sampling active=FALSE;
16      int val = read_temp();
17      int average = sum / num;
18      if( average > val ) /* ... */
19  }
```

Figure 1: TinyOS related code [9]

```
1   int sum = 0;
2   int num = 0;
3   int msg_ev;
4   thread (TempAverage)
5   entry (INIT)
6       request_remote_temp();
7       delay ( 1000, STOP);
8       when (&msg_ev, COLLECT);
9       release;
10  entry (COLLECT)
11      sum = sum + msg.value;
12      num++;
13      when (&msg_ev, COLLECT);
14      snooze (STOP);
15      release;
16  entry (STOP)
17      int val = read_temp();
18      int average = sum / num;
19      if( average > val ) /* ... */
20  end thread;
```

Figure 2: PicOS code

```
1   <tuple_type> get <tuple_name> ( field_x == value|'*', ... )
2   <tuple_type> remove <tuple_name> ( field_x == value|'*', ... )
3   List<tuple_type > get_group <tuple_name> (field_x == value|'*', ... )
4   List<tuple_type > remove_group <tuple_name> (field_x == value|'*', ... )
5   <tuple_type> set <tuple_name> [ field_x = value| ⊥ , ... ]
6   <tuple_type> tuple_send <tuple_name> [ field_x = value| ⊥ , ... ]

7   define context{ tuple_name, ... }
8   when_tuple( tuple_name, next_state, boolean (*filter_func) )
```

Figure 3: PicOS tuple operations

If multiple events are awaited by the thread, the earliest of them will wake it up. Once that happens, all the pending wait requests are erased, an so the thread has to specify them from scratch at every wake-up. For example in Figure 2 when the thread resumes in state COLLECT (after a message event trigger), it preforms aggregation of data in lines 11-12, while in lines 13-14 it re-issues the wait requests[1] before relinquishing the CPU. Finally, once a time-out event occurs, state STOP is assumed, which concludes the protocol cycle. Notice, how the task stages in the PicOS code can be explicitly organized into an FSM form, which obviates the need for state management flags (like **sampling_active** in the traditional variant).

While *flow control management* in PicOS is more convenient compared to the TinyOS model, a PicOS thread yielding the CPU also relinquishes its stack, which forces the programmer to resort to MSM schemes. Hence, the global variables **num** and **sum** have been carried over the previous solution (to be used for essentially the same purpose). Moreover, as the program evolves and its requirements (data and control structures) grow, the overhead and complexity of accommodating the new cases into the existing mess of global variables and flags become more and more difficult to manage. As noted in [12], programming can become tricky when the system needs to be extended, especially when global data requires concurrency control and locking.

[1]Operation **snooze** sets up the alarm clock for the residual time from the previous **delay**.

3. PICOS THREADS AND TUPLES

Figure 3 outlines the tuple interface that we have added to PicOS. For brevity and clarity the functions are presented in a pseudo-code at a high level. The operations are divided into two sections: the top section includes access operations to the tuple space, while the bottom part lists the *local* operations, which are thread-related. The underlying facilitator of the tuple abstraction, is the use of a *local repository*. The repository acts as the focal point of data sharing and communication in the multi-tasking environment – both locally and for the immediate radio-range neighborhood. First it stores new received tuples and manages tuple buffer allocations in the repository. Then, it propagates updates as *tuple events* to the local program, so the new data can be acted upon. In the local setting, the multi-threaded abstraction of PicOS engages the tuple space in the same fashion as Linda [2]. In the network scope, data can be shared through a *send* operation which (unreliably) mirrors/copies the tuples to neighboring nodes. A remotely received tuple is added to the local repository as if a remote **set** operation was preformed. Thus, the underlying tuple framework provides a common interface to handle shared data in the distributed setting.

Operations **get** and **remove** correspond to Linda's **rd** and **in**. They accept templates for matching tuples: each field can either provide a specific (required) value or a wildcard **'*'**. Operation **remove** is similar to **get**: they both retrieve a matching tuple from the repository, with **remove** additionally removing the tuple. If several tuples match the arguments of **get**/**remove**, one of them is chosen nondeterministically. If no matching tuple is found, the function returns *null*. Operations **get_group** and **remove_group** retrieve sets of matching tuples.

Operation **set** corresponds to Linda's **out**, which inserts a tuple into the repository. Each field can either provide a valid (assigned) value or be undefined '⊥' (by default). With **send_tuple**, which is similar to **set**, the program inserts the tuple into the local repository and in addition broadcasts the tuple as a message over the RF channel, which may update (some of) the tuple repositories at neighboring nodes. At present, this operation is asymmetric: the issuing node does not know who will receive the update (there is no guarantee of delivery).

The **define context** construct is part of the thread header. Its role is to define the categories of tuples relevant to the particular thread, which are then deemed to belong to its *context*. As we explain later, context tuples can be automatically assigned by tuple events when waking up threads.

Operation **when_tuple** is a variant of PicOS's generic **wait** request that allows the issuing thread to await a tuple event triggered when a new tuple appears in the local repository. The filter function (provided by the programmer) can describe elaborate qualifying conditions that must be met by a tuple to trigger the event.

Some notable syntactic detail is that in operations **set** and **tuple_send**, we use square brackets [...] to encapsulate associative assignments of data to tuple fields. In the remaining operations from the upper section, the arguments in round parentheses (...) describe field patterns to be matched to retrieved tuples. Individual tuple fields can be accessed through a mathematical projection notation, *e.g.,* *field_name*(*tuple instance*).

4. THE EXAMPLE REVISITED

Figure 4 lists the temperature collection example reprogrammed using PicOS tuples. Note that the comments in lines 1 and 20 annotate for "option **b**", which will be discussed later. Basically, Figure 4 covers two slightly different versions of the code, which have been merged for brevity. We start from the simpler version (option **a**).

Note that the global variables **num** and **sum** have been replaced by a tuple, **aggData**, and the temperature readings are now represented by another tuple, **temperature** (lines 1-2). One more tuple, **reqTemp** (line 3), describes requests sent to neighboring nodes. Such a request includes a sessions identifier (line 9) used to match readings to requests.

The thread includes a context definition (line 5) describing the tuples to be handled by the thread. Such definitions can be viewed as providing placeholders for tuples with some specific layouts, each placeholder occurring in two instances (offering two slots): *current* and *new*. The *current* variant accommodates a local tuple (one residing in the node's local repository), while the *new* part is typically filled by received (retrieved) tuples, possibly triggering tuple events. Similar to the previous version, the action of initiating a new data collection (line 10) may involve multiple invocations of **tuple_send** (from a separate thread).

By calling **when_tuple** (in line 12 and subsequently in line 21) the thread declares that it wants to be resumed in state COLLECT whenever a tuple qualified by **filter_func** appears in the node's view (this may be a tuple received as a message from a neighboring node). The filter function for option **a** (Figure 5) rejects tuples tagged with the wrong session identifiers (i.e., belonging to a different collection). A typical filter function compares (some of) the attributes of the incoming (*new*) tuple to the corresponding attributes of the *current* variant. The collected temperature readings are aggregated in state COLLECT (lines 15-19).

Another point to notice is that the context's *current* instance persists across state invocations; in the present version of the system, the programmer is responsible for the integrity of current context setting as well as the specification of predicates triggering the tuple condition (which wakes up the thread in the respective state). As will be shown in option **b**, such settings can be carried out from within the **filter_func** code.

Operational model: filtering. Tuple events are generally scheduled using a FIFO model. Any tuple appearing at the head of the FIFO queue that is not explicitly waited for (does not match any filter function of a pending **wait_tuple** request) is removed from the queue and ignored (dropped). Formally, such a tuple is not relevant to any of the current contexts and thus not needed by the program. This process becoms a natural (pre)filtering mechanism whose semantics seem to well match a tight-memory implementation (as the program doesn't have to worry about storing outstanding tuples). A more lax scheme is possible whereby tuples are stored for a limited time, such that they can be accessed later (when their contexts come to life). The timeout may be explicit; alternatively, a certain dedicated amount of memory (buffer pool) can be set aside to accommodate the unclaimed tuples. The pool can be recycled as needed to accommodate new tuples, *e.g.,* by discarding those that have remained untouched for the longest time.

