Daniel Balasubramanian, Christophe Jacquet,
Sahar Kokaly, Tamás Mészáros and Pieter Van Gorp, editors

Proceedings

8th International Workshop on Multi-Paradigm Modeling MPM 2014

co-located with Models 2014

Valencia, Spain, 30 September 2014

To contact the editors:

Daniel Balasubramanian
Institute for Software Integrated Systems
Vanderbilt University
1025 16th Ave. S, Suite 102
Nashville, TN 37212, USA
daniel@isis.vanderbilt.edu

Christophe Jacquet
Department of Computer Science
Supélec Systems Sciences (E3S)
3 rue Joliot-Curie
91192 Gif-Sur-Yvette cedex, France
Christophe.Jacquet@supelec.fr

Sahar Kokaly
Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario, L8S 4L8, Canada
kokalys@mcmaster.ca

Tamás Mészáros
Department of Automation and Applied Informatics
Budapest University of Technology and Economy (BUTE)
Budapest 1117, Magyar tudósok krt. 2. Hungary
Meszaros.Tamas@aut.bme.hu

Pieter Van Gorp
School of Industrial Engineering
Eindhoven University of Technology (TU/e)
De Lismortel 2, 5612AR Eindhoven, The Netherlands
p.m.e.v.gorp@tue.nl

Contents

Preface

The MPM workshop series brings together researchers and practitioners interested in using explicit and heterogeneous models throughout the design of a system. The 8th edition will take place on 30 September 2014 and will be co-located with MODELS'14 in Valencia, Spain.

Out of the 9 papers submitted and reviewed by at least four members of the program committee, 8 were selected for oral presentation. The average reviewer score for the accepted papers was $+0.48$ on a scale of -3 to $+3$.

In addition to the presentation of the selected papers from the technical program, MPM'14 will feature an invited presentation by Akos Ledeczi on WebGME.

This volume contains versions of the selected papers that the authors had the opportunity to enhance by taking into account reviewers' comments.

September 8, 2014

Daniel Balasubramanian
Christophe Jacquet
Sahar Kokaly
Tamás Mészáros
Pieter Van Gorp

Steering Committee

Program Committee

Integrating System Modeling and Cost Models Using Meta-Modeling Techniques

Viktor Steiner
Evopro Innovation Ltd.
Budapest, Hungary
steiner.viktor@evopro.hu

Gergely Mezei
BUTE DAAI
Budapest, Hungary
gmezei@aut.bme.hu

Abstract. The precise estimation of time and resource consumption plays a pivotal role in planning software development projects at their earliest development phase. Since cost parameters are mostly determined by the architecture, a possible approach is to design a platform independent architectural model of the prospective software and estimate the cost based on it.

In this paper, we introduce a method, which produces a cost estimate by processing the architectural model of the software being designed. The provided method analyzes the architectural models, and utilizes a modified version of Function Point Analysis to determine the probable cost based on the analysis. The paper also presents a preliminary verification process to evaluate the accuracy of the cost estimation method. The main achievement of the introduced method is that it estimates cost in platform independent units, which can be refined to give accurate cost estimation for different platform implementations.

Keywords: Meta-modeling, Cost Estimation, Function Point Analysis

1 Introduction

Cost estimation is a major challenge in software industry. It is hard to find features that can precisely predict the expected cost of the complete development process. Since architecture has the greatest impact on development costs, architectural models can provide a basis for the estimation. In this paper, we provide a method, which analyzes the architectural models created in the design phase and estimates the expected cost of the software to build.

Our solution is unique among cost modeling methods, since it applies Multi-Paradigm Modeling techniques to achieve its goal. Compared to other existing cost estimation techniques, our approach does not require creating a separate cost model manually. Instead, we analyze architectural models and generate the cost model from them. Our method maps model elements of the software architecture domain to the concepts of the cost modeling domain. Since we use a platform independent architectural and cost modeling domain, our results can be applied early in the development process, before deciding which technology to use for the implementation. This also means that our method is also useful for facilitating the decision between possible implementation technologies, because the cost estimation result can be refined into estimations on different platforms and compare the cost predictions.

The paper is organized as follows: In Section 2, we give a short summary of the state of the art in the field of cost estimation approaches. Section 3 introduces the Visual Modeling and Transformation System [1], which we used to implement the cost estimation method. In Sections 4 and 5, our cost estimation method is presented. The method is a modified version of Function Point Analysis [2], adapted to SysML [3] models. Section 6 introduces a verification process, which is used to test the accuracy of the results provided by the cost estimation method. Finally, Section 7 concludes the outcome of our work and gives main directions of future work.

2 Related Work

Existing cost estimation methods typically do not use existing resources such as requirements, specification, or architectural models for calculating the probable cost, they use their own cost model, which must be prepared separately. This is problematic for various reasons, e.g. (i) It takes extra time and effort to estimate probable development cost. (ii) Cost estimation becomes a mostly manual task, since the cost model does not rely explicitly on existing resources. Manual steps increase the probability of errors in the estimation. (iii) A cost modeling expert is always needed, who prepares and analyzes the cost model. These issues arise in most of the existing cost estimation methods, for example, in COCOMO II [4].

However, there are some methods, which use resources from the development process. An estimation technique that measures development effort based on use cases [5] and another that uses requirements as a basis [5] can be mentioned here. Although these methods use available development resources, they still need too many manual steps to produce the estimation. This is because both requirements and use cases are high level concepts, which can hardly be formalized, which could enable programmatic analysis. On the other hand, use cases and requirements are available very early in the development process, therefore the estimation can be performed earlier compared to our method. However, performing the analysis on architectural models can be much more accurate, since more formalized data is available. Since these two approaches can both be performed during the development process, they can complement each other, by giving a vague initial estimation, and then later calculate a refined, more accurate estimation.

Although we had not found methods that estimate development cost from architectural models, there are some methods that are based on similar concepts, of which [6] is the most closely related. The method uses architectural models to predict performance, and to facilitate architectural design decisions. The latter is among our goals as well, as our method can be used to compare and evaluate different architectural versions based on their estimated costs.

3 The modeling environment

A cost estimating algorithm requires a modeling environment, in which architectural system models can be created, models can be processed and analyzed programmatically. We had chosen SysML [3] to model the architecture. The main reason behind this decision was that SysML is a widely used and accepted general purpose systems modeling language, and we used it in several projects previously in Evopro Innovation Ltd. However, our method is only partially specific to SysML, it can be applied with other architecture modeling languages as well.
The selected modeling tool, in which the SysML language environment was created, is the Visual Modeling and Transformation System [1]. In VMTS, any modeling language can be defined by creating its metamodel. The framework offers a highly customizable workbench to edit the models visually and the models can be processed programmatically using the VMTS Domain Specific Language API.
Our SysML dialect was defined by a meta-model based on the OMG SysML and the related parts of the UML specification. Creating the whole meta-model of the UML and SysML languages was not our goal, we intended to calculate our cost estimation in the architectural design, thus, we focused on those parts of the UML/SysML meta-models that describe the architecture. As described later, in sections 4 and 5, we identified the following aspects of SysML as required: (i) the Block Definition Diagram, (ii) the Internal Block Diagram, (iii) the Requirement Diagram, (iv) the Use Case Diagram and (v) the Sequence Diagram. As the first step of our work, we created these languages and customized their visual appearance and behavior according to the SysML standard.

4 Cost estimation

In the past decades, different methods were developed for software cost estimation. Our goal was to find the best suitable method among these for our purposes. The selected method had to be: (i) current, (ii) used in software industry (to ensure that it predicts the development cost correctly) and (iii) publicly documented to avoid copyright issues. Moreover, we have decided to focus on solutions capable of estimating the size of systems created with object oriented principles. Finally, we have chosen the method described in [7]. [7] describes a collection of methods, each usable in different phases of software development projects. We only needed the ones that deal with calculating the size of the software, since the size has the greatest impact on development costs, and it can be measured in the architectural design phase. In our method, the cost is estimated based on the software size, which is typically measured in two ways: (i) Source Line of Code (SLOC) and (ii) Function Points. In case of SLOC, the number of source lines required to implement the software in a particular language is measured. In contrast, function point measuring methods quantify the functionality of software in an abstract, platform independent unit. We selected the later one, since it is platform and technology independent. Moreover, function points, despite they are abstract measurement units, can be converted to an estimated number of source lines, based on past development experiences.

4.1 Function Point Analysis

Function point measuring methods are collectively referred to as Function Point Analysis (FPA) methods. FPA has no official standard, several different implementations exist. In our solution, we used the approach described in [7], and a more detailed version of the same method in [2]. As we mentioned before, FPA measures the size of software based on its functionality. In FPA's interpretation, functions of a software are always transactions, which are executed on some kind of data set. Therefore, function point count is determined by logically related data sets, and by the transactions associated to them. The basic terms of FPA can be seen on Fig. 1.

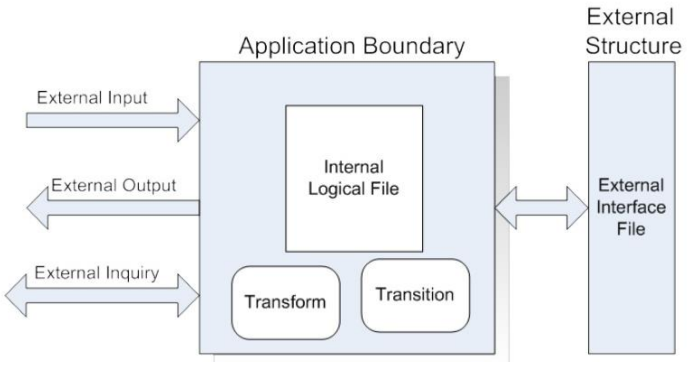


Fig. 1. Overview of the basic terms of Function Point Analysis

We modified the original Function Point Analysis to adapt it to SysML architectural models. Firstly, we examined, which of the basic FPA terms are necessary in order to implement the method properly:

- ***Application Boundary:*** It specifies the communication interface between the system and the outside world.
- ***Internal Logical File (ILF):*** Logically related set of data, maintained by the application.
- ***Transaction:*** An elementary process, which obtains data through the application boundary. There are three different kinds of transactions we distinguish:
  - ***External Input (EI):*** A transaction obtaining data from the environment
  - ***External Output (EO):*** A transaction submitting data to the outside environment.
  - ***External Inquiry (EQ):*** A transaction, which gets data from inside of the application, through the application boundary. The data is queried according to query parameters. The requested data cannot be derived (calculated) data.

We discovered that the above terms are necessary to implement the method, and can be matched to SysML concepts, as described in section 4. However, the remaining terms from Fig. 1. are not needed for the implementation. External Interface File (EIF) is an ILF, maintained by another application. It is not part of our model, because it is not an Object Oriented Programming concept and our focus was on estimating the cost based on OOP software models. Transformation & Transition: A transformation is a sequence of mathematical calculations transforming the input data into the required form. A transition is an event, which changes the state of the application. These concepts can clarify the results of the estimation, but they are not elaborated in the architectural design phase.

5 Mapping Function Point Analysis to SysML

In this section, we present how we managed to map the basic FPA terms into SysML concepts, and calculate function points by analyzing SysML models.

5.1 Application boundary

The first step of FPA is to define the application boundary. Here we have to analyze the data used by the application, and determine whether it is maintained by the application. If it is, then the data resides inside the application boundary, otherwise it belongs to the outside environment. When we design the architecture of software in SysML, the first step is the definition of the application boundary, however, it is not displayed explicitly as a model item: The first step of software modeling is usually the definition of functionality, in the form of Use Case diagrams. In Use Case diagrams, actors belong to the environment, and the highest level use cases, which they are associated with, are matched to FPA transactions that obtain data through the application boundary.

5.2 Data types

The second step of FPA is to identify and rate the data sets maintained by the application. These data sets always appear as an ILF. An ILF is a user identifiable group of logically related data that resides entirely within the application boundary, and is maintained by External Inputs. An ILF has an inherent meaning, it is internally maintained, it has some logical structure and it is stored in a file, as defined in section 9 of [2]. According to this definition, ILFs are almost identical to persistent entities of an OOP application, whose structure and connections can be modeled on an Entity Relationship diagram, or in our case, on a SysML Block Definition diagram. However, this data model must be programmatically distinguishable from the other system elements that are also modeled on Block Definition diagrams. This can be achieved by performing a small modification on the original SysML meta-model and adding an attribute – a flag – to the Package meta element. FPA analyzer can decide whether to search for the data model elements in those Packages, or not.

After identifying ILFs, the next step is to evaluate their complexity and rate them. Complexity analysis is based on two concepts: (i) A Record Element Type (RET) is a user recognizable sub group of data elements within an ILF. (ii) A Data Element Type (DET) is a unique, user recognizable, non-recursive (non-repetitive) and dynamic field in a RET. Additionally, a DET can invoke transactions or can act as additional information regarding transactions. During the evaluation process, Record Element Types and Data Element Types within an ILF are counted.

By definition, an ILF itself is also a user recognizable group of data elements, therefore it always consists of at least one RET. ILFs consist of more than one RET, if they contain multiple logically related sub groups of data, which have no meaning on their own and can only be interpreted inside the ILF. The RET concept can be illustrated using the following two cases:

- There are two logically related sets of data (A and B), which have a common subset, a key field, by which they are connected to each other. Both A and B can be interpreted on their own, thus they are both considered a separate ILF.
- There are two sets of data (A and B), where B is a subset of A. In this case, B cannot be interpreted on its own. Therefore, it cannot be an ILF, it can only be considered a RET inside A. An example for this case is a music CD that contains songs. Both the CD and the songs have attributes, but the songs cannot be interpreted on their own, without the data contained by the CD.

When interpreting the RET concept on SysML models, we assumed that the data model of the designed application is available in the form of SysML Block Definition diagrams. The persistent entities of the data model can be interpreted as ILFs or RETs, as it was mentioned above. According to the definition, a data set can only be considered a RET if it is a real subset of an ILF. This means that an entity in the data model can only be considered a RET if it has only one parent, and its children are RETs. If the above condition is satisfied, an entity is considered a RET, if not, it is considered an ILF. Note the parent-child relation here means the usual one-to-many relation used in Entity-Relationship diagrams.

The Data Element Type concept can be easily mapped to SysML models. According to the definition, a DET is very similar to a data field of a persistent entity. Since we have already identified ILFs and RETs, the only remaining task is to count the data fields for each identified RET, and the evaluation of the data model is complete. The actual weight of an ILF can be read out from the corresponding cell of the table defined in Section 9 of [2].

5.3 Transactions

The third step of Function Point Analysis consists of the identification and evaluation of transactions. Our method is based on Use Case diagrams of the system at this step. As it was pointed out previously, top level use cases – which are directly connected to actors – represent transactions, thus their automatic identification is easy. On the other hand, programmatic determination of the transaction kind (External Input, External Output or External Inquiry) is not possible. This is mainly because if we want to distinguish between the types, we need to

- **Determine the main direction of the data flow.** This would distinguish input transactions (EI) from output transactions (EO and EQ). Here we use the term “main direction” instead of simply using “direction” because according to FPA definition, all three transaction types can send data in both directions. For example, an input transaction can send back a status code, which indicates whether the transaction is successful or not. In SysML, however, we can only show the direction(s) in which data flows, there is no such concept as main direction.

- **Check whether the output data is derived or not.** This would distinguish between output transaction types (EO and EQ). Based on FPA definition, derived data is the result of some kind of calculation. In case of SysML models, the only indicator of data derivation is that data type changes during the execution of a transaction. This, however, is not an accurate conclusion, because it is possible that no calculation is performed, the data is just transformed into another form. For example, an array of items is transformed into a linked list of the same items. In this case, type of the data is changed, but the actual information remains the same.

Consequently, in case of SysML models, we cannot distinguish FPA transaction types from each other. Because of this, we decided to give up the idea of fully automated cost analysis, and add some additional information to the SysML meta-model, which helps identifying transaction types. As mentioned before transactions are mapped to top level Use Cases in SysML models. Therefore, we decided that the most optimal solution is to add an attribute to the Use Case meta-element, which marks the transaction type. After setting this attribute, the evaluation of transactions can be performed automatically. The process consists of two steps: counting of (i) data element types and (ii) referenced file types.

The first step is to calculate the data element types. Parameter and return values get into and out of the application through the application boundary. In case of OO applications, the boundary is usually an interface, whose operations start the execution of transactions. In SysML, this concept can be modeled as Use Case and Sequence Diagrams, where there is a Sequence Diagram associated to each use case shown on the Use Case Diagrams. The Sequence Diagrams show the order of operations that implement the particular use case. Here we only analyze the first operation of a sequence, which is the interaction point between the application and the environment. The parameter and return types of that operation are used to calculate the complexity of the transaction being analyzed, therefore, these are the types that have to be counted. Note that there are transactions that cannot be analyzed this way. For example, take a transaction, which queries the database for some data, and displays the results on the GUI. The operation that triggers the transaction does not give back a return value, it just updates some part of the GUI, but it is clear that data gets through the application boundary. We solved this problem by creating a new descendant of the Operation element in the SysML meta-model, called GUIOperation. On this kind of operation, the modeler can set the properties that it updates, thus, the complexity of the function can be fine-tuned.