An extension: per-node average. At first sight, it may appear that the migration to the tuple-space paradigm adds unnecessary overhead to our otherwise simple application. To see the benefits of this paradigm, consider an additional requirement: we would like to keep track of the average temperature at every reporting node, and derive a total average of averages at the end of the collection process. Notably, this kind of "enhancement" of the original solution (Figure 4) can be achieved quite easily by switching to option **b**, which involves simple changes in lines 1 and 20 and replacing the filter function. With the modification in line 1, we extend the original layout of **aggData** by an **ownerId** field, to be able to associate the data stored in the tuple with a specific node (representing the per-node average). The change in line 20 makes sure that the **set** command references the

```
1   define tuple aggData [ int sum, int num]  /* option b: [..., nid ownerId] */
2   define tuple temperature [nid owner, int sessId, int value]
3   define tuple reqTemp [int sessId]

4   thread (TempAverage)
5   define context{ aggData, temperature }
6   entry (INIT)
7      set current.aggData [ sum = 0, num = 0];
8      set current.temperature [ owner = nodeId, sessId =1, value = ⊥ ]
9      set current.reqTemp [ sessId =1]
10     request_remote_temp( current.reqTemp );
11     delay ( 1000, STOP);
12     when_tuple ( temperature, COLLECT,filter_func(&current, &new ) )
13     release;
14  entry (COLLECT)
15     if (owner(new.temperature) != nodeId) {
16       int sum_l = sum(current.aggData) + value(new.temperature);
17       int num_l = num( current.aggData)++;
18       set current.aggData[ sum = sum_l, num = num_l ]  /* option b: [ ..,ownerId = owner(new.temperature)] */
19       remove temperature;}
20     snooze (STOP);
21     when_tuple ( temperature, COLLECT, filter_func(&current, &new ) );
22     release;
23  entry (STOP)
24     current.temperature = get temperature( owner == nodeId, * ) //context pattern
25     int average = sum(current.aggData) / num(current.aggData);
26     if( average > ( value(temperature) ) /* ... */
27  end thread;
```

Figure 4: PicOS Tuples - Temperature Collection

```
boolean filter_temp (byte *current, byte *new)
{
  if ( sessId(current.temperature)!=
            sessId(new.temperature) )
       return TRUE; // drop tuple event
  else
       return FALSE; // pass event
}
```

Figure 5: a. Simple filter function

```
boolean filter_temp (byte *current, byte *new)
{
  if ( sessId(current.temperature)!=
            sessId(new.temperature) ) {
         return TRUE; // drop tuple event
  }
  current.aggData = get aggData
       ( ownerId==owner(new.temperature), *, *)
  if ( current.aggData == NULL ) {
     current.aggData = set aggData
               [ 0, 0, ownerId = owner(new.temperature)]
  }
  return FALSE;
 }
```

Figure 6: b. Extended filter function

new field in **aggData**. Finally, we have to take care of **aggData**'s context. Note that previously the node dealt with a single tuple of this layout (storing the global variables **sum** and **num**), while now a separate tuple will be needed for every neighbor. As we do not know the identities of all those neighbors in advance, we shall use a new filter function (see Figure 6). In addition to the the previous test, the new function includes a **get** command to query the (local) repository for the context of the reporting node. If it doesn't exist, then we have received the first report from this particular neighbor, and we have to set up a new entry. In any case, by the time the event is triggered (and the thread wakes up in state COLLECT) the context is set properly to the tuple of the reporting node. Despite its utmost simplicity, the example well illustrates the power of the context-setting operation: the modification at state COLLECT is extremely simple and intuitively clear, even though the action carried out at that state is now considerably different from the previous version. One more easy modification (skipped for brevity) needed to make the code complete is in state STOP, where the final calculation must now be based on the per-node averages.

Summary of benefits. The new approach has eliminated the following (displeasing) elements from the original version: ***a)*** the need to explicitly replicate global variables for each context of reporting nodes; ***b)*** conditional control instructions (or indexing) in each state and retrieval of the relevant context or, alternatively, spawning multiple threads for each context (which would be memory-expensive); ***c)*** the need to decode every message in a special thread and coordinate among the multiple threads (via explicit triggers)

As noted in [12], the major difficulty in program extension is preventing race conditions when accessing shared state. In our solution, shared state is managed through the use of a centralized local repository. Section 7 details the implementation, which in essence specializes an existing buffer management mechanism of the PicOS kernel. Moreover, al-

most no overhead code is needed to translate the new code constructs to basic PicOS code.

5. THREAD CONTEXT ASPECTS

As demonstrated, programs expressed in the tuple-space paradigm can be modified by changing the conditions triggering tuple events with minor, if any, modifications to the "proper" code. Consequently, a good way of integrating useful/popular features into the system must assume that those conditions amount to a powerful programming tool. In particular, it makes sense to view them as modular *rules* and provide for some natural mechanism of applying them. For example, the filtering conditions in some routing protocols for WSN [4] have the form of a chain of rules that are applied sequentially to every received packet, stopping as soon as a reason is found not to rebroadcast it. The general principle of such schemes is to utilize cached data as the knowledge base for the rules. In the absence of data, the rules will *fail* to find reasons for *not* forwarding, so the protocol will operate redundantly (erring on the side of over-eager, but otherwise harmless, collaboration). As more data (knowledge) is collected, the rules will *succeed* more often, thus trimming down the (redundant) retransmissions and improving the quality of routing.

Our data-centric approach boils down to a rather simple filtering process whereby context tuples are passed as parameters to filter functions appended to the basic **wait_tuple** trigger conditions. We suggest that by refining these conditions and organizing them into configurable chains we can relatively easily build a wide range of different, very dynamic, possibly quite complex, distributed algorithms runnable on possibly large sets of resource-constrained communicating nodes. Inspired by *aspects* in the synchronous framework [6], we have found that those aspects that are categorized as *regulative advices* [10], i.e., ones that only restrict the reach space of the base system, can be effectively formed within our system.

We define an *aspect advice* as a set of additional conditions and functions at PicOS state boundaries, which, in our system, can be incorporated as part of the triggering conditions for **when_tuple** operation. Their parameters are exactly the same as the **filter_func** used within the **when_tuple** definition (i.e., **current**, **new**), which gives them the same kind of access to the context data. Needless to say, it is easy to organize them into a chain of rules for natural sequential evaluation. The additional functionality is configurable through the use of a syntactic tool applied in the compilation phase and using a certain template format for describing aspects in the FSM framework (adapted from [6]).

Temperature collection with regulative aspects. Let us revisit once again the temperature collection example and consider a modification whereby the temperature reports are solicited from a specific subset of the neighboring nodes. The selected subset is determined by a *neighbor-group service*, which sets a node group tuple in the repository. The layout of that tuple is:

tuple *nbr_group* = (nid nbr_1, nid nbr_2, ..., nid nbr_n)

We can now describe the requested additional behavior with the aspect template definition *Temp_Group* and its corresponding **filter_advice** function listed in Figure 7 (left and right side correspondingly). The aspect defines point-cuts on transition points expressed by a **wait_tuple** system call (lines 4-5), which covers two join-points from the base program (see lines 12 and 21 in Figure 4). Note the wild card option applied to the state field. If required, we can easily add an option to restrict join-points to specific thread instances and states (or their subsets). The transformation advice (**TRNS_ADVISE** in lines 6-7) adds a conjunct filter function, **filter_advice**, to the event filter function from the original program. With the new function, if the temperature reporting source is not in the *nbr_group* tuple, the event is dropped. Note that the transition advice is actually a "do-nothing" advice, which effectively restricts the reach space of the base program. Thus, we attain the required functionality with no need to change the original code. This functionality can be compared to AspectJ's [11] **around** operation where there is no use of the **proceed** call.

Comments and summary. The regulative aspect type appears to be useful and natural in those cases when the original functionality has to be trimmed down, possibly in a complex manner. While this may be natural in many problems, especially when the base solution has been devised as reasonably general, the regulative type is rather counter-intuitive to most aspect advices (although possible with AspectJ's **around** operation), since it restricts rather than adds functionality. A point to notice is that the default **filter_func** used in the definition of **when_tuple** has a similar role to **filter_advice** functions, except for its additional side-effect of setting the current context (using **current** parameter by a reference). Thus, should the programmer be interested in exercising complete control over state-transitions through advice constructs, she can define a trivial **filter_func** in the base program and override context settings in the consecutive stages of advice. We also plan to include in the filtering advice an implication action (through a $\rightarrow$ operation) added to the aspect template, which will kick in when the advice fails (i.e., the filter passes the data). This implication action can be used to add an independent advice on transitions complementing the restrictive form.

Finally, in order to ensure that aspects do not cause faults in the base program, we note that the advice code needs to adhere to the basic requirements of PicOS threads, i.e., never block within a state and avoid hogging the CPU with extensive computations. This way the CPU never gets locked in a single task and the system appears responsive to all threads. Another responsibility of the aspect programmer is to make sure that the advice code "safely" updates the program's variables, i.e., in a way that does not affect the correctness of the base program.

6. RELATED WORK

The concepts of Linda [2] have been extensively investigated and implemented in various platforms. KLAIM [18] is a calculi based coordination language, that extends the tuple-space abstraction with localities primitives. The tuple-space is split to into multiple localities making it possible to handle data and computation in the distributed and mobile environments. A related approach can be found in LIME [16], which

```
1  Thread Aspect Temp_Group
2  VAR: // base program
3    context { temperature, aggData}
4  POINTCUT: // transition type
5    when_tuple ( temperature, *, filter_*)
6  TRNS_ADVISE : // add filter function
7      ...and filter_advice()
```

```
boolean filter_advice (byte *current, byte *new)
{
  nbrs =get nbr_group ;
  if ( owner(new.temperature) ∉ nbrs )
     return TRUE; // drop event
  else return FALSE; // do nothing
}
```

Figure 7: Aspect template of temperature collection example.

also adjusts the global tuple space to fragmented transient local tuples. The latter sematics are modeled by state-based logic based on Mobile Unity [17]. Both platforms have been subject to several domain-specific extensions and implementations.

Our approach to network tuple sharing resembles the Hood neighborhood abstraction [15]. In that view, local data can be mirrored to a set of neighborhood nodes through a primitive RF broadcast operation. The mechanism of group sharing corresponds directly to the asymmetric (unreliable) nature of the broadcast medium, i.e., a node is not necessarily aware of the neighbors that mirror its data (whose configuration can dynamically change). In addition, the Hood abstraction provides a data filtering mechanism which is used both for data sharing and neighborhood discovery. This approach can be paralleled to our rule based system of event-filtering. However, the multi-tasking environment of Hood, implemented on top of TinyOS as a set of language components, drastically differs from our scheme. Application design requires components to be glued (or "wired" in the language's terminology) through event interfaces reminiscent of TinyOS's event based programming, and its MFC challenges.

The TeenyLIME framework [3] preserves the semantics of tuple-space operations in the (one-hop) neighborhood. On the one hand, this relieves the programmer from the burden of managing the configuration changes (and manually tracking the neighborhoods, *e.g.,* in the face of node mobility). On the other hand, the framework comes with a hardwired (and rather complex) feature which may be superfluous in many applications. TeenyLIME is implemented on top of TinyOS as middleware that provides an API for Linda operations. The middleware is a monolithic component with a fixed set of underlying algorithms. Our approach is more low-level at its current scope, while sharing many of the motivations of the previous. We believe that our present implementation can be extended to support additional features, as configuration of reliability of tuple spaces within the neighborhood, if such features are indeed called for by the application. In fact, our experience dictates otherwise: a WSN application works at its best, if it never assumes that a sizable group of nodes can operate together as a reliable cohort for any nontrivial amount of time. In this context, a built-in reliable non-local tuple space does not appeal to us as a desirable feature.

Another approach somewhat related to ours transpires in the FACTS language [14], which is both reactive and rule-based. The high-level rules are compiled to run over a middleware layer. While our approach shares with FACTS the basic idea of tuples, our execution model is strongly integrated with PicOS threads and thus de facto becomes a component of the kernel software stack. Even though the high-level flavor of FACTS rules may have some aesthetic appeal, it also renders those rules less flexible and introduces an extra layer of complexity. Based on [14], one can suspect that the middleware rule evaluation engine of FACTS consumes a significant share of RAM. The framework only supports a primitive **send** operation for data sharing. No details regarding the underlying MAC scheme or the reliability of abstractions are provided.

Our modular extensions by regulative aspects overlaps with Co-AOP concepts [8]. This is a general framework for specifying explicit join points (EJP) with abstract interfaces to define interaction scopes of aspect advice with the base code. The authors claim that their approach improves the extensibility of code. Our system provides a fixed scheme of aspect interaction points with the base code, which are inserted at state boundaries as explicit join points. Moreover, context tuples act as exposed arguments for the aspect interaction.

7. IMPLEMENTATION STATUS

PicOS avoids layers (and, consequently, the concept of middleware) while promoting a flavor of plug-ins as the preferred way of incorporating new functionality into the kernel [5]. Plug-ins are *inserted* into a system module dubbed VNETI (Versatile NETwork Interface) depicted in Figure 8. Our tuples environment has been prototyped as a plug-in service taking advantage of the built-in buffer management mechanism of VNETI. Owing to the fact that PicOS threads (in contrast to activities in TinyOS) can comfortably block at state boundaries, PicOS can afford a lightweight dynamic memory allocator, which is used by VNETI to create flexible pools of transparent linked buffers. Those buffers serve as the basis for our repository implementation of the tuple space, both local and distributed (one-hop neighborhoods). The latter is an almost free feature stemming from the fact that selected buffers from VNETI pools can be (automatically) queued for transmission over the RF.

Concepts of modular extensions through aspects are currently at the investigation stage. This work overlaps with our recent efforts aimed at revising and refining the PicOS compiler, which allows us to think of new syntactic tools to be added to the language to facilitate those extensions.

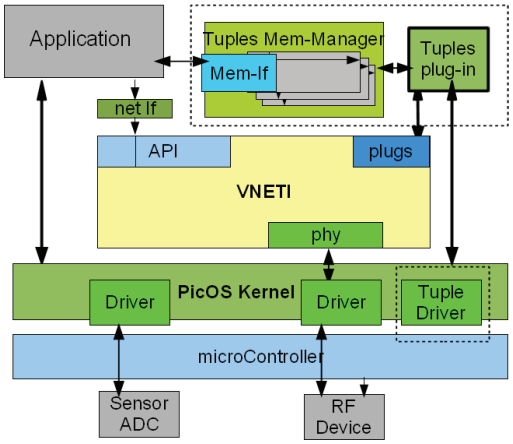


Figure 8: PicOS system architecture layout.

8. CONCLUSIONS

We have presented a tuple based extension to the PicOS threading model aimed at mitigating some of the problems caused by the frugality of the memory-constrained programming environment characteristic of a low-cost WSN node. Our extensions follow up on the elegance, simplicity, and power of Linda's tuple space concept. In contrast to Linda, we cannot get away with a single, global, and reliable tuple space; thus, our goal was not to emulate Linda in the WSN environment, but rather to study the possible ways of adapting the idea of tuples to the new (and capricious) type of distributed systems represented by WSNs. Our preliminary work demonstrates that the idea of tuples combined with thread contexts can significantly simplify programming without worsening the system's characteristic regarding its memory demands. The programs turn out to be considerably more legible and much easier to modify/extend than with the traditional approach to data structures and communication.

Our future work will include a quantitative evaluation of additional protocols and applications to provide assessment of design parameters such as code complexity, code modularity, operational efficiency. We will also focus on additional language extensions and constructs to provide a viable WSN development tool. An important issue is to find a good mechanism for extending the local and neighborhood spaces onto a global network-wide space. Inspired by TARP's approach to routing [4], we envision a fuzzy meta-routing scheme with rule driven rebroadcasts, which will collectively push the tuples towards the regions (neighborhoods) where they are needed. This calls for programming concepts and constructs that will allow the program to define special rules according to suggested distributed configurations/roles thereby mitigating the inherent unreliability of individual nodes.

9. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of USENIX'02*, pp. 289–302.

[2] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[3] P. Costa, L. Mottola, Amy L. Murphy, and G. P. Picco. Teenylime: transiently shared tuple space middleware for wireless sensor networks. In *Proceedings of MidSens'06*, pp. 43–58.

[4] P. Gburzyński, B. Kaminska, and W. Olesinski. A tiny and efficient wireless ad-hoc protocol for low-cost sensor networks. In *Proceedings of DATE'07*, pp. 1562–1567.

[5] P. Gburzynski and W. Olesinski. On a practical approach to low-cost ad hoc wireless networking. *Journal of Telecommunications and Information Technology*, 2008(1):29–42, January 2008.

[6] M. Goldman and S. Katz. MAVEN: Modular aspects verification. In *Proceedings of TACAS 2007*, pp. 308–322, Springer, LNCS, volume 4424.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[8] K. Hoffman and P. Eugster. Cooperative aspect-oriented programming. *Science of Computer Programming*, 74(5-6):333–354, 2009.