The second step is to count the file type references that are (by definition) unique ILFs that a transaction references during its execution. To count them, the prerequisites are the same as they were in the previous step. Namely, each top level use case should have a corresponding Sequence Diagram, which shows the order of operations implementing the particular use case. When the necessary diagrams are ready, processing them to find referenced file types is an easy task. We only need to analyze the operations of a transaction, and count their parameter and return types. Each type is counted only once, because file type references are unique by definition.

By now we identified the transactions, and determined their types. We have counted the referenced file types and data element types, thus the evaluation process of transactions is complete. The actual weights of the transactions can be read out from the corresponding cells of the tables in Sections 5, 6 and 7 of [2].

5.4 Evaluation

The last step of FPA is to calculate the Function Point count, by using the following formula [2]:

$$FP = \sum_{i=1}^{m} ILF_i + \sum_{i=1}^{n} EI_i + \sum_{i=1}^{o} EO_i + \sum_{i=1}^{p} EQ_i, \tag{1}$$

where ILF_i, EI_i, EO_i and EQ_i are the weights of the files and transactions that were calculated according to the methods defined in earlier sections. The resulting value is a platform independent quantity that not only measures software functionality, but it can be converted to platform specific source code estimations as well. For the conversion, a table is used, which is maintained and updated by Quantitative Software Management Inc. [8] The table contains factors to convert Function Points into SLOC estimation on different programming languages. The conversion factors are based on historical data from completed software projects (currently 2192 different projects). The converted SLOC values provide a basis for comparing source code estimations on different platforms, and selecting the most suitable platform.

Note that the original FPA method uses a Value Adjustment Factor to fine tune the Function Point count, based on non-functional requirements. This adjustment can be done with our result as well, as described in Sections 11 and 13 of [2].

6 Verification

In order to check that our method produces correct results, a verification method was implemented. Our original plan was to apply the method from the beginning of a new project and validate its results after completing the project. We realized that this would require months to apply and we had difficulties in convincing the project management. We had strict time constraints in the current projects and the project management also wanted to have a preliminary validation of the results before introducing the proposed method in real development. We have decided to use a simpler but not as precise method: we generated SysML models from the source code of projects already completed. We used the source code to generate an architectural model from a complete software and run the cost estimation method on it, thereby verifying its accuracy. We were aware that the accuracy in this case depends on how precisely the generated model complies with the source code, and how much the generated models differ from the architectural models created in the design phase. We made assumptions (e.g. the architecture does not change in a large extent during the development) keeping in mind that the verification method is only preliminary and it is necessary to prove the correctness of our method, which can be fine-tuned later, once it is used in production scenarios.

As the subject project, we used an mCPS system developed by Evopro Innovation Ltd. [9] The system consists of the following components: (i) database, (ii) server, (iii) a thick administrator client and a (iv) mobile client. The first three components were using Microsoft technologies (Microsoft Azure, .NET WCF and WPF), while mobile client applications were implemented on every significant mobile platform (Android, iOS, Windows Phone, Windows 8). We parsed the source code mainly by the open source tool, NRefactory [10] which produced an AST. From the AST, we generated the architectural model by using the DSL API of VMTS. Note that the method used here is not specific to this case study, it can be applied on any projects written mainly in C#.

We created two test cases, i.e. a combination of components that build up the test system, on which the verification process is performed. We have defined the following two test cases: (i) Database + server: here, the application boundary is an API, since the database and the server does not have any graphical user interface. (ii) Database + server + admin client: in this case, the application boundary is the user interface of the admin client. After performing the verification process, we compared the real and the estimated source lines of code (SLOC). We calculated estimated SLOC values from function points based on the previously mentioned table [8]. However, there is no way to convert a fraction of a FP value to an SLOC estimation in language 1 and the remainder to language 2. This affects the accuracy of the estimation, because the tested application is not a pure C# application in either test case, there are some T-SQL and XAML language parts in the components as well. The estimated SLOC values, however, are probabilistic values (most likely, minimum and maximum values), which compensates the above inaccuracy of the conversion factors.

In test case 1 (Database + server), we identified 421 Function Points, which is translated to **12209** – **29470** (most likely **22734**) lines of C# code. The real application consisted 19696 lines of C# and 3055 lines of T-SQL code (**22751** in all). In test case 2 (Database + server + admin client), the result of the estimation was 699 Function Points, which is converted to **20271** – **48930** (most likely **37746**) lines of C# code. The actual SLOC values were 38365 lines of C#, 3055 lines of T-SQL and 5056 lines of XAML code (**46476** in all).

As it is shown above, the first estimation is almost exactly the same as the most likely estimation value. The difference is less than 0,1%! The second estimation is not that accurate, but the estimation is between the limits. After analyzing the verification results, we discovered that the cause of the relative inaccuracy in the second test case was the amount of XAML code, since a high percentage of the code was duplicated. This indicates that the verification algorithm should be enhanced with the capability of detecting code duplication. Apart from this, the results were convincing, the project management decided that the method is ready to be tested in production environment.

7 Conclusion

We designed and implemented a method that analyzes architectural models, and provides a cost estimation based on it. We did so because we discovered that nowadays, there is a great need for a cost estimation method that produces results automatically based on already available resources. Our technique is implemented using VMTS [1], a meta-modeling tool capable of creating and processing architectural models in the SysML language [3]. The advantage of the solution is that it does not need separate cost models, information is extracted from the architectural models automatically. Note that although we rely on an extended version of SysML, the extensions are used to create a more precise architecture model and they are not only used by cost estimation.

By automatizing the cost estimation, there is no need for extra time and effort allocated to the cost estimation. Another benefit is that the method uses a modified version of Function Point Analysis [2], which produces a platform independent result. In this way, the estimation process can be performed before even deciding, on which platform the software should be implemented. The result of the estimation can also be refined into source code estimations on different platforms. In this way, possible implementations using different technologies can be compared and the best can be selected. Besides presenting the method, we elaborated a basic, preliminary verification method and discussed the results. Although the verification was not based on real production process, its promising results show that the method is worth to examine further. In the future, we plan to test the method on several production scenarios and use it in real development environments and add support for estimating the cost of mixed platform projects.

8 Acknowledgement

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred.

9 References

1. Visual Modeling and Transformation System (VMTS): https://www.aut.bme.hu/en/Pages/Research/VMTS/Introduction
2. David Longstreet: Function Point Training and Analysis Manual: http://www.softwaremetrics.com/Function%20Point%20Training%20Booklet%20New.pdf
3. OMG SysML 1.3: http://www.omg.org/spec/SysML/1.3/
4. Constructive Cost Model II (COCOMO II): http://csse.usc.edu/csse/research/COCOMOII/cocomo_main.html
5. Arlene F. Minkiewicz, Estimating Software from Use Cases & Estimating Software from Requirements: http://legacy.pricesystems.com/research/white_papers.asp
6. Steffen Becker, Heiko Koziolek, Ralf Reussner, The Palladio component model for model-driven performance prediction, Journal of Systems and Software, v.82 p.3-22, January, 2009
7. STSC, Software Development Cost Estimating Guidebook: http://www.stsc.hill.af.mil/consulting/sw_estimation/softwareguidebook2010.pdf
8. Quantitative Software Management Inc., Function Point Languages Table: http://www.qsm.com/resources/function-point-languages-table
9. mCPS - End to End Mobile Publication: http://www.evoprogroup.com/page/mcps
10. Daniel Grunwald, Using NRefactory for Analyzing C# Code: http://www.codeproject.com/Articles/408663/Using-NRefactory-for-analyzing-Csharp-code

Discrete-Continuous Semantic Adaptations for Simulating SysML Models in VHDL-AMS

Daniel Chaves Café[1,2], Cécile Hardebolle[1], Christophe Jacquet[1], Filipe Vinci dos Santos[2], and Frédéric Boulanger[1]

[1] Supélec E3S – Computer Science Departement,
[2] Thales Chair on Advanced Analog System Design,
{`daniel.cafe, cecile.hardebolle, christophe.jacquet, filipe.vinci, frederic.boulanger`}`@supelec.fr`

Abstract. Our research focuses on the simulation of heterogeneous systems modeled in SysML, in particular, systems that mix different engineering domains such as mechanics, analog and digital circuits. Because of their nature, expressing multi-paradigm behavior in heterogeneous systems is a cumbersome endeavor. SysML does not provide a standard method for defining the operational semantics of individual blocks nor any intrinsic adaptation mechanism when coupling blocks of different domains. We present in this paper a way to address these obstacles. We give well-defined operational semantics to SysML blocks by using profile extensions, together with a language for the description of adaptors. We apply our approach to a test case, using a toolset for SysML to VHDL-AMS transformation, capable of automated generation of VHDL-AMS code for system verification by simulation.

1 Introduction

In the Electronic Design Automation (EDA) industry, the need for modeling and verification of mixed-signal systems gave rise to several system design languages supporting Analog and Mixed Signal (AMS) extensions. Some examples are VHDL-AMS [6] and SystemC-AMS [8]. These extensions support the use of different models of computation concurrently in a single design thus enabling the modeling of heterogeneous systems. As complexity increases, these textual languages are no longer suitable for proper documentation and communication among different teams. For these use cases, graphical languages are preferable, and they play well with Model Driven Engineering workflows.

SysML, the Systems Modeling Language, is an industry standard for systems specification. It provides a large set of diagrams which can be used to specify system's requirements, model their behavior or even detail the interconnections of structural blocks. Despite its flexibility, SysML does not provide clear semantics. On the one hand, this can be helpful for engineers wishing to describe systems in an early development phase, especially when some implementation details are not yet entirely defined. In this case, SysML is a helpful communication tool.

On the other hand, the lack of clear semantics can be cumbersome if one wants to run simulations from the SysML diagrams.

For the purpose of solving the lack of semantics of SysML diagrams, we have developed a technique to generate executable code from SysML models which is based on two foundations : (a) Explicitly state the semantics of modeling elements, and (b) Define the semantic adaptations between heterogeneous models. The focus of this work is the creation of an adaptor instantiation language for semantic adaptation for specifying interfaces precisely and without ambiguity.

The organization of this article is as follows : The state of the art is presented in section 2 detailing existing heterogeneous modeling techniques. A case study is then introduced in section 3 to illustrate the problem. The actual implementation of our solution is detailed in section 4. And finally we discuss the approach and the results in sections 5 and 6.

2 Related Work

One of the precursors of heterogeneous modeling is the well-known Ptolemy II [7] framework. Here, heterogeneity is handled by hierarchy. Components are nested in black boxes called actors for which the semantics of execution and communication are described by an entity called *Director*. It defines the model of computation (MoC) of the actor. In Ptolemy, computation and communication are defined for a large set of MoCs. These include Process Networks, Dataflow, Discrete Event, Finite State Machines, Continuous Time and others. Unfortunately, Ptolemy does not provide explicit ways to define adaptations between models that use different MoCs. For example, interactions between discrete event (DE) and synchronous dataflow (SDF) models can result in redundant events in the DE domain if a given value does not change. In the same way, an SDF model might not be regularly activated as discussed in [2].

ModHel'X [10] was developed to explore semantic adaptations in heterogeneous models. ModHel'X improves upon the execution algorithm of Ptolemy by introducing an adaptation phase. This yields an effective way to define the semantics of the interactions between different models of computation. The current implementation of ModHel'X is based on a non-standard metamodel which makes it hard to integrate with existing toolchains.

We propose to introduce ModHel'X's good practices of stating the semantics of different components and explicit modeling of the semantic adaptation between heterogeneous components, into an industry standard modeling language: SysML. In our approach SysML acts as a pivot language from which we generate executable code for widely deployed languages, such as SystemC-AMS and VHDL-AMS. We use a custom profile to extend the semantics of SysML blocks for continuous-time and discrete-event blocks. These two domains are generalized into two stereotypes $\ll$ *analog* $\gg$ and $\ll$ *digital* $\gg$. A third stereotype is dedicated to the description of $\ll$ *adaptor* $\gg$ blocks. Those provide explicit behavior on how to adapt data, time and/or control. To do so, a mini-DSL was designed to allow the instantiation of off-the-shelf types of adapters. Depending on the target language these could either be present in standard libraries or custom designed.

Substantial work has been carried out to apply SysML/UML to the design of electronic (analog and digital) systems. Many researchers focused on the generation of VHDL-AMS code from SysML diagrams. D. Guihal [9] and J. Verriers [14] extended the VHDL metamodel proposed in [1] and [13] to use AMS constructions in their code generators. J.-M. Gautier et al. [3] used model transformations to generate VHDL-AMS code from SysML Block Definition Diagrams and Internal Block Diagrams. They have used block constraints to define physical equations in VHDL-AMS modules.

Although these previous works have shown methods to generate VHDL-AMS code from SysML diagrams, they have not dealt with the semantic inconsistencies that heterogeneity introduces. Our previous work [4] presents a technique to deal with this problem. It targets SystemC-AMS simulation language. The present work is a follow-up that introduces a new adaptor instantiation language and MoC definition mechanisms that are better suited for model driven engineering. This is also an opportunity to show that previously developed techniques apply to other target languages as well, namely VHDL-AMS.

3 A case study of a MEMS Accelerometer

Micro Electro Mechanical Systems (MEMS) motion-sensing devices are a good example of heterogeneous systems that mix mechanical, analog and digital components in the same system. They can be used to measure a variety of physical quantities such as acceleration, pressure, force, or chemical concentrations. To make such measurements, MEMS sensors can take advantage of several transduction mechanisms, for example, piezoresistive or capacitive sensing. Here we build a simple model of a capacitive sensing accelerometer to illustrate our proposal.

3.1 Description of the system

Our case study is a capacitive sensing accelerometer composed of two electrodes and an intermediary membrane free to move only in the vertical axis as illustrated in figure 1. This structure forms two capacitors between the middle membrane and both the top and bottom walls. The vertical movement of the membrane implies the variation of both capacitances since $C \propto 1/(g_0 \pm x)$, where g_0 is the gap distance at rest and x is the displacement of the membrane from rest. One can either connect the membrane to ground hence fixing the middle voltage V_{middle} to zero or one can leave it disconnected thus fixing the current to zero. In the first case, the change in stored charge caused by the displacement of the membrane leads to a current flow. In the second case, since the middle electrode is disconnected, there is no current flow, and by charge conservation the voltage across the membrane must change with the displacement.

Using the second method, we obtain a linear relation between the membrane's voltage and its displacement provided that we apply a symmetric voltage on both top and bottom electrodes (i.e. $V_{top} = -V_{bottom} = V_0$) as explained in [5]:

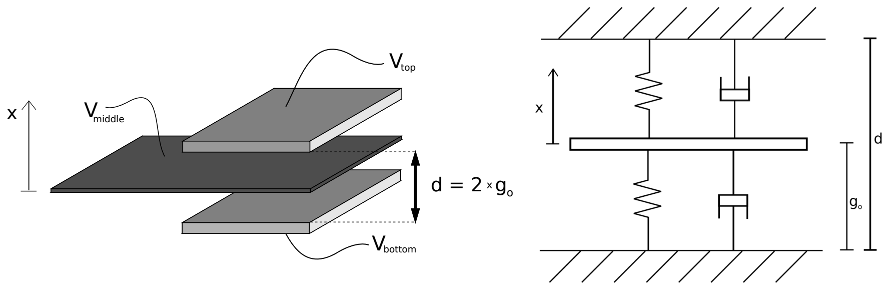


Fig. 1. Electrical vs Mechanical Model

$$V_{middle} = V_0 \frac{x}{g_0} \tag{1}$$

The membrane's displacement depends on several forces. In our example we consider only inertial, spring and friction ones. These are assumed to act exclusively at the center of the membrane. The spring force is proportional to displacement and the damping (friction) to velocity. We are here interested in studying the behavior of this system when an external force is applied to the membrane, typically gravity, but it could be any external force. Applying Newton's law, we end up with:

$$F_{external} = -kx - c\dot{x} + m\ddot{x} \tag{2}$$

Several precautions must be taken to accurately extract V_m. Our model uses the most simple read-out circuit, an operational amplifier configured as a buffer. The output of the buffer is fed to a voltage comparator, giving a one-bit output that undergoes further processing in the digital domain. The details of the rest of the system fall outside of the scope of the discussion.