[9] O. Kasten and K. Römer. Beyond event handlers: programming wireless sensors with attributed state machines. In *Proceedings of IPSN'05*, pp. 7.

[10] S. Katz. Aspect categories and classes of temporal properties. In *Transactions on Aspect-Oriented Software Development I*, pages 106–134. Springer, LNCS, volume 3880, 2006.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. pp. 327–353, In *Proceedings of ECOOP'01* Springer, LNCS, volume 2072.

[12] P. Levis. *TinyOS Programming*. Cambridge University Press, 2009.

[13] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of SenSys'06*, pp. 167–180.

[14] K. Terfloth, G. Wittenburg, and J. H. Schiller. FACTS - a rule-based middleware architecture for wireless sensor networks. In *Proceedings of COMSWARE'06*

[15] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of MobiSys'04*, pp. 99–110.

[16] A. L. Murphy, G. P. Picco and G. C. Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol, 2006.

[17] P. J. McCann and G. C. Roman. Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts. In *Proceedings of COORDINATION '97*. pp. 338–354.

[18] R. De Nicola and G. L. Ferrari and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. IEEE Trans. Softw. Eng. pp. 315–330. IEEE Press, volume 24, 1998.

Simulink Analysis of Component-Based Embedded Applications

Feng Zhou Søren Top Krzysztof Sierszecki Christo Angelov
Mads Clausen Institute for Product Innovation,
University of Southern Denmark
Alsion 2, 6400 Soenderborg, Denmark
+45 6550 1684
{zhou, top, ksi, angelov}@mci.sdu.dk

ABSTRACT
The widespread use of embedded systems requires the creation of industrial software technology, which will make it possible to engineer systems that are correct by construction. That can be achieved through the use of validated (trusted) components, verification of design models and automatic configuration of applications from validated design models. These guidelines have been instrumental for developing COMDES - a component-based framework for real-time embedded control systems. In this framework, an application is conceived as a network of distributed embedded actors that communicate with one another by means of labeled messages (signals), whereby I/O signals are exchanged with the controlled plant at precisely specified time instants, resulting in the elimination of I/O jitter. The paper presents an analysis method that can be used to validate COMDES design models using the Simulink environment. It is based on a semantics-preserving transformation of a COMDES design model into a Simulink analysis model, which preserves both the functional and timing behaviour of the original design model. The discussion is illustrated with an industrial case study – a Medical Ventilator Control System, which has been used to validate the developed design and analysis methods.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering; D.2.11 [**Software Architectures**]: Domain-specific architectures

General Terms
Design, Experimentation

Keywords
Embedded control systems, component-based design, domain-specific frameworks, model-based analysis, semantics-preserving model transformation

MOMPES '10, September 20, 2010, Antwerp, Belgium

1. INTRODUCTION
The widespread use of embedded systems poses a serious challenge for software developers who have to address a number of stringent and contradictory requirements: reduced development costs and time to market, error-free operation and predictable behaviour under hard real-time constraints, open architecture featuring reusable components and software reconfiguration, etc.

The conventional development process cannot easily cope with these problems, since it is largely based on informal design methods and manual coding techniques. This has a negative impact on both the economy of production and the safety of embedded systems. In particular, software safety is severely affected by design errors that are typical for informal design methods, as well as implementation errors that are introduced during the process of manual coding.

Therefore, it is necessary to develop new design methods that will make it possible to engineer systems that are *correct by construction*. This is an ambitious goal that can be eventually accomplished by combining two complementary methodologies, i.e. model-driven and component-based design of embedded software [1]. The main elements of this approach include:

- Repositories of prefabricated and validated (trusted) components that can be used to build applications in a particular application domain
- Computer-aided design of applications using formal design models that are appropriate for the application domain
- Verification of design models with respect to functional and non-functional requirements via semantics-preserving transformation of design models into appropriate analysis models
- Automatic configuration of applications from validated design models using prefabricated software components

It can be expected that the adoption of the outlined software development process will result in the creation of industrial software technology for embedded applications, similar to those already available in mature areas of engineering such as electronic design, mechanical engineering, etc.

These guidelines have been instrumental in developing the COMDES framework and specifically, its third version [4]. This is a software framework for time-critical distributed control applications, featuring a hierarchical component model and signal-based communication between components at all levels of specification. With that framework, an application is composed

from actors, which are configured from prefabricated function blocks. This is an intuitive and simple model that is easy to use and understand by application experts, i.e. control engineers.

The validation of design models is an important aspect of the overall development process. There are a number of analysis methods and tools developed over the years, which are now widely used by the engineering community. Therefore, our approach has been to use such tools rather than invent new ones. However, that is possible if the analysis models used as input to those tools are consistent with the design models. This requires a semantics-preserving transformation of design models into analysis models used by the tools under consideration. Presently, our research is aimed at developing transformations that will make it possible to use two such tools, i.e. SIMULINK and UPPAAL.

This paper presents the transformation of COMDES design models into consistent Simulink models. Our approach is somewhat similar to Giotto modeling in SIMULINK [3], but is much more complex due to the hierarchical nature of the COMDES design model and the use of prefabricated components instead of generative programming techniques.

The discussion is illustrated with a case study – a Medical Ventilator Control System [8], which has been implemented in COMDES and validated in SIMULINK. The rest of the paper is structured as follows: Section 2 presents an overview of the COMDES framework, focusing on the main features of the design models used and their implications for the software development process. Section 3 presents the transformation of COMDES design models into SIMULINK models that preserve the functional and timing behaviour of the original models. Section 4 presents a discussion of automatic model transformation based on the corresponding meta-models. Section 5 concludes the paper by presenting a summary of the proposed analysis method.

2. COMDES FRAMEWORK: AN OVERVIEW

2.1. COMDES design models

COMDES is a domain-specific framework, which combines open system architecture with a model of computation that guarantees highly predictable behaviour, in the context of hard real-time distributed control systems [4], [5]. Its main features are summarized below:

A complex control system is decomposed into functional subsystems. A subsystem consists of one or more actors, i.e. active objects that are considered to be units of functionality as well as units of concurrency, such as sensor, controller, actuator, etc. A distributed embedded application is modeled as an actor network. Actors interact by exchanging labeled messages (signals), such as pressure, temperature, etc. Signal-based communication is transparent, i.e. independent of actor allocation.

Actors are executed in accordance with a clocked synchronous model of computation known as Distributed Timed Multitasking, which can be used to engineer highly predictable real-time systems that are exempt from input and output jitter and provide for a constant delay from sampling to actuation [5].

The COMDES application model is illustrated with Fig. 1, which shows the actor network specifying the Medical Ventilator control system implemented in the case study [8]. It consists of five communicating actors that can be grouped into two subsystems – *Ventilation Control* and *MMI Communication.* The first subsystem consists of actors *Sensor*, *Controller* and *PhaseSwitch*. The Sensor actor reads signals from A/D converters and calculates the values of process variables such as pressure and airflow, which are used as input data by the Controller actor. The latter implements a modal control system with several modes of operation, whereby the transitions between various modes are triggered by signals generated by the PhaseSwitch actor. The second subsystem consists of the *Transmitter* and *Receiver* actors, which are used to maintain communication with a Man-Machine Interface (MMI) unit over a serial link. System actors communicate with one another by means of labeled messages whose identifiers are shown on top of the corresponding communication links (see Fig.1).

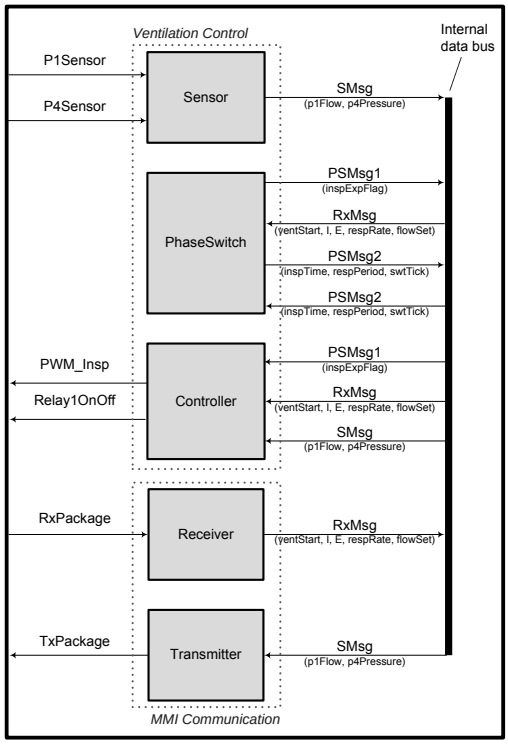


Figure 1. Medical Ventilator Control System: actor model

An actor is modeled as an integrated circuit consisting of a signal processing unit (SPU) and I/O latches, which are composed of input and output signal drivers, respectively, e.g. Fig. 2. The input latch is used to receive incoming signals and decompose them into local variables that are processed by the SPU. The output latch is used to compose outgoing signals from local variables produced by the SPU and broadcast them to potential receivers. Physical I/O signals are treated in the same manner but in this case, the latches invoke hardware-specific routines in order to exchange physical signals with the environment.

A control actor is mapped onto a real-time task having three parts: *task input*, *task body* and *task output*, implementing the input latch, SPU and output latch, respectively. One or more tasks may be allocated onto a particular processor, and their execution is managed by a real-time kernel, such that task inputs and outputs are activated at precisely specified time instants.

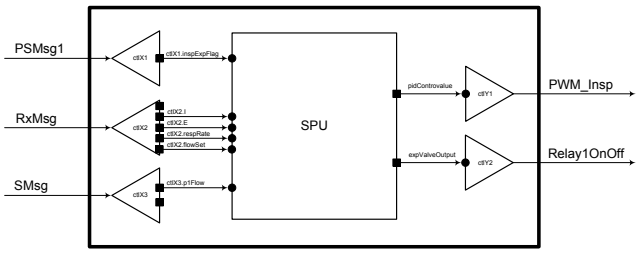


Figure 2. Controller actor: internal structure

These are relatively short pieces of code whose execution time is orders of magnitude smaller than the execution time of the actor task, which is typical for control applications. They are executed atomically in logically zero time, in separation from the task body (split-phase task execution). Specifically, task input is executed when the actor task is released, and task output – when its deadline arrives or immediately after the computation is finished if no deadline has been specified (see Fig. 3). Consequently, task I/O jitter is effectively eliminated as long as the task is schedulable and comes to an end before its deadline.

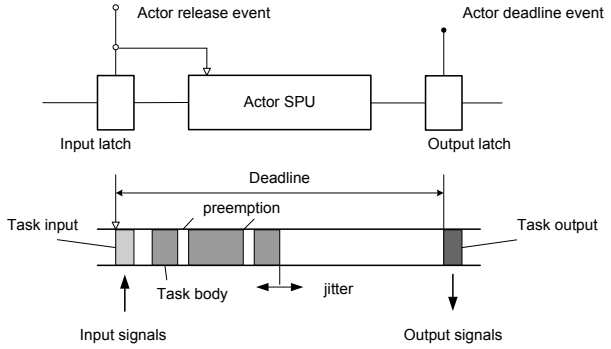


Figure 3. Actor execution under DTM

The COMDES model of computation, i.e. Distributed Timed Multitasking (DTM), extends the original model [2] to distributed systems, whereby I/O signals are generated at transaction release/deadline instants, thereby eliminating transaction I/O jitter [5].

An actor SPU is specified in terms of a data flow model, i.e. an *acyclic* function block network configured from instances of prefabricated components – function blocks (FBs). Function blocks are reusable and reconfigurable components that are stored in a repository in executable, binary format. These can be used to engineer heterogeneous embedded applications, such as sequential, continuous and hybrid (modal) control systems, or any combination thereof. The framework defines several kinds of function block - basic, composite, signal generator (SG) and state machine (SM) function blocks.

Basic and composite function blocks are components implementing different control and/or signal processing functions. The SM function block implements the reactive (control flow) aspect of actor behaviour by indicating the current control action to be executed, in response to a particular event. The SG implements the transformational (data flow) aspect of actor behaviour. It is a composite component featuring alternative sequences of function blocks used to execute different control actions, as indicated by a master SM.

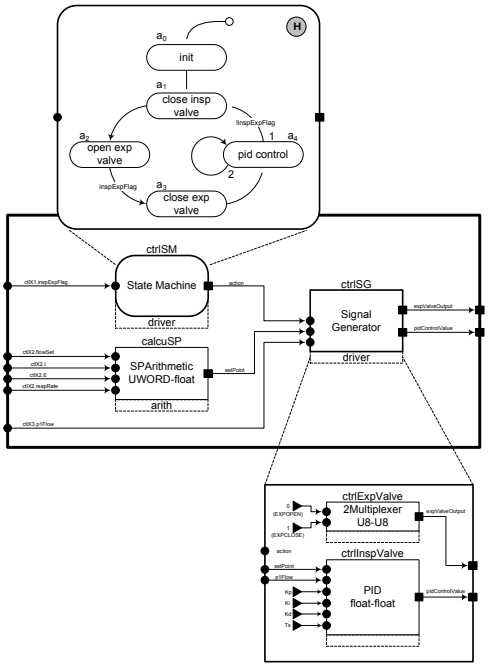


Figure 4. Controller actor: signal processing unit

The SM and SG function blocks can be composed together in order to implement actors with stateful behaviour, e.g. those used in sequential control systems as well as hybrid (modal) control systems, such as the Controller actor of the Medical Ventilator case study (see Fig. 4). Its signal processing unit contains an SM function block instance implementing the state transition graph shown in the figure, where each state (mode) is associated with a particular control action. Control actions are executed by the Signal Generator, which encapsulates instances of function blocks *PID* and *2Multiplexor*, whose functions are invoked in order to execute the indicated control actions.

Component design models have been used to develop design patterns for reusable and reconfigurable components implementing the above component models [6].

2.2. Implications for the software development process

The COMDES architecture emphasizes one of the main principles of systems engineering, i.e. *separation of concerns.* This makes it possible to separately treat different aspects of complex systems, such as system structure and behaviour, computation and communication, functional and timing behaviour, reactive and transformational behaviour, etc. [5].

Separation of concerns facilitates the design and analysis of embedded systems, which is reflected in the adopted software development process and the supporting software engineering environment. Consequently, different aspects of system behaviour can be verified in separation using appropriate techniques and tools, following a semantics-preserving transformation of system design models into the corresponding analysis models.

In particular, the behaviour of predominantly continuous systems can be analyzed through numerical simulation by exporting design

models into the SIMULINK environment, whereas the reactive behaviour of sequential systems can be investigated through property verification using model-checking tools such as UPPAAL. Likewise, separation of concerns makes it possible to analyze timing behaviour in separation from functional behaviour, by means of numerical response-time analysis techniques and tools.

The ultimate goal of our effort is to create a software development process that will eliminate both design and implementation errors, resulting in software that is *correct by construction*. That will be achieved through the use of validated (trusted) components, verification of design models and automatic configuration of applications from validated design models. On the other hand, timed multitasking makes it possible to engineer highly predictable systems operating in a flexible, dynamic scheduling environment.

On-going research is aimed at developing an integrated toolset supporting the entire software development process: specification, analysis, code generation and configuration of applications from prefabricated components [7].

3. SEMANTICS-PRESERVING COMDES-SIMULINK TRANSFORMATION

The main idea of our approach is to export the original design model of the system under investigation to the SIMULINK environment and analyse it via simulation, such that the original execution semantics is preserved during the simulation.

COMDES employs a hierarchical design model, whereby an application is modeled as a network of actors interconnected by communication channels. An actor is modeled as an acyclic network of function blocks interconnected by signal lines that visually represent the data flow in the model. Such a network may not contain loops. However, loops are allowed in the actor network, but they are effectively broken by the latching of input signals at the beginning of each period.

A SIMULINK model is very similar, being composed of blocks and signal lines, whereby constituent blocks can be standard SIMULINK blocks or System-Function (S-Function) blocks which are supplied by the user for specific purposes [9]. Hierarchy can also be modeled using SIMULINK subsystems.

There are however a number of issues that have to be addressed in the process of transformation, so as to take into account the functional and timing aspects of system behaviour and ultimately, develop an analysis model, which operates in exactly the same manner as the original design model. These are discussed in the following sections.

3.1 Transformation of functional behaviour

This transformation is facilitated by the similarity between COMDES design models and SIMULINK analysis models representing the controller part of the system, both of which are data flow models. Hence, it is possible to export a COMDES design model to the SIMULINK environment, by wrapping COMDES components into S-functions and wiring them together, following the interconnection pattern of the original design model. This kind of transformation can be characterized as *heterogeneous two-plane modeling in SIMULINK* (see Fig. 5).