The model of the operational amplifier consists of a single piecewise equation considering the gain and saturation. The latter assures that the output does not exceeds the supply voltages of $V_{DD} = +15V$ and $V_{SS} = -15V$. The piecewise equation is as follows:

$$V_{out}(V_{in}) = \begin{cases} V_{SS} & : V_{in} < V_{SS}/gain \\ V_{DD} & : V_{in} > V_{DD}/gain \\ V_{in} \times gain & : elsewhere \end{cases} \tag{3}$$

3.2 SysML Model

In SysML, we have divided the system into five major blocks as illustrated in figure 2: the **accelerometer** models the electromechanical dynamics, the **opamp** models the operational amplifier, the **sampler** adapts analog data to the digital world by periodic sampling, the **comparator** checks for a threshold

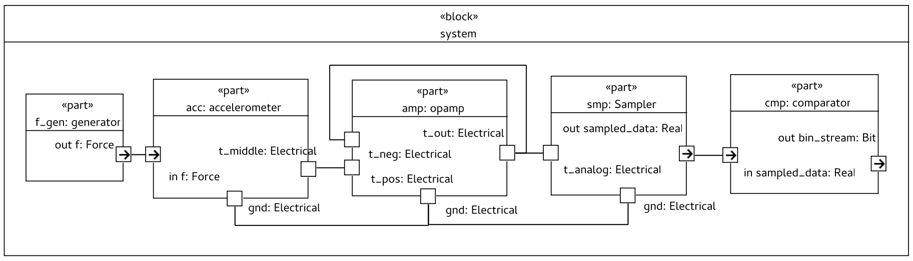


Fig. 2. SysML Model [IBD]

crossing generating a bit stream from the output of the sampler, and finally a **source** sine wave force generator stimulates the model.

While analog blocks use differential equations defined in continuous time, digital circuitry is best modeled in the discrete domain. These two formalisms handle different types of data and react differently to inputs. If we wish to model and simulate a system with both of them together, we must specify not only the operational semantics (i.e. the Model of Computation) of a particular block but also the semantic adaptation between both domains.

To solve the semantic ambiguity issue, we use custom stereotypes defined in a separate profile. Stereotypes are element modifiers that allow us to give precise meaning to base elements of SysML. In our case, we have chosen to apply specific stereotypes to SysML blocks in order to specify the use of a given model of computation. Since we are dealing with continuous-digital integration, we have added the notion of $\ll$ *analog* $\gg$ and $\ll$ *digital* $\gg$ blocks to SysML as seen in figure 3. We have also added one stereotype $\ll$ *adaptor* $\gg$ to specify blocks that are in the frontier of two different MoCs.

For an analog block, we use SysML/UML constraints to describe the physical relations shown previously. The equations defined in SysML constraints are considered to be continuous. The interconnections in an analog block impose other equations that can be inferred from the topology of the system. In the case of electrical circuits, these are the Kirchoff laws. Digital blocks on the other hand are connected by signals that transmit events. Even though in the real world, digital circuits have analog behavior, this formalism abstracts these electrical phenomena making digital circuits design simpler.

One particular case that is worth noting here is the definition of piecewise equations. We have used a particular syntax in SysML constraints to describe these kind of relations. For instance, equation 3 describes the simplified behavior of an operational amplifier and is represented by one SysML constraint preceded by the keywords *PIECEWISE FUNCTION* in our mini-DSL (see figure 4-left). In VHDL-AMS, this is translated to *USE* conditions as we see in figure 4-right.

We have also defined the quantities that exists between terminals, such as voltage or current using SysML properties (see figure 3). These are translated to VHDL-AMS quantities directly.

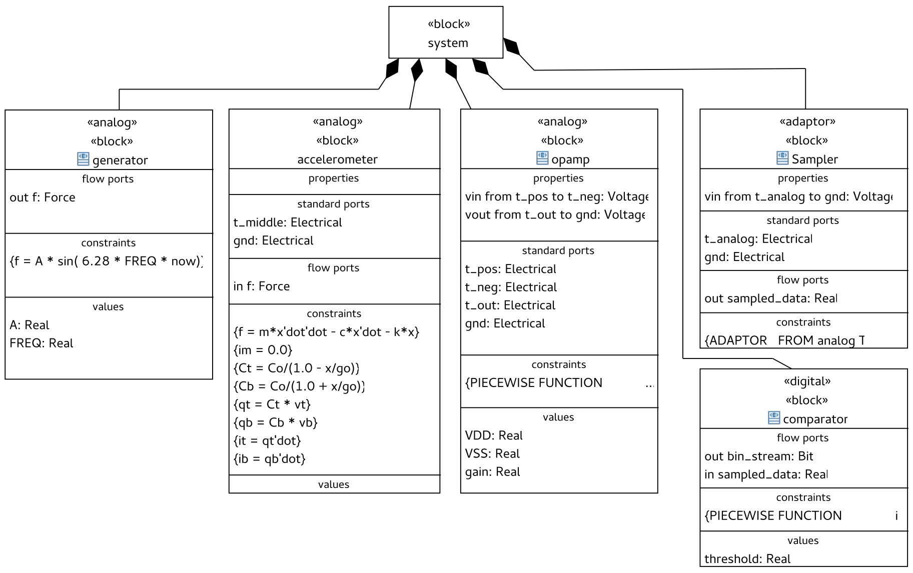


Fig. 3. SysML Model [BDD]

```
PIECEWISE FUNCTION
  V_in < VSS/gain:
     V_out = VSS,
  V_in > VDD/gain:
     V_out = VDD,
  elsewhere:
     V_out = gain * V_in
```

```
IF V_in'ABOVE(VDD/gain) USE
  V_out == VDD;
ELSIF NOT V_in'ABOVE(VSS/gain) USE
  V_out == VSS;
ELSE
  V_out == gain * V_in;
END USE;
```

Fig. 4. Definition of piecewise equations (SysML vs VHDL-AMS)

3.3 Adaptation Mechanisms

The $\ll$ *adaptor* $\gg$ stereotype defines a block whose main purpose is to adapt data, time and/or control from one domain to another. This special block defines the adaptation semantics of heterogeneous interfaces. If they are well defined, then generating executable code from that model should produce the same result regardless of the target language (VHDL-AMS in this case, but it could be any other AMS-capable language, such as SystemC-AMS).

In our example, the block **sampler** is an adaptor from analog to digital domain. It samples data periodically. We do not specify the behavior of the adaptor using our language, rather we instantiate and parameterize a pre-defined adaptor. This is achieved using a SysML constraint starting with the $ADAPTOR$ keyword. Figure 5 shows our mini-DSL being used to instantiate the sampler.

Analog data is generated at a dynamic timestep in VHDL-AMS simulators. The adaptor specification guarantees that output data will be sampled at a fixed *timestep* of $2\,\mu$s. This case of adaptor is interesting because we are not only

```
ADAPTOR
  FROM analog TO digital
  IS sampler
  PARAMS
    input     : vin,
    output    : sampled_data,
    timestep  : 2us
```

Fig. 5. Adaptor specification in SysML constraints

adapting the time base but also the data format. In the analog domain, ports are considered to be terminals connected to nodes of a circuit. Kirchhoff equations can be then deduced from the topology of the circuit. The adaptor must extract the voltage between the input terminal and a reference and propagate it to a discrete event domain as data tokens. The parameters *input* and *output* of figure 5 indicate the analog voltage to read and the binary stream to write.

Certain adaptors that we make available to system designers have off-the-shelf counterparts in the target language; others do not. In the former case, our transformation chain chooses adaptors from a standard library; in the latter case it defines new adaptors in the target language.

In this case study, the sampler isn't present in the VHDL-AMS library so we generate the module responsible for this specific adaptation. The generated code can be separated into two different VHDL processes. One for setting the time step and a second one, triggered by the first, to model the adaptor semantics. In this case, the semantics are fairly simple. It consists on copying the analog voltage from input terminal and its reference to the discrete event output at a scheduled moment in time, i.e. every $2\,\mu$s.

The output of the sampler is connected to a comparator which will generate a bit stream from its input. When the input analog voltage crosses a value given by the *threshold* parameter, the digital output switches to a logical value of '1', or '0' otherwise.

4 Model Transformation

Our approach, illustrated in figure 6 consists in two separated phases. Starting from a SysML model, we first perform a model-to-model (M2M) transformation T_1 in order to obtain a VHDL-AMS model. We then generate VHDL-AMS code through a model-to-text (M2T) transformation T_2.

The model-to-model step T_1 translates every SysML element into its equivalent VHDL-AMS element. This step provides the model with semantics on how to interpret SysML elements. For instance, UML ports are converted to terminals while SysML flow ports are translated to quantity ports. In the same way, a SysML constraint will be transformed into an equation with its variables translated into VHDL-AMS quantities.

For this first step, we have used the Atlas Transformation Language (ATL) [11]. ATL is a language for defining model transformations by a set of rules. Being a

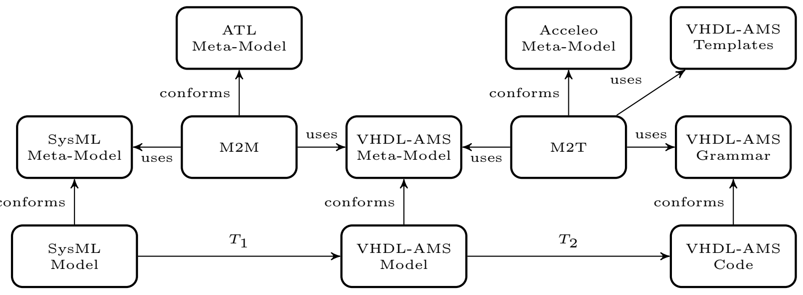


Fig. 6. Our approach

model itself, the transformation has its own meta-model. ATL is based on pattern recognition of input elements and conditions which trigger the creation of output elements in the resulting model.

The VHDL-AMS metamodel is an improvement from previous works [1, 9, 14]. It includes the notion of parameterizable adaptors and some slight modifications to the general structure of how libraries are used inside a model. This has proven to be very practical in our implementation but it introduces elements to the metamodel that are not totally part of VHDL-AMS. Instead, a two-step approach separating pure syntax from semantics could also be considered.

Finally, transformation T_2 is responsible for generating the actual code that will be used for running the simulation. For this we use the ACCELEO [12] model-to-text engine from Obeo. In ACCELEO we write templates that specify the code to generate for the various model elements. The adaptors, for instance were instantiated depending on their type. This is specified in our mini-DSL by the keyword *IS* and is parameterized by the list of parameters listed after *PARAMS*. The generated code is a template with two VHDL processes (as explained in section 5).

5 Simulation Results

Applying both transformations (T_1 and T_2) to the SysML model presented in figure 3, we obtain several VHDL-AMS files (one per block) which we use to run simulations. In figure 7 we show the output of the Hamster VHDL-AMS simulation tool for a sinusoidal input force.

Note that, despite the non-linear variation of both top and bottom capacitances, the output voltage is linear and follows the input stimulus, which is conform to equation 1. The left side of figure 7 allows us to conclude that the threshold detection mechanism described by the block **comparator** works correctly as the binary stream output follows the sign of the **opamp**'s output.

A closer look allows us to confirm that digital data is sampled at a fixed timestep even tough the analog data is not. The signal *clk* generates events every $2\,\mu$s, both on the rising and falling edge. The detail of the right part of figure 7 shows that the output voltage was already negative several simulation

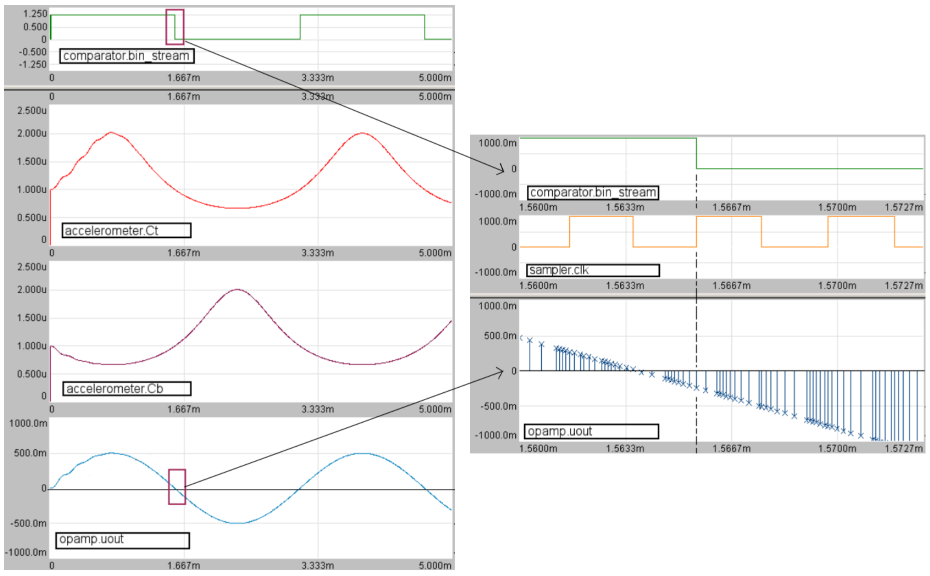


Fig. 7. Simulation Results

cycles before the threshold detection. This translates into a delay between the effective crossing of the threshold (bottom left of figure 7) and its detection. This is the expected behavior since the specification of the adaptor constraint of figure 5 specifies a 2 μs sampling period thus there can be a delay of up to 2 μs.

6 Conclusions

In this paper we introduce an approach for simulating continuous-digital interaction in SysML models. We validate the behavior through simulation and we generate executable VHDL-AMS code using automatic model transformations. We address the ambiguity of SysML diagrams by assigning them concrete semantics (MoCs) using a simple profile. In order to solve the semantic adaptation problem, we explicitly design adaptation mechanisms using a dedicated language based on SysML constraints. These are translated into specific VHDL-AMS constructs that enforce the specified behavior.

The case study presents a typical case where integration issues occur. We have specified not only the model of computation of individual SysML blocks using stereotypes but also the semantic adaptations between continuous and discrete domains using the notion of adaptors.

In future work, we wish to separate the semantic parts of our transformation from purely syntactical ones. This would allow us to focus on a more generic approach so as to deal with heterogeneous interactions independently from the language used to run simulations. In order to do so, we have considered using a generic intermediary metamodel to facilitate transformations to other languages.

References

1. V. Albert. Traduction d'un modèle de système hybride basé sur réseau de Petri en VHDL-AMS. *Master de conception en architecture de machines et systèmes informatiques, Université Paul Sabatier, LAAS-CNRS*, 2005.
2. Frédéric Boulanger and Cécile Hardebolle. Execution of models with heterogeneous semantics. In *Tutorial on critical systems simulation at CSDM'12*, December 2012.
3. Fabrice Bouquet, Jean-Marie Gauthier, Ahmed Hammad, and Fabien Peureux. Transformation of SysML structure diagrams to VHDL-AMS. In *Design, Control and Software Implementation for Distributed MEMS (dMEMS), 2012 Second Workshop on*, pages 74–81. IEEE, 2012.
4. Daniel Chaves Cafe, Filipe Vinci dos Santos, Cecile Hardebolle, Christophe Jacquet, and Frederic Boulanger. Multi-paradigm semantics for simulating SysML models using SystemC-AMS. In *Specification & Design Languages (FDL), 2013 Forum on*, pages 1–8. IEEE, 2013.
5. Franck Chollet and Haobing Liu. A (not so) short introduction to Micro Electro Mechanical Systems. http://memscyclopedia.org/introMEMS.html, pages 149-152. Nov 2013.
6. Ernst Christen and Kenneth Bakalar. VHDL-AMS - a hardware description language for analog and mixed-signal applications. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 46(10):1263–1272, 1999.
7. Johan Eker, Joern W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
8. Christoph Grimm, Martin Barnasconi, Alain Vachoux, and Karsten Einwich. An introduction to modeling embedded analog/mixed-signal systems using SystemC AMS extensions. In *DAC2008 International Conference*, 2008.
9. Guihal, D and Andrieux, L and Esteve, D and Cazarre, A. VHDL-AMS model creation. In *Mixed Design of Integrated Circuits and System, 2006. MIXDES 2006. Proceedings of the International Conference*, pages 549–554. IEEE, 2006.
10. Cécile Hardebolle and Frédéric Boulanger. ModHel'X: A component-oriented approach to multi-formalism modeling. In *Models in Software Engineering*, pages 247–258. Springer, 2008.
11. Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. *Satellite Events at the MoDELS 2005 Conference*, pages 128–138, 2006.
12. J. Musset, E. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire. Acceleo user guide, 2006.
13. Guillaume Savaton, Jérôme Delatour, and Karl Courtel. Roll your own hardware description language. In *OOPSLA & GPCE Workshop Best Practices for Model Driven Software Development*, 2004.
14. Jean Verries. *Approche pour la conception de systèmes aéronautiques innovants en vue d'optimiser l'architecture. Application au système portes passager.* PhD thesis, Université Paul Sabatier-Toulouse III, 2010.

ProMoBox in Practice : A Case Study on the GISMO Domain-Specific Modelling Language

Romuald Deshayes[1], Bart Meyers[2], Tom Mens[1], and Hans Vangheluwe[2,3]

[1] Département d'Informatique, Université de Mons, Mons, Belgium
firstname.lastname@umons.ac.be
[2] Modeling, Simulation and Design Lab (MSDL), University of Antwerp, Belgium
firstname.lastname@uantwerp.be
[3] Modeling, Simulation and Design Lab (MSDL), McGill University, Canada

Abstract. Domain-specific modelling (DSM) helps designing systems at a higher level of abstraction, by providing languages that are closer to the problem space than to the solution space. Unfortunately, specifying and verifying properties of the modelled system has been mostly neglected by DSM approaches. At best, this is only partially supported by translating models to formal representations on which properties are specified and evaluated based on logic-based formalisms. This contradicts the DSM philosophy as domain experts are usually not familiar with such formalisms. To overcome this shortcoming, the *ProMoBox* approach lifts property specification and verification tasks up to the domain-specific level. For a given DSM language, some operations at the metamodel level are needed to allow specification and verification of properties. This paper reports on a practical case study of how to apply the *ProMoBox* approach on *GISMO*, a DSM language designed specifically for developing gestural interaction applications.

1 Introduction

Domain-specific modelling (DSM) helps designing systems at a higher level of abstraction. By providing languages that are closer to the problem domain than to the solution domain, low-level technical details can be hidden. An essential activity in DSM is the specification and verification of properties to increase the quality of the designed systems [1]. Providing support for these tasks is therefore necessary to provide a holistic DSM experience to domain engineers. Unfortunately, this has been mostly neglected by DSM approaches. At best, support is limited to translating models to formal representations on which properties are specified and evaluated with logic-based formalisms [2], such as Linear Temporal Logic (LTL). This contradicts the DSM philosophy as domain experts desiring to specify and verify domain-specific properties are not familiar with such formalisms.

In previous work we proposed the *ProMoBox* framework to shift property specification and verification tasks up to the DSM level. The scope, assumptions and limitations of this approach are presented in [3]. The *ProMoBox* framework consists of (i) generic languages for modelling all artefacts that are needed for specifying and verifying properties, (ii) a fully automated method to specialise and integrate these generic languages in a given DSML, and (iii) a verification backbone based on model checking that is directly pluggable to DSM environments such as AToMPM [4]. Properties in *ProMoBox* are translated to LTL and a Promela model is generated that includes a translation of the system, its environment and its rule-based operational semantics. The Promela model is checked with the SPIN model checker [5] and if a counter-example is found it is translated back to the DSM level.

This paper presents a case study that applies *ProMoBox* to *GISMO*, a DSML for executable modelling of gestural interaction applications. We illustrate how to make some minor changes and additions to the metamodel of the DSML in order to enable the generation of all needed languages (an input language, output language, property language, and runtime language) that are required for the specification and verification of properties at DSM level. We subsequently report on the results of verifying these properties.

The paper is structured as follows. Section 2 presents the *GISMO* DSML used as a case study. Section 3 explains the changes required to the *GISMO* metamodel in order to apply the *ProMoBox* approach. Section 4 presents the results of applying *ProMoBox* to *GISMO*. Section 5 exemplifies the specification of domain-specific properties on *GISMO* models and provides some results and counter-examples found after verification of these properties. Section 6 presents related work, and Section 7 concludes.

2 *GISMO*: a DSML for gestural interaction

GISMO is a DSML developed by the first author to facilitate development of gesture-based interactive applications [6]. Specifying gestural interaction with objects is achieved in a state-based way. The *GISMO* metamodel is depicted in Fig. 1. A *GISMO* model is composed of states and gestures. State changes may be performed when a gesture is performed by a user. In previous work we have developed a framework [7] that takes care of interpreting such high level gestures from the raw data coming from 3D sensors such as Microsoft's Kinect motion sensor. The operational execution semantics of *GISMO* is based on ICO [8], a formalism based on high-level Petri nets.

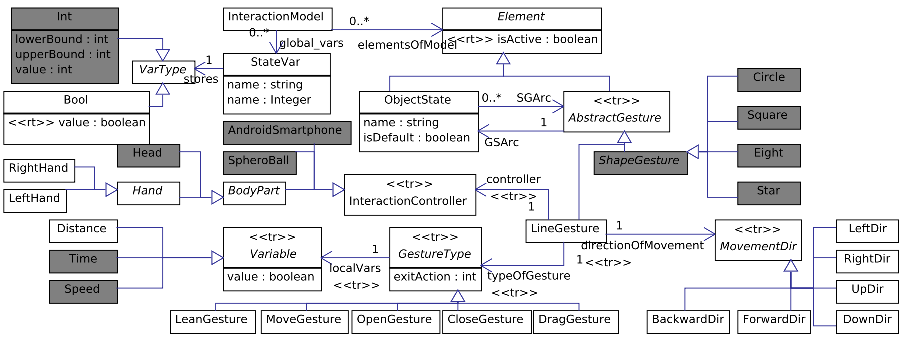


Fig. 1. Metamodel of the *GISMO* DSML for gestural interaction.

Fig. 2 shows an example of a domain-specific model in *GISMO*, representing the gestural interaction of a user with a virtual bow as part of some computer game. States are represented as labelled rounded rectangles. The unique active state is displayed in green. Gesture boxes are used to specify the gesture that is expected from the user. They are composed of the body part (e.g., left or right hand) involved in the gesture, the direction of movement (e.g., left, right, up, down), the type of gesture (e.g., moving, dragging, opening) and the observed dimension of the gesture (i.e., distance d, time t or speed s).

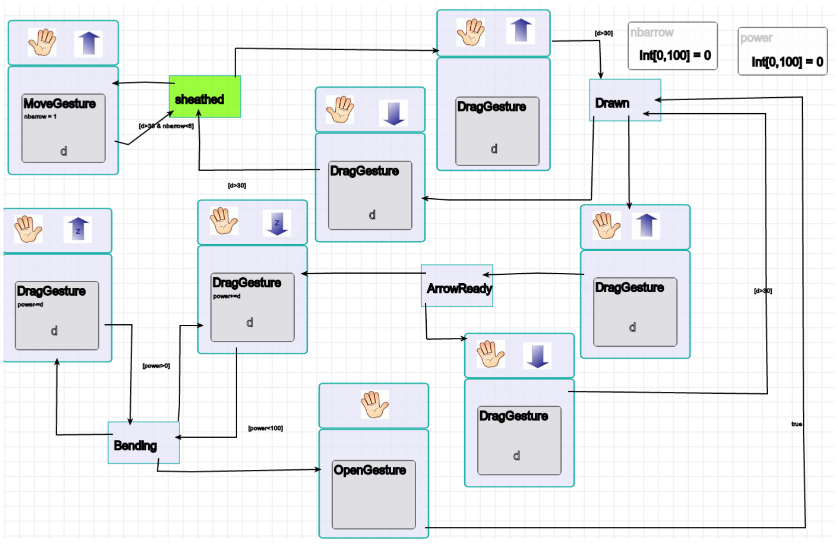


Fig. 2. GISMO model representing the expected gestural interaction of a user with a virtual bow.

The model is executed as follows. Whenever the user performs a specific gesture matching an expected outgoing gesture from the currently active state, the active state changes to the destination of the outgoing edge of the performed gesture. Global variables can be defined on *GISMO* models (e.g. nbarrow and power in Fig. 2). They store relevant information about the object being modeled. Triggering a state change can be conditioned by a boolean precondition that may combine the observed dimension of the gesture and the global variables. For example, a state change from *Drawn* to *ArrowReady* is triggered if the distance d of the *drag* gesture exceeds 30cm and the global variable nbarrow is strictly positive. Exit actions of a matching gesture can be executed to perform operations on a global variable, such as reassigning its value by the value retrieved from the observed dimension. In the example of Fig. 2, the two *drag* gestures connected to state *Bending* add or subtract from global variable power the value of the gesture's distance dimension.

3 Simplifying the *GISMO* metamodel

Enabling verification of domain-specific model properties requires a certain amount of preparatory work in order to make it applicable in practice. In particular, the *ProMoBox* approach [3] requires the metamodel of the DSML under study to be annotated before starting the generation process. Depending on the complexity of the language, other changes may be required as well. Most of the simplifications aim to reduce the combinatorial search space of the SPIN model checker. Not performing such simplifications would result in an exponentially longer verification time. This scalability problem is inherent to model checking, as the search space grows very fast in terms of the number of different possible inputs and input types.

This section focuses on the changes and simplifications required to the *GISMO* metamodel in order to enable specification and verification of domain-specific properties. A first simplification ignores the *time* and *speed* dimensions of input gestures. Thus, we restrict ourselves to properties based on the *distance* dimension only. Moreover, as unbounded variable domains are too costly to check, we provided a binary classification of a gesture's *distance* value into either small or big. A wider range of distance classes could be used, at the expense of a longer verification time.

As another simplification we limit the types of global variables to boolean values only, i.e., integer values are not supported during property verification. We also represent each global variable by a unique integer identifier instead of a variable name since strings are not supported by Promela, the verification modelling language to which our DSML models are translated. This is not a huge concern since it can be fully automated and made transparent for the DSML engineer.

Finally, we simplified the preconditions and exit actions of *GISMO* models. In *GISMO*, preconditions are boolean expressions resulting from the logical composition (*AND, OR, NOT*) of comparisons between global variables, gesture dimensions and integer values (*e.g.,* nbarrow>0 & d$\geqslant$30 between *Drawn* and *ArrowReady* states in Fig. 2). Exit actions, on the other hand, can express assignment operations of global variables (*e.g.,* power += d in the gestures linked to the *Bending* state). For model verification we limit the precondition checks to verifying if a global variable is true or false; and exit actions are limited to changes of the boolean value of the global variable. Generally speaking, most of the simplifications on the *GISMO* DSL can be semi-automated and applied to any DSL, thus reducing the task of the domain engineer.

4 Applying *ProMoBox* to *GISMO*

The *ProMoBox* framework [3] relies on a family of fully automated generated modelling languages based on the DSML metamodel. These languages are required to modularly support specification and verification of model properties. The *design language* allows DSM engineers to design the static structure of the system. The *runtime language* enables modellers to define a state of the system, *e.g.,* an initial state as input of a simulation, or a particular "snapshot" during runtime. The *input language* lets the DSM engineer model the behaviour of the system environment, *e.g.,* by modelling an input scenario as an ordered sequence of events containing one or more input elements. The *output language* can be used to represent execution traces (expressed as ordered sequences of states and transitions) of a simulation or to show verification results in the form of a counter-example. Output models can also be created manually as part of an oracle for a test case. The *property language* can be used to express properties based on modal temporal logic, including structural logic and quantification.

A fully automated method specialises and integrates these languages to any given DSML, thus minimising the effort of the language engineer. This is realised by manually annotating the DSML metamodel entities (classes, associations and attributes) with the necessary stereotypes required for every language construct. Stereotype $\ll rt \gg$ (for runtime) annotates metamodel entities that serve as output (*e.g.,* a state variable); stereotype $\ll ev \gg$ (for event) annotates entities that serve as input only (*e.g.,* a marking); stereotype $\ll tr \gg$ (for trigger) annotates static entities that may also serves as input (*e.g.,* a transition event). More stereotypes could be envisioned, but the generation of the sub-languages currently only supports those three. Stereotypes $\ll ev \gg$ and $\ll tr \gg$ cannot be used jointly on the same metamodel entity.

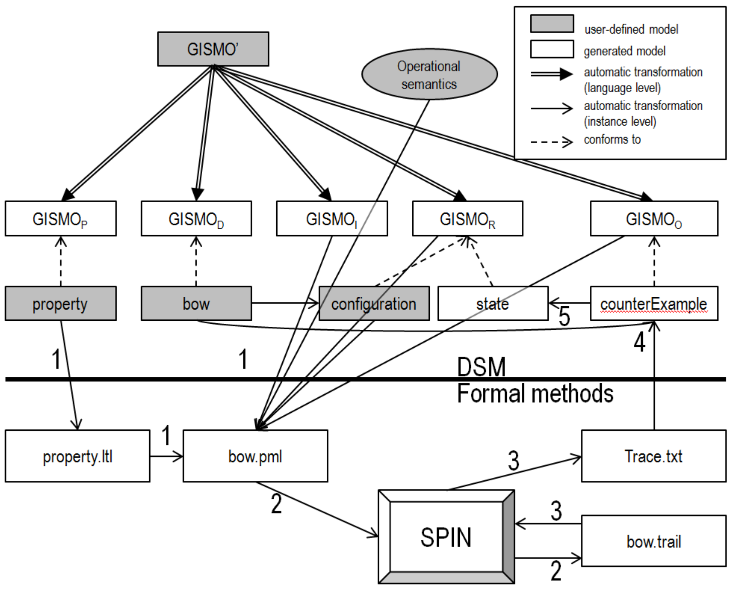


Fig. 3. The *ProMoBox* approach applied to *GISMO*.

Fig. 3 illustrates how to apply *ProMoBox* to *GISMO*. Only the grey parts of Fig. 3 need to be modelled explicitly, the white parts are generated from the annotated and simplified *GISMO*' metamodel, shown in Fig. 1. Classes colored in grey have been removed by the simplification process; stereotype annotations have been added.

A family of languages is generated from *GISMO*' using a template-based approach. This approach makes *ProMoBox* applicable to different types of DSMLs. The main idea is that generic metamodel elements (shown as grey rectangles in Fig. 4) are interwoven with the DSL metamodel elements.

As an example, Fig. 4 shows the metamodel of the generated *output language* $GISMO_O$. The metamodel of generated *design language* $GISMO_D$ closely resembles $GISMO_O$, except that the grey classes in the figure are absent, *OutputElement* is replaced by *DesignElement*, and the *isActive* attribute of *Element* and the *value* attribute of *Bool* is also absent, because of the $\ll rt \gg$ annotations in Fig. 1. The metamodel of generated *runtime language GISMO*$_R$ looks like *GISMO*$_O$, except that the grey classes are absent, and *OutputElement* is replaced by *RuntimeElement*. Fig. 2 shows an example runtime instance model of *GISMO*$_R$, representing a snapshot of the bow model during its execution. The currently active state is displayed in green. The generated metamodel of input language *GISMO*$_I$ is depicted in Fig. 4, together with an example instance model. This model represents a sequence of a *MoveGesture* followed by two *DragGestures* (each gesture is surrounded by a green circle) provided by the user as input and processed by the operational semantics rules to execute the runtime model *GISMO*$_R$. The generated metamodel of *property language GISMO*$_P$ allows to define temporal properties over the system behaviour by means of four constructs: quantification ($\forall$ or $\exists$), temporal patterns, structural patterns and domain-specific pattern elements. These property-related constructs are added to the *GISMO*' metamodel by weaving the generic metamodel template of Fig. 5 into the DSML. The resulting language has a concrete syntax that highly corresponds to the DSML, and the quantification and temporal patterns (instead of LTL operators) are raised to a more intuitive level by using natural language. The full metamodel of *GISMO*$_P$ is not shown here, but a similar example can be found in [3].

Properties specified in *GISMO*$_P$ are translated to LTL, and a Promela model is generated that includes a translation of the initialised system, the environment, and the rule-based operational semantics of the system. This translation is generic, and thus independent of the DSML. The properties are checked by the SPIN model checker. If any counter-example is found, the verification results are translated back to the DSM level.

The limitations of the framework are related to the mapping to Promela as explained in [3]. In its current state, *ProMoBox* does not allow dynamic structure models. Because of the nature of Promela, boundedness is ensured in the translation. Other constraints can be circumvented, as described in Section 3.

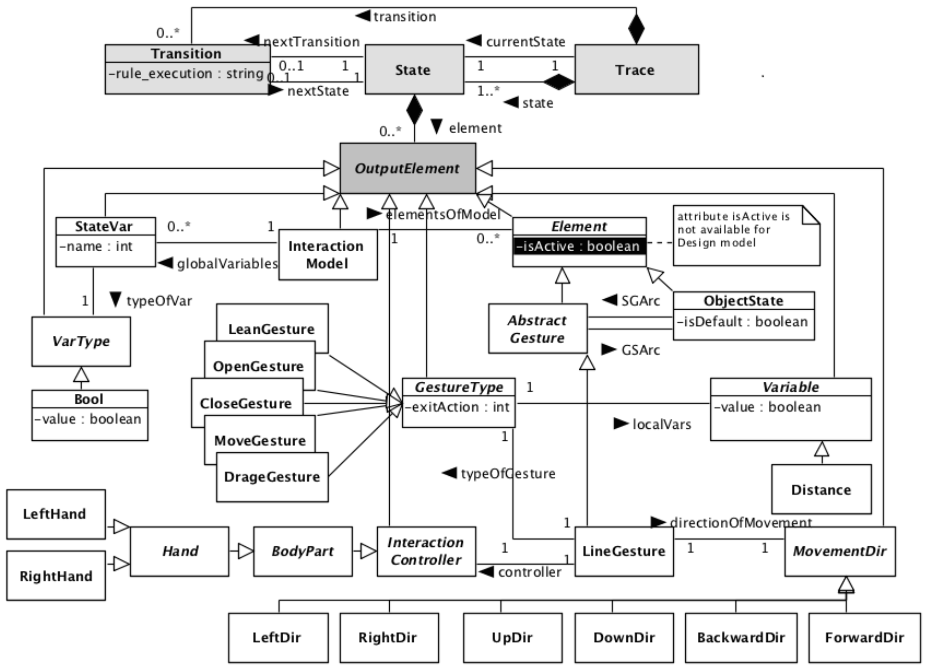
5 Specifying and checking properties on GISMO models

We implemented the *ProMoBox* framework in AToMPM [4], and the generic compilers that compile models to and from Promela or text were written in Python. The resulting generated Promela code is around 800 lines of code.