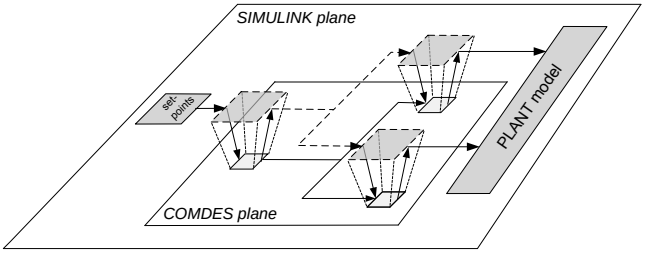


Figure 5. Heterogeneous modeling of embedded systems

With this modeling technique each function block of the original COMDES model is wrapped into an S-function. S-functions operate in the SIMULINK plane and are interconnected with each other using ports. When activated, S-functions invoke the encapsulated function blocks, which operate in the COMDES plane. Accordingly, communication between function block takes place in accordance with the original softwiring technique, whereby a function block uses pointers to access the output buffers of other function blocks in order to fetch the necessary input data.

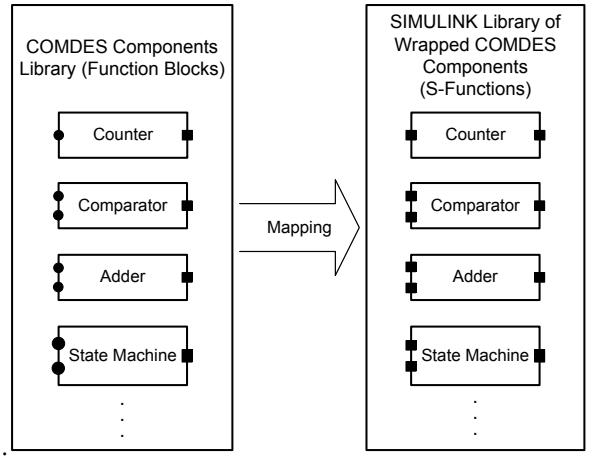


Figure 6. SIMULINK library of wrapped components derived from the COMDES component library

In order to implement this modeling technique it is necessary to create a library of wrapped COMDES components in the form of S-functions. It can be easily seen that the wrapped components library is a one-to-one mapping from the original COMDES library (see Fig. 6).

It is also necessary to figure out a way of mapping the COMDES softwiring technique to the Simulink interconnection technique. Finally, it is necessary to find appropriate modeling techniques for complex components such as composite and signal generator function blocks.

3.1.1 Transformation of basic and state machine function blocks

Each S-function from the wrapped component library encapsulates the corresponding COMDES function block, e.g. Comparator, Counter, etc. The S-function block communicates with other components in the Simulink plane via input and output ports, whereas a FB uses input pointers to access memory locations containing input data. The original version of the

wrapping technique is shown in Fig. 7. It assumes that the encapsulated FB has access to S-function input ports via the corresponding input pointers (shown as dashed arrows), and the data stored in the FB's output buffers is copied to the output ports of the S-function.

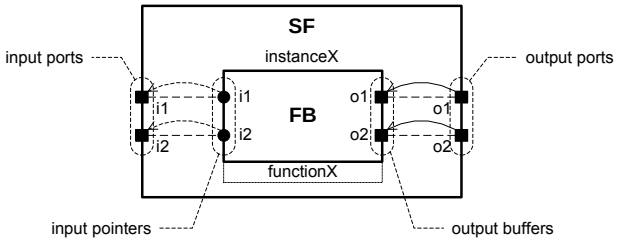


Figure 7. Encapsulating a function block into an S-function

The connection between two S-functions say SF_A and SF_B is shown in Figure 8. In this case SIMULINK takes charge of copying the data from output ports of SF_A to the input ports of SF_B.

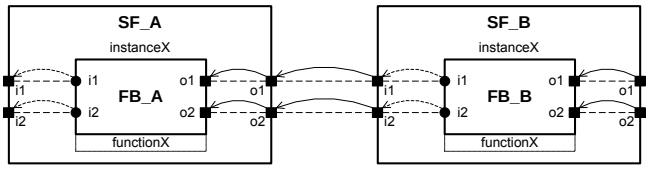


Figure 8. Connecting S-functions

Unfortunately, the above technique does not preserve the original function block interconnection semantics, i.e. the use of pointers to access the output buffers of producer function blocks from within the consumer function block.

This problem can be avoided by copying the address of the FB output buffer to the corresponding output port of the S-function, instead of the output data itself. So, the buffer address will be transferred from the SF_A output port to the connected SF_B input port and finally assigned to the corresponding FB input pointer. The latter can be used to directly access the output buffer of the function block encapsulated in SF_A (see Fig. 9).

In this way, wrapped function blocks communicate in exactly the same manner as in the original design model, whereby S-functions provide a shell with which the internal FB can be executed in the SIMULINK environment, and also – provide access points for monitoring signals in SIMULINK.

In COMDES, each function block is a type, which can have one or more instances, and each of them will execute a specific function (method) on the instance data (see e.g. function blocks FB_A and FB_B in Fig. 9). In SIMULINK, FB instances can be specified by means of the corresponding S-function mask. The mask is used to take the input parameters supplied by the user and pass them to the internal FBs. This configuration approach has been applied to all basic function blocks in the wrapped component repository.

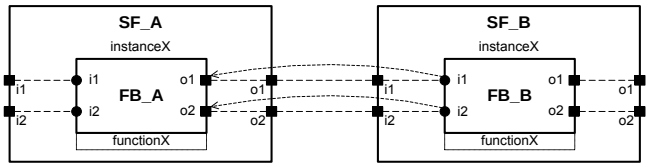


Figure 9. Connecting function blocks via the S-function shell

However, the above approach cannot be used when it comes to configuring an SM instance, because the number and type of inputs varies with different instances, and each of them requires a different configuration structure (State Machine Table), containing the state transition graph of the particular instance.

To solve this problem, a dynamic link library (DLL) is used in conjunction with the S-function encapsulating the SM instance. With this approach, the State Machine Table is compiled to a DLL independently, and it can be subsequently used by the state machine S-function to implement the desired state machine behaviour. In this case, the S-function contains only the standard method of the State Machine type - the so called *state machine driver*, which is used to process the state machine table of the SM instance. Thus, the SM component can be wrapped into an S-function and used in different applications.

Furthermore, the input ports configuration is also compiled into the DLL, so as to specify the number and type of inputs used by a specific SM instance, and their connections. The other configuration parameters (e.g. instance number, instance function, etc.) are left for the S-function mask, which is thus identical for all function block types.

3.1.2 Transformation of composite and signal generator function blocks

Composite function blocks and signal generators cannot be wrapped into S-functions in the same way as basic function blocks. The reason is that they are hierarchical models, whereas the S-function is a flat model, which rules out the nesting of S-functions.

However, that problem can be easily solved by means of SIMULINK subsystems encapsulating S-functions wrapping constituent COMDES components. A composite function block contains a single sequence of FB instances. Hence, it can be modeled by a single subsystem block encapsulating the corresponding sequence of S-functions.

A Signal Generator is composed of multiple FB sequences, which are selected for execution by a master state machine indicating the sequence to be executed during a particular invocation. In Simulink, each of these sequences is modeled by a component denoted as *Switch Case Action Subsystem*. These subsystems are triggered by a *Switch Case* block, as shown in Fig. 10, which depicts the SIMULINK model of a Signal Generator used by the Controller actor.

3.2 Modeling actor timing behaviour under Timed Multitasking

Timed multitasking is simulated by means of SIMULINK subsystems modeling the input and output latches of the actor (see Fig. 11).