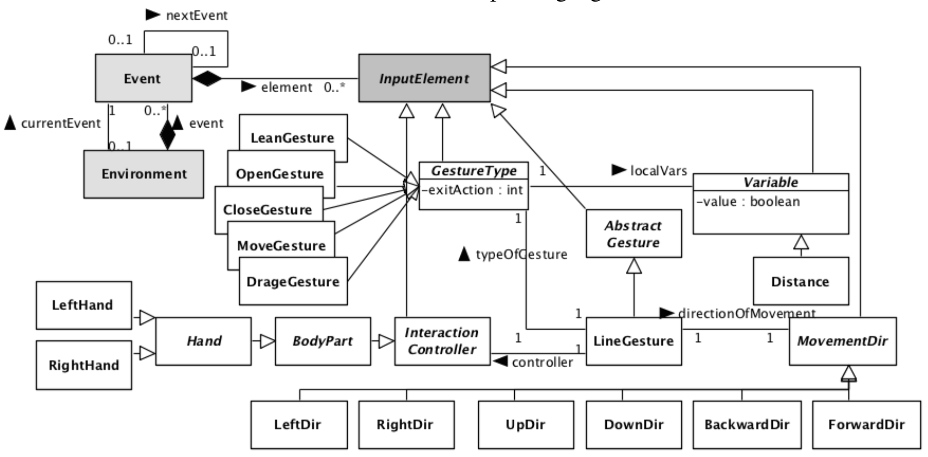
We verified fifteen properties on the runtime instance model of Fig. 2. Five properties are described below:

P_1 It is always possible to return to a previously active state.
P_2 A bow cannot be bent if there is no arrow on it. (see Fig. 6 at the top left; the global variable *nbarrow* is represented by integer id 1).
P_3 All states of a model can be reached from any state.
P_4 Whenever the bow is fired, the amount of available arrows should decrement (see Fig. 6 at the top right; the global variable *nbarrow* is represented by integer id 1).
P_5 After firing an arrow, one should eventually be able to fire another one.

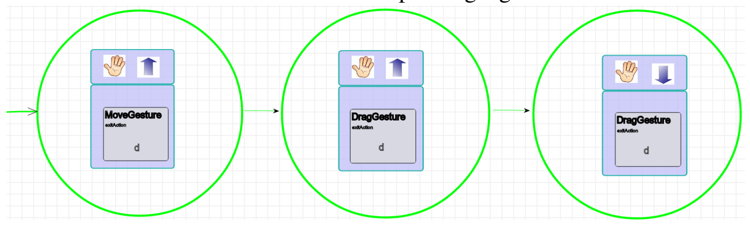
The above properties are transformed to LTL, and are inserted in Promela code consisting of the initial state of the system, the environment and the rule-based operational



Generated metamodel of output language $GISMO_O$



Generated metamodel of input language $GISMO_I$



Example of an instance model of $GISMO_I$

Fig. 4. Generated metamodels $GISMO_O$ and $GISMO_I$, and instance model of $GISMO_I$.

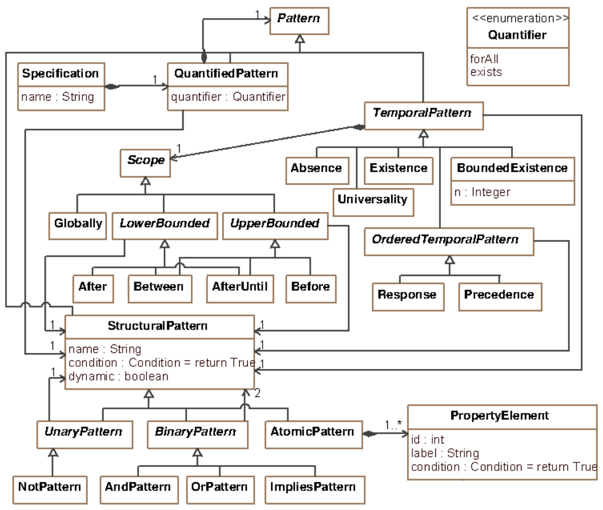


Fig. 5. The generic metamodel template used for generating the property language.

semantics as shown in step 1 of Fig. 3. In step 2, SPIN verifies whether the system satisfies the formula, returning "True" if it does. If there is a counter-example, steps 3 to 5 are followed: the counter-example trace is played back by SPIN, and a readable trace is printed (step 3), this trace is converted automatically to the counter-example output model (step 4), and this counter-example can be played out state by state by showing a runtime model for each state (step 5). The properties P_2 and P_4 yield a counter-example, one of which is shown in Fig. 6 at the bottom. The trace represents a sequence of five states, leading to the undesirable state (the last one, where the state is Bending, but the nbarrow variable has value 0). Each state can be mapped to Fig. 2, and the current state is highlighted, as well as the current input gesture. Because of these counter-examples, we were able to find and fix an error in our bow model of Fig. 2, namely that picking up an arrow (represented by the MoveGesture to the left of sheated) is not required. In another instance, we were able to find and correct an error in one of the operational semantics' rules.

In comparison to [3], we made some changes that influence the performance, such as splitting up quantified rules into several LTL formulae (e.g., Prop. 3). The performance in terms of time and memory consumption is good: evaluation never takes more than a second, and never requires more than 100 MB of memory. Also, the search tree depth never exceeds 4000, and the number of states that are visited stays well under 20000.

6 Related Work

In the last decade, a plethora of language-specific approaches have been presented to specify and verify properties for different kinds of design-oriented languages. A subset of these approaches verify component-based systems [9], concurrent systems [10] and architectural models expressed in UML [11]. Gabmayer *et al.* survey approaches

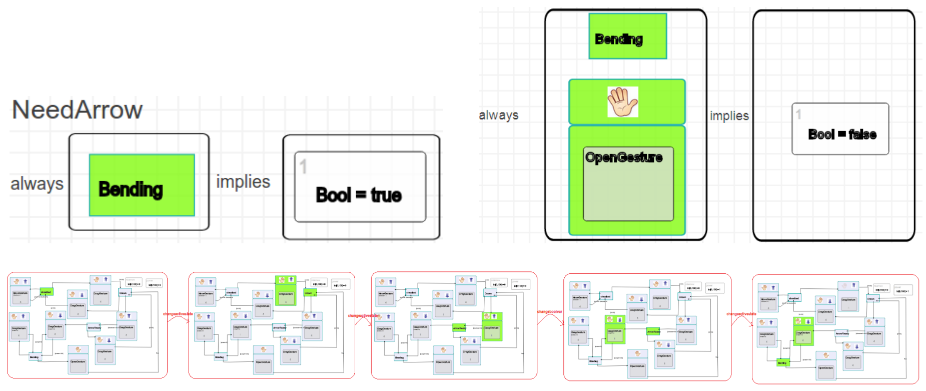


Fig. 6. The two properties P_2 (top left) and P_4 (top right) that yield a counter-example, and the counter-example of P_2 (bottom).

aiming at specifying and verifying temporal model properties by model checkers [12]. These approaches offer either language-specific property languages, or LTL properties have to be defined directly on the formal representation, thus not aiming at supporting DSMLs engineers in the task of building domain-specific property languages.

Generic solutions shift the specification and verification tasks to the model level in a more generalized manner. Some approaches propose OCL extensions for defining temporal properties (TOCL) on models [13, 14]. Combemale *et al.* propose a pattern-based extension to modelling languages aiming at supporting temporal property verification using TOCL, while producing model-based input and output automatically for model checking purposes [15]. Klein *et al.* [16] present an approach to define properties at the model level in a generic way, by extending a language for specifying structural patterns based on Story Diagrams with support for specifying temporal patterns.

7 Conclusion and Future Work

This article reported on a practical application of the *ProMoBox* approach [3] for verifying temporal properties on DSMLs. A small number of models is required as input to specify properties and transform them to SPIN, verify them and visualise possible counter-examples, while the user is shielded from the underlying formal model checking intricacies. *GISMO*, a DSML for specifying executable models of gestural interaction applications, was used as a case study. We illustrated how verifying properties on *GISMO* models can be realised with *ProMoBox*, after applying a series of necessary simplifications on the *GISMO* metamodel to ensure that the model is bounded and to avoid a combinatorial explosion of the model checking. We conclude that applying *ProMoBox* to *GISMO* is feasible. Annotating the *GISMO* metamodel to enable the automatic generation of a property language is straightforward. Simplifying the metamodel to enable the model checker to verify properties in a reasonable time took some more effort and reflection. Nevertheless, this is a common step when applying model checking. Lifting the power of model checking to the level of domain-specific models is possible, and the expressiveness of the properties is only limited by the template metamodels of *ProMoBox*, that can be easily extended to include all the concepts of model-checking formalisms such as LTL and Promela.

In future work, we are interested in broadening the types of languages that are supported by *ProMoBox*, e.g. languages and properties that explicitly include time. Also, we will investigate alternative approaches to model checking, such as the generation and execution of test cases, so that the approach becomes more scalable.

References

1. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering. FOSE '07, Washington, DC, USA, IEEE Computer Society (2007) 37–54
2. Risoldi, M.: A methodology for the development of complex domain-specific languages. PhD thesis, University of Geneva (2010)
3. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Wimmer, M., Vangheluwe, H.: ProMoBox: A framework for generating domain-specific property languages. In: Int'l Conf. Software Language Engineering (SLE). (2014)
4. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Mierlo, S.V., Ergin, H.: AToMPM: A Web-based Modeling Environment. In: MoDELS Demonstrations. (2013) 21–25
5. Holzmann, G.J.: The model checker spin. IEEE Trans. Softw. Eng. **23** (1997) 279–295
6. Deshayes, R.: A domain-specific modeling approach for gestural interaction. In: Visual Languages and Human-Centric Computing (VL/HCC). (2013) 181–182
7. Deshayes, R., Mens, T., Palanque, P.: A generic framework for executable gestural interaction models. In: Visual Languages / Human Centric Computing. (2013) 35–38
8. Navarre, D., Palanque, P., Ladry, J.F., Barboni, E.: ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. ACM Trans. Comput.-Hum. Interact. **16** (2009) 18:1–18:56
9. Cimatti, A., Mover, S., Tonetta, S.: Proving and explaining the unfeasibility of message sequence charts for hybrid systems. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design. FMCAD '11 (2011) 54–62
10. Li, X., Hu, J., Bu, L., Zhao, J., Zheng, G.: Consistency checking of concurrent models for scenario-based specifications. In: SDL 2005: Model Driven. Volume 3530 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 298–312
11. Pelliccione, P., Inverardi, P., Muccini, H.: Charmy: A framework for designing and verifying architectural specifications. IEEE Trans. Softw. Eng. **35** (2009) 325–346
12. Gabmeyer, S., Kaufmann, P., Seidl, M.: A classification of model checking-based verification approaches for software models. In: Proceedings of the STAF Workshop on Verification of Model Transformations (VOLT 2013). (2013) 1–7
13. Ziemann, P., Gogolla, M.: Ocl extended with temporal logic. In Broy, M., Zamulin, A., eds.: Perspectives of System Informatics. Volume 2890 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2003) 351–357
14. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: OCL meets CTL: Towards CTL-Extended OCL Model Checking. In: Proceedings of the MODELS 2013 OCL Workshop. Volume 1092 of CEUR Workshop Proceedings. (2013) 13–22
15. Combemale, B., Crégut, X., Pantel, M.: A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In: Asia-Pacific Softw. Eng. Conf. (APSEC). (2012) 282–287
16. Klein, F., Giese, H.: Joint structural and temporal property specification using timed story scenario diagrams. In: Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering. FASE'07 (2007) 185–199

On Synergies between Model Transformations and Semantic Web Technologies

Robert Bill[1], Simon Steyskal[1,2], Manuel Wimmer[1], and Gerti Kappel[1]

[1] Vienna University of Technology, Austria
[lastname]@big.tuwien.ac.at
[2] Siemens AG Österreich, Siemensstrasse 90, 1210 Vienna, Austria

Abstract. The integration of heterogeneous data is a reoccurring problem in different technical spaces. With the rise of model-driven engineering (MDE), much effort has been spent in developing dedicated transformation languages and accompanying engines to transform, compare, and synchronize heterogeneous models. At the same time, ontologies have been proposed in the Semantic Web area as the main mean to describe the intension as well as the extension of a domain. While dedicated languages for querying and reasoning with ontologies have been intensively studied, specific support for integration concerns leading to executable transformations is rare compared to MDE.
Based on previous studies which relate metamodels and models to ontologies, we discuss in this paper synergies between transformation languages of MDE, in particular Triple Graph Grammars (TGGs), and Semantic Web technologies (SWTs), namely OWL/SPARQL. First, we show how TGGs are employed to define correspondences between ontologies and how these correspondences are expressed in SPARQL. Second, we show how reasoning support of SWTs is applied to allow for underspecified model transformation specifications as well as how the different assumptions on existing knowledge effect transformations. We demonstrate these aspects by a common case study.

Keywords: Model Transformation, Model Integration, Triple Graph Grammars, OWL, SPARQL

1 Introduction

The integration of heterogeneous data has first emerged in the database area [26]. However, data integration is a reoccurring problem, not only in the database area, but in different technical spaces [8, 11]. With the raise of model-driven engineering (MDE), much effort has been spent in developing dedicated transformation languages and accompanying engines to transform, compare, and synchronize heterogeneous models.

At the same time, ontologies have been proposed in Semantic Web to describe the intension as well as the extension of a domain. While dedicated languages for querying and reasoning with ontologies have been intensively studied (e.g., classification of individuals and consistency checking are provided by standard reasoner), specific support for integration concerns leading to executable transformations is rare.

In order to understand the differences and commonalities between MDE and Semantic Web technologies (SWTs), several studies have investigated about the languages used in both fields to describe the domain of discourse [28]. Thus, bridges are already available between these two worlds for transforming metamodels and corresponding models to ontologies and vice versa. Some studies go also beyond purely structural information [12], but bridges concerning dynamic information are still mostly unexplored. Moreover, for specific domains, such as configuration management [4], both technologies are applied, but mostly in an isolated manner as we currently explore in a recent project[3].

Based on previous studies which relate metamodels and models to ontologies [5,10], we discuss in this paper synergies between transformation languages of MDE, in particular Triple Graph Grammars (TGGs) [24], and SWTs, namely a combination of OWL/SPARQL. First, we show how TGGs are employed to define correspondences between ontologies visualized as metamodels and how these correspondences are operationalized by a compilation of TGGs to OWL/SPARQL. Second, we show how reasoning support of SWTs is applicable to allow for underspecified model transformation specifications, i.e., the concrete types of instances are assigned in a post-processing step using OWL reasoner. Third, we discuss how switching between the closed world assumption (CWA) to an open world assumption (OWA) is beneficial for particular integration scenarios where only partial knowledge of existing models is present. We demonstrate these aspects by a common case study.

The rest of this paper is structured as follows. In the next section, we introduce the running example for this paper as well as the technological prerequisites. In Section 3, we discuss the mapping between TGGs and OWL/SPARQL in a general form, whereas in Section 4 we demonstrate the compilation of TGGs to OWL/SPARQL by-example and discuss how features of ontologies may be exploited for model transformations. In Section 5 we discuss related work before we conclude in Section 6.

2 Preliminaries

2.1 Motivating Example

As an example we will illustrate how heterogeneous views on computer networks can be joined (cf. Fig. 1). The first view comprises the physical network structure including cables of various types and speed as well as computers. The second view contains the application structure of the network, i.e., various computers are running different services which might require each other. In that case, a connection to a matching service is required. Connections can be modeled by their physical structure, as well as their logical network structure. By using TGGs we are able to define correspondences between both structures, which can be used to either (i) express graph transformation rules to transform individuals from one schema to another or (ii) check whether or not such alignments hold for given models. Extending those correspondence definitions with SWTs, allows even more sophisticated reasoning, inferencing, and querying tasks.

[3] `http://cosimo.big.tuwien.ac.at`

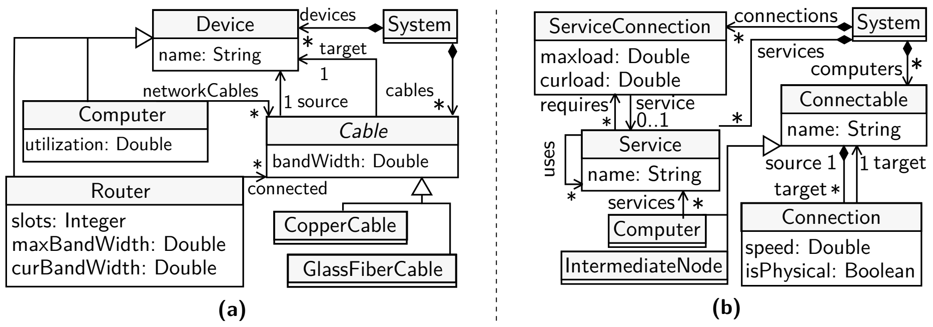


Fig. 1. Representations of networks: (a) physical and (b) logical

2.2 Triple Graph Grammars (TGGs)

TGGs have first been introduced by Andy Schürr [24]. A TGG rule combines elements from a left model (LM), a right model (RM) and a correspondence model (CM). Each TGG rule contains a left hand side graph (LG) conforming to LM, a right hand side graph (RG) conforming to RM and a correspondence graph (CG) conforming to CM which connects elements from LG and RG. Vertices and edges may not be deleted by any rule, they can only be preserved or created. In contrast to usual graph transformation rules, TGG rules are inherently bidirectional. A TGG engine searches for rule applications creating the required input graphs and creates elements of all other graphs during that process. For example, in a transformation scenario an input graph for LM would result in a graph for CM and RM. TGG rules might also have additional constraints, e.g., negative application conditions or attribute constraints, also restricting the value of an attribute depending on attribute values of other objects in the TGG rule.

The left-hand side of Fig. 5 shows a simple example of a TGG rule of a computer network that relates connections of the same speed without declaring them equal. Black elements denote elements which have been matched already, green elements are elements which are matched or created then. In this case, the difference between both models is only a syntactical one.

2.3 Semantic Web Technologies (SWTs)

RDF & OWL. The Resource Description Framework (RDF)[4] is a framework to describe and represent information about resources and is both human-readable and machine-processable, which enables the possibility to easily exchange information among different applications using RDF triples.

In RDF everything is a resource, uniquely identified by its URI and all data is represented as $(subject, predicate, object)$ triples, where subjects and predicates are URIs and objects can either be literals (strings, integers, . . .) or URIs.

Since RDF itself does not contain sophisticated semantics to express characteristics of concepts or to define more expressive relationships among them, the Web Ontology

[4] `http://www.w3.org/TR/rdf-mt/`

Language (OWL)[5] was developed. Introducing OWL allows the usage of reasoning systems such as Pellet [27], FaCT++ [29] or HermiT [25] to (i) infer new knowledge and (ii) detect inconsistencies based on the modeled semantics.

SPARQL. The Protocol And RDF Query Language (SPARQL) is basically the standard query language for RDF[6]. Its syntax (cf. Figs. 2-4) is highly influenced by an RDF serialization format called Turtle [1] and SQL. In its current version, SPARQL allows besides basic query operations such as union of queries, filtering, sorting and ordering of results as well as optional query parts, the use of aggregate functions (`SUM, AVG, MIN, MAX, COUNT, ...`), the possibility to use subqueries, perform update actions via SPARQL Update and several other requested features as indicated in [18].[7]

Another important feature of SPARQL are CONSTRUCT queries, which allow the construction of new RDF graphs based on a previously matched (against one ore more input graphs) SPARQL graph pattern (cf. Fig. 3). Their main purpose lies in data integration scenarios, where data from one ore more data sources have to be normalized to fit a common schema [19, 23].

```
SELECT ?a ?name
WHERE {
  ?a a :Device .
  ?a :name ?name .
}
```

Fig. 2. SELECT Query

```
CONSTRUCT {
  ?a :hasName true .
}
WHERE {
  ?a a :Device .
  ?a :name ?name .
}
```

Fig. 3. CONSTRUCT Query

```
ASK WHERE {
  ?a a :Device .
  ?a :name ?name .
}
```

Fig. 4. ASK Query

3 Aligning TGGs and SWTs

In the following, we map basic TGG rules to SPARQL queries. The TGG rule definition is based on [2] and extended by labels. Advanced features of graph transformations like multi–nodes are not supported. Subsequently, we discuss two benefits of using SWTs: (i) automatic type inference and (ii) reasoning under CWA and OWA.

Definition 1 (E–Graph)**.** A labeled E–Graph $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (src_j, trg_j)_{j\in\{G,EA,EA\}}, l)$ consists of graph vertices V_G, data vertices V_D, graph edges, node attribute edges and edge attribute edges E_G, E_{NA}, E_{EA}, source and target edge functions src_j and trg_j mapping edges to corresponding vertices and a labeling function l defining a label for each vertex and edge. As shorthand we use $src(e)$ and $trg(e)$ to denote source and target edge functions operating on edges of any kind.

Definition 2 (Typegraph)**.** A typegraph TG is an E-Graph specifying the relation between types. A type morphism t maps nodes and edges of a graph to the corresponding nodes and edges in the type graph.

[5] `http://www.w3.org/TR/owl2-overview/`
[6] `http://www.w3.org/TR/sparql11-overview/`
[7] For a comprehensive overview on the semantics of SPARQL queries see [17, 22].

Definition 3 (Triple graph)**.** A triple graph $TRG = (LG \xleftarrow{m_s} CG \xrightarrow{m_t} RG)$ consists of three E–Graphs LG, CG and TG and morphisms m_s and m_t mapping corresponding nodes from the correspondence graph to other graphs.

Definition 4 (TGG rule)**.** A parametrized TGG rule $TGG = (SG, ZG, ac)$ consists of a source triple graph SG, a target triple graph $ZG \supset SG$, and application conditions ac. For the sake of simplicity, we consider application conditions as boolean formulas using $\wedge$ and $\vee$, over atomic positive application conditions (PACs) and negative application conditions (NACs) defined as triple graphs which might share vertices with each other and SG and TG.

Example 1. The TGG rule $r2n = (tr_s, tr_t, \text{true})$ represented in Fig. 5 could be specified as follows: The type graph of both models is derived from the metamodel. For example, a subset of the typegraph for `network1` could be specified as $G_{n1} = (\{d_{n1}, c_{n1}\}, \{double\}, \{s_{n1}, t_{n1}\}, \{b_{n1}\}, \{\}, \{(s_{n1}, c_{n1}), (t_{n1}, c_{n1})\}, \{(s_{n1}, d_{n1}), (t_{n1}, d_{n1})\}), \{(b_{n1}, c_{n1})\}, \{(b_{n1}, double)\}, \{\}, \{\})$ combined with a standard double DSIG algebra.

The source graph of TGG rule $tr_s = (LG_s \xleftarrow{m_{ss}} CG_s \xrightarrow{m_{ts}} TG_s)$ with $LG_s = (\{d_1, d_2\}, \emptyset, \ldots, \emptyset)$, $CG_s = (\{g2c_1, g2c_2\}, \emptyset, \ldots, \emptyset)$, $TG_s = (\{cb_1, cb_2\}, \emptyset, \ldots, \{\})$, $m_{ss} = \emptyset$, $m_{ts} = \emptyset$. The target graph $tr_t = (LG_t \xleftarrow{m_{st}} CG_t \xrightarrow{m_{tt}} TG_t)$ with $LG_t = LG_s \cup (\{c_1\}, \{n\}, \{s_1, t_1\}, \{b_1\}, \emptyset, \{(s_1, c_1), (t_1, c_1)\}, \{(s_1, d_1), (t_1, d_2)\}, \{(b, c_1)\}, \{(b, n)\}, \emptyset, \emptyset)$, $CG_t = CG_s \cup (\{\}, \emptyset, \ldots, \emptyset)$, $RG_t = RG_s \cup (\{c_1\}, \{n\}, \{s_1, t_1\}, \{b_1\}, \emptyset, \{(s_1, c_1), (t_1, c_1)\}, \{(s_1, cb_1), (t_1, cb_2)\}, \{(sp, c_1)\}, \{(sp, n)\}, \emptyset, \emptyset)$. The labeling function assigns the type names from the metamodel in Fig. 1 to elements of the type graph and lables equal to variable name to elements of other graphs.

Mapping TGG rules to SPARQL queries. Both transformation and correspondence construction can be expressed using SPARQL queries in the merged ontology using the labeling function. If a model should be synchronized, at first the maximum correspondence is searched, then for the unmatched elements a transformation might be conducted. Each vertex and edge in a TGG rule can be used as context, matched or created element, depending on the use of the TGG rule. In every case, the context nodes are exactly those occurring in the source model, but not in the target one. In a transformation, elements of source model of the transformation in the target rule, but not in the source, are used as matched elements while the others are used as created elements. In a corresponding search scenario, only the elements of the correspondence graph are created, all others are matched. According to [7], an attribute `hasMatch` is introduced to specify which elements have been matched already. For elements, it is initially unset and may be set to `true`. For edges, it has the same domain and range as the edge to match, but `hasMatch_` as prefix to the original name. Thus, it is not necessary to modify the ontology prior to using the converted TGG rules.

In many cases, TGG rules as defined previously can be transformed to corresponding SPARQL queries. Table 1 shows matching concepts. The general structure of a generated SPARQL query is `CONSTRUCT <transformed created elements> WHERE <context elements>`. The first two lines, e.g., indicate that the occurrence of a vertex n labeled *c1* with a type labeled *computer* in the TGG rule should

P.	**Type**	**TGG**	**SPARQL**
`W`	c,m	Vertex n	`?`$l(n)$ `a` $l(t(n))$`.`
`W`	c,m	Vertex n_1, n_2, $n_1 \neq n_2$, injective matching	`FILTER (?`$l(n_1)$ `!= ?`$l(n_2)$`).`
`C`	cr	Vertex n	`?`$l(n)$ `a` $l(t(n))$`.`
`W`	c,m	Edge e	`?`$l(s(e))$ $dom(e)$`:`$l(e)$ `?`$l(t(n))$.
`C`	cr	Edge e	`?`$l(s(e))$ $dom(e)$`:`$l(e)$ `?`$l(t(n))$.
`C`	m,cr	Vertex n	`?`$l(n)$ `tgg:hasMatch true.`
`C`	m,cr	Edge e	`?`$l(s(e))$ `tgg:hasMatch_`$dom(e)$`_`$l(e)$ `?`$l(t(e))$`.`
`W`	m	Vertex n	`FILTER NOT EXISTS {?`$l(n)$ `tgg:hasMatch true.}`
`W`	m	Edge e	`FILTER NOT EXISTS {?`$l(s(e))$ `tgg:hasMatch_`$dom(e)$`_`$l(e)$ `?`$l(t(e))$`}`
`W`	c	Vertex n	`FILTER EXISTS {?`$l(n)$ `tgg:hasMatch true.}`
`W`	c	Edge e	`FILTER EXISTS {?`$l(s(e))$ `tgg:hasMatch_`$dom(e)$`_`$l(e)$ `?`$l(t(e))$`}`
`C`	m,cr	Mapping $e_1 \mapsto e_2$ from m_{ss}, m_{st}, tt or ts	`?`$l(e_1)$ `owl:sameAs ?`$l(e_2)$
`C`	m,cr	Mapping $e_1 \mapsto e_2$ from m_{ss}, m_{st}, tt or ts	`?`$l(e_1)$ `owl:sameAs ?`$l(e_2)$
`W`	-	Atomic PAC ac	`BIND (EXISTS {<expand` ac `acc. to Tab. 1>}) AS ?`$gn(ac)$
`W`	-	Atomic NAC ac	`BIND (NOT EXISTS {<expand` ac `acc. to Tab. 1>}) AS ?`$gn(ac)$
`W`	-	Full AC $ac = ac_1(\wedge\|\vee)ac_2$	`FILTER (?`$gn(ac_1)$ `(&&/||) ?`$gn(ac_2)$`)` or rather applied recursively until the atomic operations.

Table 1. SPARQL patterns occuring in the `WHERE` (W) and `CONSTRUCT` (C) part for context (c), matching (m) and created (cr) elements

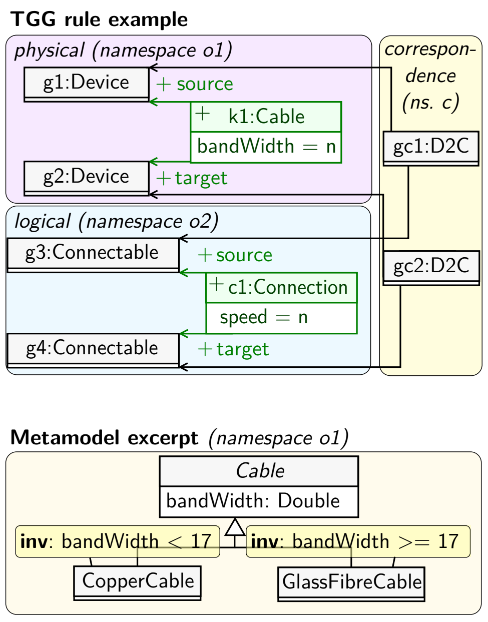
result in `c1 a computer.`, which is placed in the `CONSTRUCT` part of the SPARQL query for context nodes and nodes to be matched and in the `WHERE` part for created nodes.

The helper function dom returns the graph for an element. It returns `o1` for elements of the left graph, `o2` for elements of the right graph and `c` for elements of the correspondence graph. The helper function gn assigns a unique name to each atomic application condition.

Many current TGG implementations allow the use of functions to set values for attributes. Existing approaches for converting OCL into SPARQL, could be used for functions requiring matched nodes, context nodes and, for created nodes, (other) created nodes. The result then can be assigned using `BIND`. Some constraints on created nodes might be formulated using SWRL expressions, e.g., the subset of OCL defined in [12]. In this case, the specific match of vertices in a TGG rule has to be stored to be able to subsequently apply the constraint. Thus, attributes `:hasSwrlRule_`$name$`_`$index$ might be set true to specify that an element is used as element nr. $index$ in the SWRL rule $name$. In such a way, not only conditions in the TGG can be formulated, but also invariants of each individual model or the merged model. Simple invariants may also be modeled directly in OWL. In the following, we will show an example to illustrate how this may help reducing the complexity of TGG rules significantly.

Automatic Type Inference. Considering the two views on an imaginary network, one might distinguish the creation of `Coppercables` and `Glassfibrecables` based on the `speed` of a correlating `Connection` between two previously aligned `Devices` and `Connectables` (e.g. creating a `Glassfibrecable` if the `speed` exceeds a certain threshold or a `Coppercable` otherwise). Although such distinctions can be modeled with TGGs, at least two TGG rules (in our case; one matching `Coppercable` and one matching `Glassfibrecable`) would be necessary.

A more convenient way to model such a behavior can be achieved by *out-sourcing* the type inference to OWL reasoners and only define one TGG rule, which describes the more general `Cable` and `Connection` correspondence as depicted in Figure 5 (with its corresponding SPARQL CONSTRUCT query). The constraints itself (i.e., defining the concept `Glassfibrecable` to be equivalent to an anonymous concept which is defined as `Cable` having a `speed` with a value over 17) can be directly modeled within the ontology using OWL axioms[8] and were generated during the initial model to ontology transformation .



```
CONSTRUCT {
 ?k1 a o1:Cable .
 ?k1 o1:source ?g1 .
 ?k1 o1:target ?g2 .
 ?k1 o1:bandWidth ?n .
 ?k1 tgg:hasMatch true .
 ?c1 tgg:hasMatch true .
} WHERE {
 ?gc1 a c:D2C. ?gc2 a c:D2C.
 ?g1 owl:sameAs ?gc1 .
 ?g2 owl:sameAs ?gc2 .
 ?g3 owl:sameAs ?gc1 .
 ?g4 owl:sameAs ?gc2 .
 ?c1 o2:source ?g3 .
 ?c1 o2:target ?g4 .
 ?c1 o2:speed ?n .
 FILTER (?gc1 != ?gc2) .
 FILTER (?g1 != ?g2) .
 FILTER (?g3 != ?g4) .
 FILTER EXISTS
   {?gc1 tgg:hasMatch true} .
 FILTER EXISTS
   {?gc2 tgg:hasMatch true} .
 FILTER NOT EXISTS
   {?c1 tgg:hasMatch true} .
}
```

Fig. 5. CONSTRUCT query which generates `Cables` for given `Connections`

Reasoning under Open and Closed World Assumption. One of the major benefits of SWTs for integration scenarios are their well defined semantics and the extensive reasoner support as already discussed previously. With OWL and OWL reasoners it is

[8] cf. [6] for a comprehensive list of OWL axioms

e.g., possible to describe cardinality constraints, perform automatic type inferencing as discussed above and to check for inconsistency in the given models.