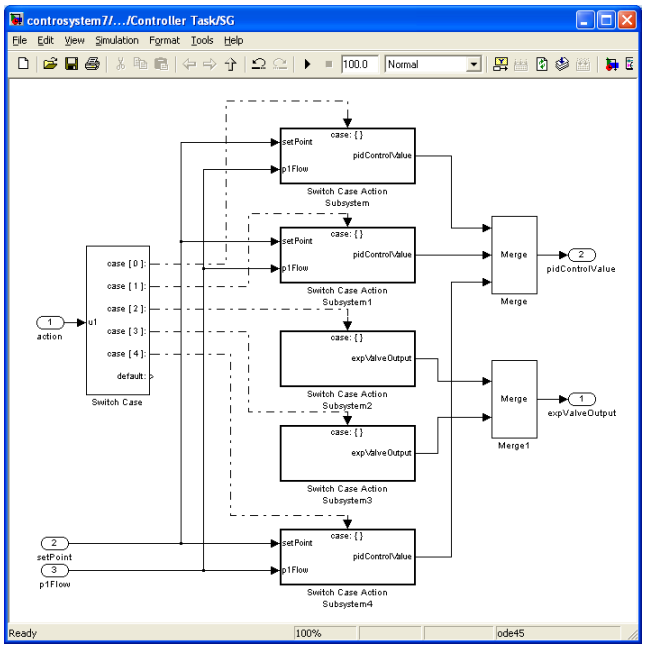


Figure 10. SIMULINK model of Signal Generator

Inside the Input Latch subsystem, the incoming messages are unpacked if they have more than one constituent variable. That is modeled by Demultiplexor (Demux) components, whose outputs are connected to Zero-Order Hold (ZOH) blocks. These are used to sample input signals and keep them unchanged during the execution period of the actor task.

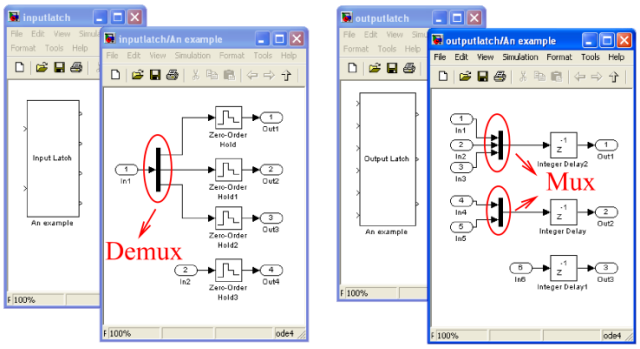


Figure 11. Internal structure of actor I/O latches

Inside the Output Latch subsystem, several output variables could be packed into one message. That is modeled by Multiplexor (Mux) components, whose outputs are connected to Integer Delay (ID) elements modeling the constant delay from sampling to actuation, as specified by the actor deadline.

The I/O latch subsystems are combined with a subsystem modeling the actor task in order to compose a subsystem modeling a COMDES actor, in accordance with the COMDES actor model (see Fig. 12). The actor task is composed from connected S-functions which are chosen from the wrapped COMDES component library. During simulation, the above subsystems have to be executed in accordance with the DTM model of computation: the actor task is released by the corresponding execution trigger, whereby the input latch is executed when the task is released and the output latch – when the task deadline arrives, as shown in Fig. 3.

To that end, they have to be appropriately parameterized. The ZOH blocks of the Input Latch keep the input signals of the actor task unchanged during the execution period, so the sample time for these blocks should be equal to the period expressed as an integer number of simulation time units. The ID blocks in the Output Latch delay the output signals for an interval of time equal to the actor deadline, specified by the corresponding number of simulation time units.

Period and deadline parameters are supplied via a mask associated with the actor subsystem. These settings will be passed to the internal I/O latch and actor task subsystems, and ultimately to their constituent components, i.e. S-functions and SIMULINK primitives. In this way, the SIMULINK model maintains the same timing behavior as the original COMDES design model.

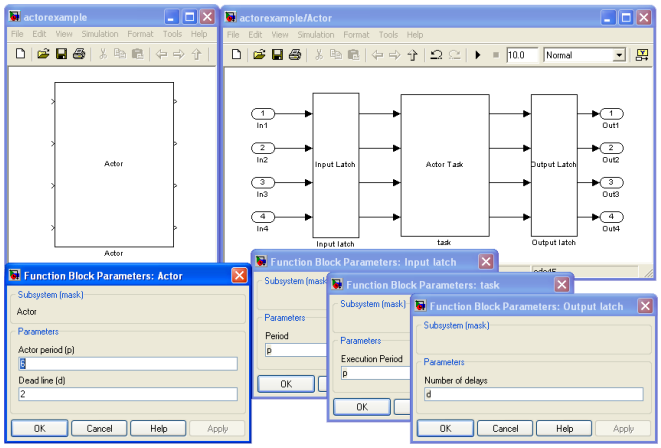


Figure 12. Actor configuration

3.3 Building a SIMULINK model with the wrapped COMDES components

The outlined modeling technique can be used to build the models of the actors constituting the entire SIMULINK model under investigation, e.g. the Controller model shown in Fig. 13. These can be used to compose the system model as shown in Fig. 14.

Here, the constants on the left-hand side are used to simulate the Receiver actor task's output variables, which are combined into one message (i.e. *RxMsg*) following the original design model (Fig. 1), and then this message is sent to both *PhaseSwitch* and *Controller* actors. Besides that message, the two actors also receive and produce other messages/signals in the same way as in the design model, i.e. *PSMsg1*, *PSMsg2*, *SMsg*, etc. System actor models are connected to the Medical Ventilator plant model, which is derived from the real plant (inspiration valve in the Medical Ventilator).

The plant is coupled to the Controller actor via the *pidControlValue* control signal and *SMsg.p1Flow* feedback signal thus forming a closed-loop control system. Finally, the experimental result can be shown in the scope, which is connected to two copies of the plant model operating with and without control respectively, for the purpose of comparison.

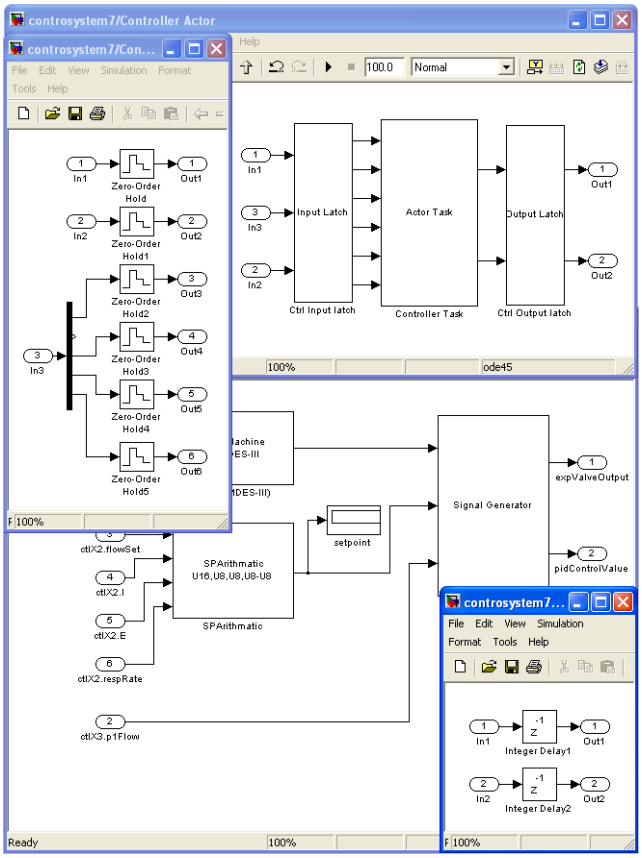


Figure 13. Simulink model of the Controller actor

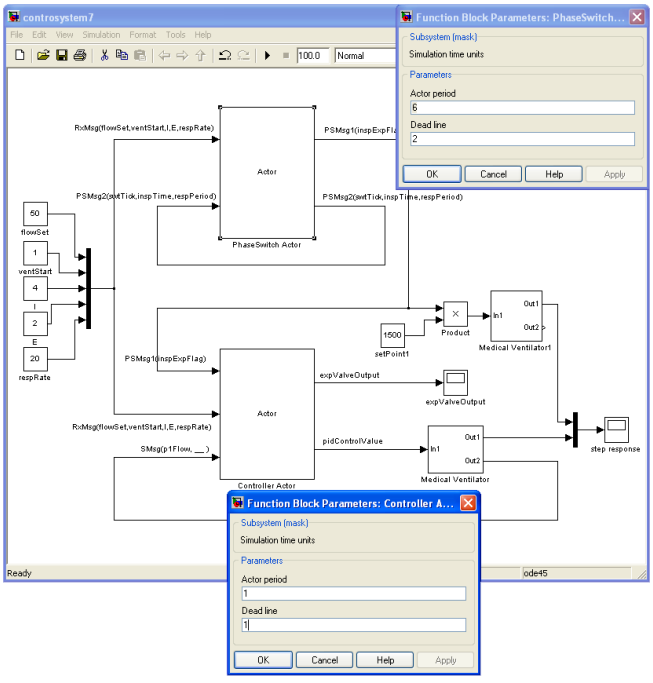


Figure 14. Simulink model of the Medical Ventilator control system

In this way, the control parameters can be tuned before they are used in the control software of the real machine, which saves both development time and cost.