While Semantic Web languages are based on OWA (i.e., if a statement is not explicitly stated, it does not mean that it does not exist), software engineering languages are mostly based on CWA (i.e., if a statement is not present, it does not exist) [20]. To deal with this issue, we translate parts[9] of the constraints expressed as OWL axioms into SPARQL queries and query for the presence of individuals which violate those constraints. E.g., consider the cardinality constraint `computers exactly 2 Computer` for concept `System` expressed in *OWL Manchester Syntax*[10]. The answer to the question whether or not a particular `System` has the right amount of `Computers`, would not be directly decidable for the OWA but for the CWA with the support of SPARQL as depicted in Listing 1.

```
ASK WHERE {
 { SELECT (count(?b) AS ?number) ?a WHERE {
     ?a a :System .
     ?a :computers ?b . } GROUP BY ?a }
 FILTER(?number != 2)}
```

Listing 1. ASK Query which returns `true` if a `System` has not exactly 2 `Computers`

4 Related Work

We discuss three lines of related work: (i) approaches for bridging models and ontologies, (ii) approaches for transforming transformations to SWTs, and (iii), approaches directly using SWTs to encode model transformations.

Bridging models and ontologies. Combining modeling approaches steaming from MDE with ontologies has been studied in the last decade [5]. There are several approaches to transform Ecore-based models to OWL and back, e.g., cf. [9, 31]. In addition, there exist approaches that allow for the definition of ontologies in software modeling languages such as UML by using dedicated profiles [13]. Moreover, there are also approaches which combine the benefits of models and ontologies such as done in [14, 16]. Not only the purely structural part of UML is considered, but some works also target the translations of constraints between these two technical spaces by using an intermediate format [3]. We build on these mentioned approaches, but we focus on correspondence definitions and their execution as transformations.

Transforming transformations to SWTs. Concerning the definition and execution of model transformations based on SWTs, we are aware of two approaches. First, [15] propose the usage of an ATL-inspired language for defining mappings between ontologies. Thus, uni-directional transformations are implementable for ontologies as it is known from model transformations. Another approach is presented in [30] which translates parts of ATL transformations to ontologies for checking the consistency of transformation rules, e.g., overlaps between rules in terms of overlapping matches. In our work, we follow this line of research, but we consider bi-directional transformations specified in TGGs. Thus, in our translations to ontologies we have to consider not only source to

[9] The decision, which constraints have to be translated, highly depends on the respective integration scenario.

[10] `http://www.w3.org/TR/owl2-manchester-syntax/`

target transformations, but we have to encode comparison and synchronization transformations as well in SPARQL.

Specifying transformations with SWTs. Finally, there are approaches which shift the definition of the model transformations to the SWTs. For instance, in [21] it is proposed to use SWRL to define the correspondences between models to allow for model synchronization. In [10], ontology matching tools are applied to search for correspondences between metamodels and to derive from these correspondences model transformations. In the context of this work, we have the assumption that correspondences are defined based on models using TGGs, but at the same time we explored which benefits from ontology reasoning may be transferred to model transformation approaches.

5 Conclusion and Further Work

In this paper we have outlined an initial mapping between TGGs and OWL/SPARQL. Especially, new features of the latest SPARQL version helped in defining a comprehensive mapping between these languages. Moreover, we also explored how reasoning capabilities can be leveraged for underspecified model transformations.

While the initial results of applying our approach seem promising, both from a mapping point of view and usage of reasoning capabilities for model transformations, further investigation are planned such as considering a mapping between TGGs and SWRL. Empirical studies are planned as well in the area of configuration management together with our industry partner Siemens AG Österreich. In particular, for performing distributed configuration management [4] where several different models and reasoners have to be connected, we plan to apply our approach to provide the necessary integration means.

Acknowledgment: This work has been funded by the Vienna Business Agency (Austria), in the programme ZIT13 plus, within the project COSIMO (Collaborative Configuration Systems Integration and Modeling) under grant number 967327.

References

1. David Beckett and Tim Berners-Lee. Turtle-terse RDF triple language. *W3C Team Submission*, 14, 2008.
2. Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376(3):139–163, 2007.
3. Dragan Djuric, Dragan Gasevic, Vladan Devedzic, and Violeta Damjanovic. A UML Profile for OWL Ontologies. In *Proc. of MDAFA*, pages 204–219, 2004.
4. Andreas A. Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner. Modeling and solving technical product configuration problems. *AI EDAM*, 25(2):115–129, 2011.
5. Dragan Gasevic, Dragan Djuric, and Vladan Devedzic. *Model Driven Engineering and Ontology Development (2. ed.)*. Springer, 2009.
6. OWL Working Group. OWL 2 Web Ontology Language. *W3C recommendation*, 2012.
7. Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *SoSyM*, pages 1–29, 2013.
8. Zhenjiang Hu, Andy Schürr, Perdita Stevens, and James F. Terwilliger. Dagstuhl seminar on bidirectional transformations (bx). *SIGMOD Record*, 40(1):35–39, 2011.

9. Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In *Proc. of MODELS*, pages 528–542, 2006.
10. Gerti Kappel, Horst Kargl, Gerhard Kramler, Andrea Schauerhuber, Martina Seidl, Michael Strommer, and Manuel Wimmer. Matching metamodels with semantic systems - an experience report. In *Proc. of BTW Workshops*, pages 38–52, 2007.
11. Ivan Kurtev, Mehmet Aksit, and Jean Bézivin. Technical Spaces: An Initial Appraisal. In *Proc. of CoopIS*, 2002.
12. Sergey Lukichev. Defining a subset of OCL for expressing SWRL rules. In *RuleApps*, pages 1–3, 2008.
13. Milan Milanovic, Dragan Gasevic, Adrian Giurca, Gerd Wagner, and Vladan Devedzic. Towards Sharing Rules Between OWL/SWRL and UML/OCL. *ECEASST*, 5, 2006.
14. Fernando Silva Parreiras and Steffen Staab. Using ontologies with UML class-based modeling: The TwoUse approach. *Data Knowl. Eng.*, 69(11):1194–1207, 2010.
15. Fernando Silva Parreiras, Steffen Staab, Simon Schenk, and Andreas Winter. Model driven specification of ontology translations. In *Proc. of ER*, pages 484–497, 2008.
16. Fernando Silva Parreiras, Steffen Staab, and Andreas Winter. On marrying ontological and metamodeling technical spaces. In *Proc. of FSE*, pages 439–448, 2007.
17. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In *Proc. of ISWC*, pages 30–43, 2006.
18. Axel Polleres. SPARQL1. 1: New features and friends (OWL2, RIF). In *Web Reasoning and Rule Systems*, pages 23–26. Springer, 2010.
19. Axel Polleres, François Scharffe, and Roman Schindlauer. SPARQL++ for mapping between RDF vocabularies. In *Proc. of OTM*, pages 878–896, 2007.
20. Tirdad Rahmani, Daniel Oberle, and Marco Dahms. An adjustable transformation from OWL to Ecore. In *Proc. of MODELS*, pages 243–257, 2010.
21. Federico Rieckhof, Mirko Seifert, and Uwe Aßmann. Ontology-based model synchronisation. In *Proc. of TWOMDE Workshop*, 2010.
22. Simon Schenk. A sparql semantics based on datalog. In *Proc. of KI*, pages 160–174, 2007.
23. Simon Schenk and Steffen Staab. Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In *Proc. of WWW*, pages 585–594, 2008.
24. Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proc. of WG Workshop*, pages 151–163, 1994.
25. Rob Shearer, Boris Motik, and Ian Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In *Proc. of OWLED Workshop*, 2008.
26. Nan C. Shu, Barron C. Housel, Robert W. Taylor, Sakti P. Ghosh, and Vincent Y. Lum. EXPRESS: A Data EXtraction, Processing, amd REStructuring System. *ACM Trans. Database Syst.*, 2(2):134–174, 1977.
27. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.
28. Steffen Staab, Tobias Walter, Gerd Gröner, and Fernando Silva Parreiras. Model driven engineering with ontology technologies. In *Proc. of Reasoning Web*, pages 62–98, 2010.
29. Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In *Automated reasoning*, pages 292–297, 2006.
30. Dennis Wagelaar. Towards using OWL DL as a metamodelling framework for ATL. In *Proc. of MtATL Workshop*, pages 79–85, 2010.
31. Tobias Walter, Fernando Silva Parreiras, Gerd Gröner, and Christian Wende. OWLizing: Transforming Software Models to Ontologies. In *Proc. of ODiSE*, pages 7:1–7:6, 2010.

Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure

Miklós Maróti[2], Tamás Kecskés[1], Róbert Kereskényi[1], Brian Broll[1], Péter Völgyesi[1], László Jurácz, Tihamér Levendoszky[1], and Ákos Lédeczi[1]

[1] Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA
akos.ledeczi@vanderbilt.edu
[2] Bolyai Institute, University of Szeged, Szeged, Hungary
mmaroti@math.u-szeged.hu

Abstract. The paper presents WebGME, a novel, web- and cloud-based, collaborative, scalable (meta)modeling tool that supports the design of Domain Specific Modeling Languages (DSML) and the creation of corresponding domain models. The unique prototypical inheritance, originally introduced by GME, is extended in WebGME to fuse metamodeling with modeling. The tool also introduces novel ways to model crosscutting concerns. These concepts are especially useful for multi-paradigm modeling. The main design drivers for WebGME have been scalability, extensibility and version control. The web-based architecture and the constraints the browser-based environment introduces provided significant challenges that WebGME has overcome with balanced trade-offs. The paper describes the architecture of WebGME, argues why the major design decisions were taken and presents the novel features of the tool.

Keywords: DSML, metamodel, collaboration, web browser

1 Introduction

Applying Domain-Specific Modeling Languages (DSMLs) for the engineering of complex systems is becoming an increasingly accepted practice. Model Integrated Computing (MIC) is one approach that advocates the use of DSMLs and metamodeling tools [1]. The MIC open source toolsuite centered on the Generic Modeling Environment (GME) [2] has been applied successfully in a broad range of domains by Vanderbilt [3–7] and others [8–13]. (Note that this is just a selected subset of domains.) Design space exploration with sophisticated tool support [14] and the seamless integration of multiple third party software packages [15] are the most striking examples of the power of MIC.

However, the widespread application of MIC has uncovered the limitations of the tools. GME was designed as a desktop tool for the creation of small- to medium-sized models by a single user (or a small group of co-located users). The models are typically stored in a single file in a proprietary format. The elementary modeling concepts are somewhat arbitrary and are closely tied to their respective visualizations. The metamodels and the instance models are decoupled requiring a translation step, making DSML evolution cumbersome. As a result of the ever-increasing expectations, additional features have been added to the tools. For example, to address scalability concerns both in the size of the models and in the number of concurrent users, GME was extended with a Subversion-based backend to store the models in multiple XML files [16]. However, pessimistic locking to avoid incompatible changes to the model by multiple users proved to be inflexible and non-scalable. It became clear that these are fundamental limitations that need to be addressed at the very core of the architecture.

To address these limitations, we created WebGME, a web-based cyberinfrastructure to support the collaborative modeling, analysis, and synthesis of complex, large-scale information systems. The metamodels and the corresponding domain-specific models are tightly integrated via prototypical inheritance and stored in the cloud. Online collaboration, model version control, complex DSMLs and large instance models are transparently supported. Clients are web browser-based, resulting in platform independence and doing away with installation and upgrade issues. The user interface supports several built-in visualization techniques. Multiple APIs are provided to interface with existing external tools as well as to enable the development of custom domain-specific visualization components and code generation tools.

This paper presents the architecture and major design decisions of WebGME. Section 2 describes the meta-metamodel and the support for DSML specification. Section 3 describes the collaboration approach provided by WebGME. The data model is presented in Section 4, the overall architecture is described in Section 5, while model visualization support is summarized in Section 6. Section 7 illustrates how multi-paradigm modeling is supported by WebGME. The paper concludes with a brief overview of related work and conclusions.

2 Modeling Language Specification

The metamodel specifies the domain-specific modeling language. The metamodeling language consists of a set of elementary modeling concepts. These are the basic conceptual building blocks of any given approach and corresponding tools. It is the meta-metamodel that defines these fundamental concepts. These may include composition, inheritance, various associations, attributes and other concepts. Which ones to include, how to combine them, and what editing operations should operate on them and how are the most important design decisions that affect all aspects of the infrastructure and the domains that will use it.

Hierarchical decomposition is the most widely used technique to handle complexity. This is the fundamental organization principle in this tool, too. Copying, moving, or deleting a model will copy, move, or delete its constituent parts.

The single most important distinguishing feature of GME has been the unique use of prototypal inheritance. Each model at any point in the composition hierarchy is a prototype that can be derived to create an instance model. Derivation creates a copy of the model (and all of its parts recursively, i.e., a deep copy), but it establishes a dependency relationship between the corresponding objects. Any changes in the prototype automatically propagate to the instance. Of course, instances can be used again as prototypes for further specialization.

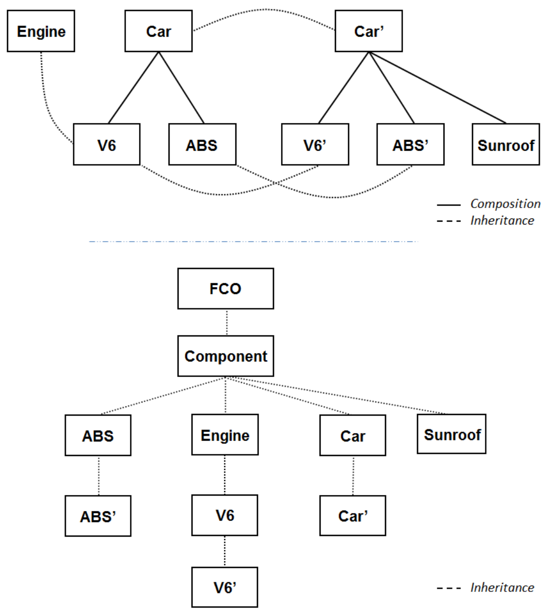


Fig. 1. Prototypical inheritance in WebGME

To illustrate this concept, consider Figure 1. Let's suppose that we have a DSML for modeling cars and the language has a concept called *Component* for modeling various parts of a car. The top half of the figure shows that we have created a simple *Car* model that has an *ABS* brake and a *V6* engine. Notice that an *Engine* model was also created and *V6* is derived from it. Note that all these models are derived from the *Component* model specified in the metamodel (not shown). Also note that solid lines represent composition, while dashed line show inheritance. Now if we take *Car* as a prototype and create a derived *Car'* instance model from it, you can see that it will contain a new instance of *ABS* and a new instance of *V6* as well. Note that we also added a *Sunroof* to this new car model. The corresponding inheritance tree is shown in the bottom half of the figure.

This approach is markedly different from inheritance in OO programming languages or in other modeling languages such as UML. First of all, it combines composition and inheritance. Note that Smalltalk and JavaScript have prototypical inheritance also, but it does not create new instances down the composition hierarchy. Second, inheritance is a "live" relationship between models that is continuously maintained during the modeling process. That is, any changes to a model propagate down the inheritance tree immediately. For example, if we change a property of the *Engine* model shown in Figure 1, it will also change in *V6* and *V6'*. On the other hand, if that property was already changed in *V6*, then the property modification in *Engine* will not have an effect on either *V6* or *V6'*. But if we reset the value in *V6* to the one inherited from *Engine*, that will propagate to *V6'* unless it has been overridden there beforehand. These rules also mean that when a model is deleted, so are all of its instances and instances of all of its children recursively. This variant of inheritance is a very powerful way to help the modeler handle the inherent complexity in large models and intricate DSMLs.

The novel idea in WebGME is to blur the line where metamodeling ends and domain modeling begins by utilizing inheritance to capture the metamodel/model relationship. Every model in a WebGME project is contained in a single inheritance hierarchy rooted at a model called FCO, for First Class Object, as shown in Figure 1. Metamodel information can be provided anywhere in this hierarchy. An instance of any model inherits all of the rules and constraints from its base (recursively all the way up to FCO) and it can further refine it by adding additional metamodeling information. This is a form of multi-level metamodeling with a theoretically infinite number of levels.

As a result of this approach, 1) metamodel changes propagate automatically to every model; 2) metamodels can be refined anywhere in the inheritance and composition hierarchies; 3) partially built domain models can become first class language elements to serve as building blocks; and 4) different (meta)model versions can peacefully coexist in the same project.

2.1 Meta-metamodel

A WebGME project is a collection of metamodels and models in a single composition hierarchy. Note that the words model and object will be used interchangeably throughout the rest of the paper. When the user creates a new project, it contains two objects called *Root* and *FCO*. *Root* is the root of the *composition* hierarchy, so every object in the project will be a node in the composition tree rooted at *Root*. *FCO* is contained by *Root* and it is the root of the inheritance hierarchy. Neither *Root* nor *FCO* can be deleted. Furthermore, the meta-metamodel of WebGME specifies that *Root* can contain an FCO and consequently, it can contain any other type of object. The meta rules of Root cannot be modified either. In addition, the initial meta rules of FCO are empty. The reason is that meta rules are inherited and they can only be extended and never restricted. For example, if we want a DSML where any model can contain any other kind of model, we can simply specify that *FCO* can contain *FCO*s. From then on, there cannot be any restrictions on composition in this DSML.