Figure 15 shows the signals generated during the Simulink simulation run, i.e. the PhaseSwitch output signal – *inspExpFlag* (the upper diagram) and the system step response (the lower diagram). The former indicates the different respiration phases (inspiration or expiration) to the Controller actor, which reacts accordingly by generating appropriate control signals for the inspiration and expiration valves of the machine.

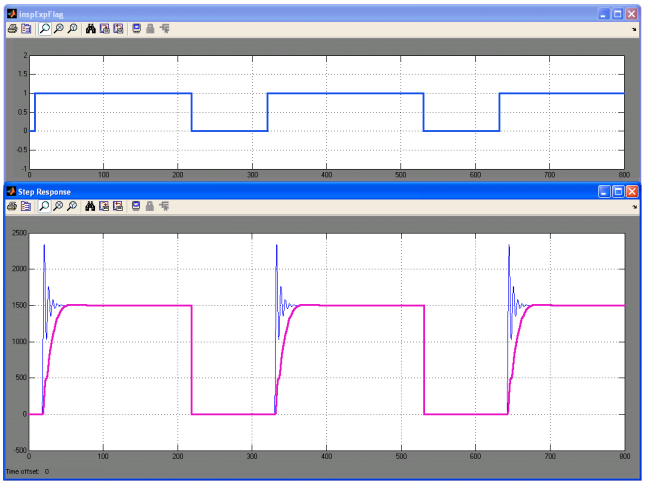


Figure 15. *InspExpFlag* signal and step response of the plant

The simulation result, i.e. the controlled step response (purple curve), is shown together with the uncontrolled step response (blue curve) in the second diagram. The comparison between the two step responses shows that both oscillation and overshoot, which are dangerous to the patient, have been effectively eliminated by the implemented control system.

4. AUTOMATIC COMDES-SIMULINK MODEL TRANSFORMATION

In the above case study, the SIMULINK model of the investigated system has been built using drag and drop components from the wrapped COMDES library, as well as primitives from the SIMULINK standard library. However, this approach is not so convenient when the system is really complex, i.e. it is time consuming and error-prone for the user to manually transform the design model into a consistent analysis model. This has motivated the investigation into automated transformation of COMDES design models into SIMULINK analysis models via an appropriate model-to-model (M2M) transformation process supported by the Eclipse development platform [11]. This process requires the development of COMDES and SIMULINK meta-models, both of which are based on the Ecore meta-meta-model defined in the Eclipse Modeling Framework (EMF). The COMDES meta-model has been developed in a previous project [7] and is now undergoing a process of refinement, whereas the SIMULINK meta-model has been developed by the Software Research Group in the University of Salzburg, Austria.

In EMF, there are several tools support the M2M transformation, such as ATL (ATLAS Transformation Language), QVT (Query/View/Transformation), SmartQVT, etc., and in this

project, ATL has been adopted. With this transformation language, it is possible to specify how a target model can be produced from a given source model. Specifically, ATL introduces a set of concepts such as ATL module, helpers, rules, etc., which make it possible to describe model transformations.

The transformation process can be summarized as shown in Fig. 16. The COMDES source model, conforming to the meta-model MMCOMDES, is transformed into a target SIMULINK model that conforms to the meta-model MMSIMULINK. The transformation is defined by the transformation model COMDES2SIMULINK, which itself conforms to the ATL model transformation meta-model. Finally, all three meta-models conform to the Ecore meta-meta-model.

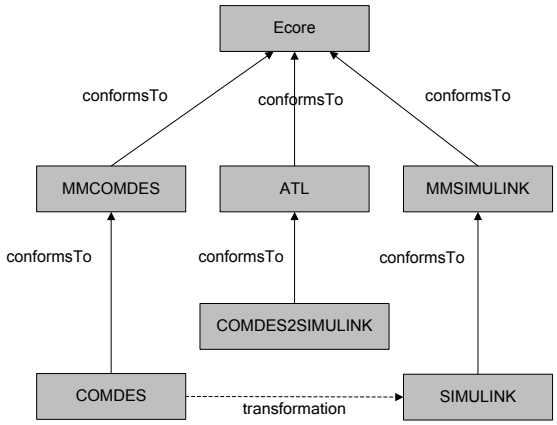


Figure 16. M2M transformation process

The envisioned model transformation process will be implemented in a special subsystem integrating SIMULINK with the COMDES development toolset [7].

5. CONCLUSION

The paper presents an analysis technique for component-based embedded applications in the context of the COMDES framework, which is based on a semantics-preserving transformation of COMDES design models into SIMULINK analysis models.

A two-plane modeling approach to the analysis of embedded applications has been developed using wrapped COMDES components (S-functions). In the upper plane, the S-functions are chained and simulated in Matlab SIMULINK. The dataflow between the components is maintained the same as in the COMDES model, and the data themselves are actually processed by the encapsulated lower-plane COMDES components. This approach has been further extended to composite components and system actors.

The presented solution preserves the COMDES semantics in terms of functional behaviour, which remains unchanged in the SIMULINK model. The timing behaviour of COMDES applications is also preserved via specific solutions making it possible to sample input signals and produce output signals at precisely specified time instants, in accordance with the timed multitasking semantics of COMDES design models.

Finally, the formal transformation from a COMDES design model to a SIMULINK model has been investigated via the ATL language in the Eclipse Modeling Framework, in order to automatically generate the target SIMULINK model.

The developed methodology is not strictly limited to COMDES applications. In a broader context, it offers an analysis method specifically tailored for embedded applications built from pre-fabricated executable components that are different from SIMULINK blocks. In this case, it is not possible to follow the conventional development process, whereby a control system is initially designed and simulated in SIMULINK, followed by code generation from the validated system model. This problem is addressed by the presented methodology, which makes it possible to configure an application from prefabricated components, and then validate the design by exporting it to SIMULINK, such that the execution semantics of the original design model is preserved.

6. REFERENCES

[1] B. Bouyssounouse and J. Sifakis (Eds.), “Embedded Systems Design. The ARTIST Roadmap for Research and Development”, *LNCS 3436* (2005)

[2] J. Liu and E.A. Lee, "Timed Multitasking for Real-Time Embedded Software", *IEEE Control Systems Magazine*, special issue on Advances in Software Enabled Control, February 2003, pp. 65-75

[3] W. Pree, G. Stieglbauer and J. Templ, “Simulink Integration of Giotto/TDL”, Proc. of ASWSD 2004, LNCS 4147 (2006), pp. 137-154

[4] C. Angelov, K. Sierszecki and F. Zhou, “A Software Framework for Hard Real-Time Distributed Embedded Systems”, Proc. of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2008, Parma, Italy, Sept. 2008, pp. 385-392

[5] C. Angelov, K. Sierszecki and Y. Guo, “Formal Design Models for Distributed Embedded Control Systems”, Proc. of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems ACES-MB 2009, Denver, Colorado, USA, Oct. 2009, pp. 43-57

[6] K. Sierszecki, F. Zhou and C. Angelov: Reusable State Machine Components for Embedded Control Systems. Proc. of the 7th International Conference on Informatics in Control, Automation and Robotics, ICINCO 2010, Funchal, Madeira - Portugal, June 2010

[7] Y. Guo, K. Sierszecki and C. Angelov, “COMDES Development Toolset”. Proc. of the 5th International Workshop on Formal Aspects of Component Software FACS 2008, Malaga, Spain, Sept. 2008, pp. 233-238

[8] F. Zhou, W. Guan, K. Sierszecki and C. Angelov, “Component-Based Design of Software for Embedded Control Systems: The Medical Ventilator Case Study”, Proc. of the Int. Conference on Embedded Software and Systems ICESS 2009, Hangzhou, China, May 2009, pp. 157-163

[9] The MathWorks, Inc., “Writing S-Functions”, version 6, http://www.mathworks.com.

[10] The MathWorks, Inc., “Using Simulink”, version 6, http://www.mathworks.com.

[11] Eclipse Modeling Project, http://www.eclipse.org/m2m