Additional relationships between objects can be expressed with *pointers* and *sets*. A pointer is a binary directed named association. Any object can have any number of different pointers. A pointer definition includes its name and a list of possible target objects. The latter restricts valid targets of the pointer to an element of the list, i.e., a model or any model derived from it. For example, if we want a pointer to be able to point to any other object in the project, we can specify the list of its valid targets to contain FCO.

A pair of pointers can be visualized as a connection. For example, the default WebGME editor takes any object with two pointers with the reserved names of *src* and *dst*, displays the object as a connection and supports the customary editing operations. Otherwise, connections are ordinary models; they can contain children, have other pointers and can be derived, etc. Therefore, the connection concept as such is not part of the meta-metamodel.

A set is a named association between one object and an unordered set of other objects. It can be considered a collection of pointers. A set is similar to

UML aggregation, but the objects the pointers can point to are not limited to be of the same kind. Any object can have any number of different sets. A set specification is exactly the same as that of a pointer.

An *attribute* defines a property of a model. The metamodel specifies the type of the attribute and a default value. Currently supported types are string, integer, float and enumeration, but these can be easily extended.

An *aspect* (also called *view*) is a subset of a model's children. Each model has a default aspect that contains all its children. Additional aspects help manage the complexity of models with many children. For example, a model of a car might have separate aspects for its mechanical, electrical and hydraulic design.

Prototypical *inheritance* is at the core of the WebGME meta-metamodel. Inherited children cannot be deleted, but new children can be added. Associations within the composition tree rooted at the base are adjusted to refer to the corresponding new instances. Associations pointing outside of the base model tree preserve their targets. New associations, attributes and aspects can be added. Association targets and attribute values in an instance can be modified. The goal of these rules is to be able to extend the inherited model, but prevent restricting it. New meta specification can be added to any object anywhere in the inheritance hierarchy. The rigid line between modeling and meta-modeling simply does not exist any more. A brief video tutorial (available at `http://webgme.org`) explains and demonstrates these concepts.

2.2 Cross-cutting concepts

Cross-cutting concepts are always difficult to model. In GME, the only way to capture relationships between models in different branches and/or levels of the composition hierarchy is through pointers and sets. However, the visual depiction of such associations is not intuitive at all since most tools display models according to composition, that is, they typically show the children of one model in one window (grandchildren may show up as ports). The target of a pointer can be indicated by its name and navigation to it can be supported, for example, by a double click operation, but an intuitive visual depiction of such relationships is sorely missing. For example, a connection between far away objects is supported by the meta-metamodel, yet there is no way to actually show it. To address this problem, WebGME introduces the concept of *crosscuts*.

A crosscut is a collection of objects that the modeler wishes to view together. Currently, the user can manually drag objects into a crosscut view. In the near future, we will define a simple query language that can be used to issue one-time queries to collect models from anywhere in the composition hierarchy. Existing associations between objects in a crosscut are depicted by various lines between the objects. For example, inheritance is shown similar to UML class diagrams, while pointers are visualized with lines and arrows. In addition to visualization, the main utility of crosscuts is that they serve as association editors. The target of pointers and set membership can be edited here. Deleting a model from a crosscut does not delete the object from the project, it simply removes it from the given crosscut.

Each crosscut has a context model, the designated container for new model elements created in the crosscut. (Note that it is atypical to create new models since a crosscut is meant to be a collection of already existing models. However, a connection is a model with two pointers, so allowing new connections in crosscuts was the motivation behind this design decision.) The default context is *Root*. As *Root* can contain anything, crosscuts can be freely constructed. However, if the modeler chooses a context different from *Root*, the composition rules of the metamodel apply (even though crosscut containment is not composition). This is actually a great way to control and manage crosscuts. On the flip side, if one wants a crosscut with no constraints, but wants to avoid creating too many crosscuts in *Root*, one can simply create a model and specify that it can contain *FCO*s. Any instance of such a model can now serve as the context for unconstrained crosscuts.

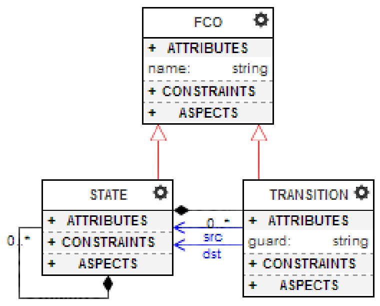


Fig. 2. HFSM Metamodel

One use for crosscuts in WebGME is for metamodeling. Recall that meta information can be specified anywhere in the composition hierarchy. Therefore, there is no single model to show to edit the metamodel of the DSML. In WebGME, a crosscut is created for the metamodel where the user drags in all models that need to contain DSML specification. It is there and only there, where meta information can be specified. Of course, the metamodel is a special crosscut, because a new association created there does not actually create a new instance of a pointer, for example, but instead specifies that the given kind of pointer of a model can point to the selected model (and its instances).

Consider Figure 2 that depicts the metamodel of a simple hierarchical finite state machine (HFSM). It shows that both State and Transition are derived from *FCO*. Note that unlike in any other tool we are aware of, this inheritance relationship was not drawn explicitly by the user. Instead, when the State and Transition models were created in the first place, they were instantiated from a model, in this case, *FCO*. The metamodel only displays these already existing inheritance relationships; they cannot be edited per se. On the other hand, the associations in Figure 2 were created in the meta crosscut. For example, the *src* and *dst* pointer specifications were drawn by the user specifying that a transition represents a relationship between two states. The default WebGME editor, in turn, will show these as connections (explained above) as expected in an HFSM.

3 Collaboration

Large-scale information systems modeling poses unique challenges that ad-hoc use of simple modeling tools cannot adequately address. As the amount of data stored, manipulated, and analyzed and the number of users and stakeholders spread across multiple institutions increase, the coordination of the modeling process becomes exponentially more difficult. There are known domain specific solutions targeting the versioning and configuration of coarse grained model hierarchies, such as the approach for CAD designs [17], but most domain models are much finer grained. Domain specific (meta)modeling tools need to scale up to support decentralization, collaboration, domain evolution and at the same time, ensure consistency.

Because of the very high level of interconnectedness among the models, even simple operations (such as copy or delete of a hierarchical model) can affect a large part of the model database. In our experience, this renders lock- or partition-based cooperation infeasible, especially when the system must always present consistent information to users and external tools. In large-scale modeling, users should have decentralized access to the model database where concurrent modifications even to the same set of models are possible. WebGME addresses this challenge by introducing a very lightweight branching scheme where different branches structurally share the same models if those have not been modified (see Section 4).

WebGME supports online collaboration where changes are immediately broadcast to all parties and everyone sees the same state. This is similar to how Google Docs works, except here the models have a much richer data model making consistency management more challenging. Since branch updates are very cheap and store only those objects that are explicitly modified, these changes can be broadcast to all participating parties and concurrent editing conflicts can be detected, retried, or rejected with immediate visual feedback.

The exact datamodel supporting the quick dissemination of objects between the server and clients is described in the next section. At this point, it is enough to know that every revision in the object database is uniquely identified by the hash value of a commit object, and that a commit object uniquely describes all models in the model hierarchy at a particular time instant.

Clients can freely create commit objects and send branch update messages to the server. Each branch update request contains the hash values of the old and new commit objects, and the database backend verifies (among others) that the old hash value matches the current branch hash before it is updated. With this protocol, the backend ensures that each branch has a linear history and rejects those branch updates that would fork the branch. If a branch update is accepted, then the new hash of the commit object is broadcasted to all clients. If a branch update is rejected (because another client has made a concurrent change and the old hash value does not match), then the client has the following three options: 1) reject the change made in the user interface and present the new version to the user, 2) automatically fork the branch (creates a new branch) and indicate this to the user, or 3) perform a merge of the local modifications to the branch with the changes already in the database and retries the branch update request. Currently, we support option 1 only, as automatic merging is not yet implemented.

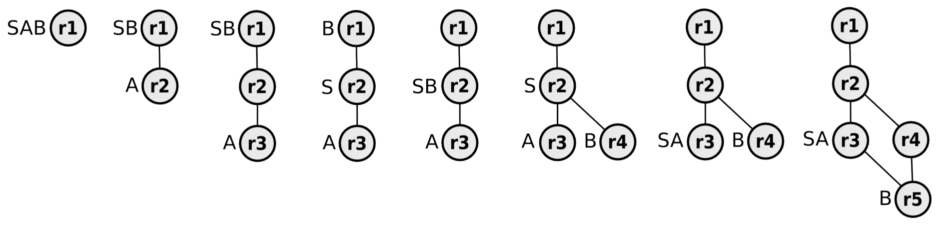


Fig. 3. Evolution of a branch during concurrent editing

As a simple example, let us consider two users concurrently editing a model hierarchy (see Figure 3). Initially, there is only one revision (r1), and both users (A and B), as well as the server (S), agree that the head of the branch is r1. Then user A makes two quick changes on his client computer (r2 and r3), which are immediately displayed (since all changes are performed synchronously without communicating with the server), and two branch update messages (r1→r2 and r2→r3) are sent to the server. After some time the server gets the r1→r2 message, at which time user B still sees revision r1, the server thinks that the head is at revision r2, and user A sees revision r3. The server sends out a notification to all clients that the head of the branch has changed to r2. User B displays the new head (r2), but user A ignores this update because it sees that he is already ahead of r2 and knows that it has already sent those updates.

At this point user B makes a concurrent change (r4) branching the history, but she does not know this yet, and notifies the server with an r2→r4 message. The server gets the r2→r3 message from A first in this example, so it updates the head to r3 and notifies all clients again. Then the server gets the r2→r4 update message from user B, but ignores it since the old revision in the update message (r2) does not match the current revision (r3). User B will get the head change message from the server with revision r3, and sees that this is not an ancestor of his current revision (r4). At this point user B knows that the branch has forked and his version is not the official one. Currently, we notify user B of this situation and discard his change, but with automatic merging, the client program for user B can perform the merge (r5) and can retry the branch head update request with r3→r5. If user A does not modify the branch in the meantime, then the server accepts this update and notifies user A.

In the near future, WebGME will also support merging branches. This will allow for an additional kind of collaboration where users fork the project, work on their own branches and once they are ready with their modifications, merge the changes back into the master branch. The structural consistency of the models are maintained by each basic operation and verified at merging. In most cases, the system will able to perform the merge automatically, but should a conflict arise because of conflicting modifications to the same models, the system will reject the merge and offer manual or guided conflict resolution on the client.

The version control scheme already enables users to work on and analyze consistent snapshots of the database without stopping others from modifying the models. This means that long running model translators, code generators or analysis tools can run while users carry on model building concurrently.

It is instructive to see why current solutions cannot meet the requirements of large scale modeling. Distributed Revision Control (DRC) is a well-known and scalable solution to source code management, but it requires clients to download full revisions to make changes [18]. However, we need to allow users to delete, copy, and move hierarchical models consisting of thousands of elements without downloading the internal structure of those models to the client. Moreover, as we have explained, the rich inter-model relations (such as inheritance) would also require the client to potentially download and modify an even larger set of objects. Many cloud-based information systems provide extreme scaling (e.g. Twitter, Facebook, Wikipedia, etc.), but do not provide branching and consistent snapshots. For example, Wikipedia has revision control of individual pages, but it does not allow users to concurrently fork the entirety of Wikipedia, update thousands of pages, and merge these changes back. Online collaborative text and diagram editors (e.g. Google Docs, Lucidchart, etc.) do not support branching and store individual artifacts separately with no integration. On the other hand, WebGME enables large-scale collaborative modeling with refactoring capabilities and consistency guarantees.

Beyond supporting collaborative online work, model evolution, and conflict resolution within individual projects, in the future, WebGME could foster model and language reuse on a much larger scale than it is currently practiced. Design publishing, discovery, and change tracking are poorly supported by current desktop-based model editors. We believe that the stimulating effects of SourceForge, Google Code, and GitHub (among others) on code reuse can and should be replicated for model-based design. The WebGME infrastructure can serve as cloud-based live repository of DSMLs and corresponding domain model libraries as current online project repositories are heavily source-file oriented and, hence, inadequate for model-based design collaboration.

4 Data Model

In this section, we describe the underlying data model of the system as an object graph and explain how the elementary operations (copy, delete, move, instantiate, update, etc.) are efficiently implemented without sacrificing extreme scalability and data integrity. First, we present a simplified view of the proposed architecture, then indicate the real difficulties.

Our primary goal is to ensure structural sharing of model objects between different branches of the database. We achieve this by organizing the objects into a containment hierarchy tree, where each object in the database stores the identifiers of its children, but not that of its parent.

If an object (e.g., G in Figure 4) needs to be updated, we simply make a copy, assign a new identifier (G'), recursively copy the parents (C, A) and replace the old child identifiers with new ones. This way the old and new versions of the graph structurally share a large portion of their objects. Since we do not store the parent in the child object, we can simply implement the copy operation as adding the identifier of an already existing model to the list of children. From the perspective of the user (and other tools), we have a new deep copy of the model, but in the database we did not have to recursively traverse its children because we just reused old content.

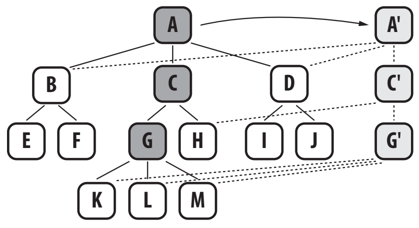


Fig. 4. Structural sharing of objects

We never modify any model object in the database, only create new objects that link to the old ones. Therefore, we can efficiently traverse and compare different versions of the model database and discover if large portions of their objects are the same. Instead of using standard identifiers for each object, we use a hash (SHA1) of the content of the model. Even if the same object is recreated on different clients (or the same modifications are preformed on the same client later), the modified objects will have the same hash value and only a single database object is created. The object database contains one root object for each version of the model, essentially tracking its evolution. The root objects are linked by commit objects that record parent commits, the new root object, and among others, the user who created the new tree. Another benefit of the use of hashes is that clients (browsers) can cache model objects freely, since they are not going to get modified, and can verify the integrity of the models.

The client (browser) does not need to download the whole database in order to perform operations on the models, only those objects that are necessary to be displayed to the user. For example, we can move, copy, or delete whole subtrees without ever downloading the internal objects. Note that the discovery of the model database is inherently asynchronous; the client needs to download new content, but modifications can be performed locally without server interaction except for eventually saving the new version of the model objects to the server. The saving of the new objects can be arbitrarily delayed, unless the user wants online collaboration with other users or wants to minimize accidental branching. Therefore, it is possible to support offline work where the user can continue editing the model without restriction if all the required content is already downloaded to the browser. The integrity of the modifications is maintained, and the changes can be uploaded to the database when connectivity is restored.

So far what we have described is very similar to how Git [19] operates, except we do not want to download the whole repository or the whole tree to the client, and we intend to use the browser as our management and editor tool. The real challenge is to track and update the rich inter-model relations in a consistent way without sacrificing the benefits presented above.

Imagine that we have an association between objects H and M in Figure 4, and we want to delete object G. Since all operations are hierarchical, object M, and therefore, the association between M and H needs to be deleted as well. As we have explained, the client loads only those objects that are absolutely necessary, which includes the parents of loaded objects, so the browser knows of A, C, and G, but not of H or M. Somehow we need to change H without loading it and remove that association that is visible in the old version of H. WebGME stores associations not at M or H or both, but at their common ancestor, that is, at C. Therefore, when we want to delete object G we immediately know all external associations that tie an object within G to another object that is not inside G (these associations are stored at C and A), so we can remove these when object G is deleted.

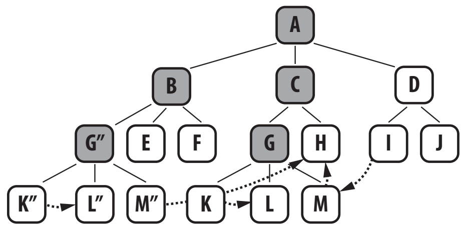


Fig. 5. Copying of associations

During a deep copy operation we also need to copy associations, but this case is more complicated than deletion because we need to distinguish internal and external associations and consider their direction. Suppose that we make a copy G'' of G and insert it in parent B (see Figure 5). An association is internal if both of its endpoints are within G, for example K-L. Internal associations are stored within the subtree of G because their common ancestors are also below G, so these associations are automatically copied. An external association pointing from an object below G to an object outside of G, e.g. from M to H, needs to be copied and the new association will point from M'' to H. An external association pointing from outside of G to an object within G, e.g. from I to M, is not copied because those reference a specific object. As we have seen, we have to maintain the direction of associations to properly maintain the semantics of copy and delete operations, and in this regard, associations behave like pointers in programming languages. Observe, that if G is loaded to the client and the new parent B of G'' is also loaded, then we can perform the copy operation and update all external associations at C and A without loading any new objects.

Movement of objects is the most complicated operation in the containment hierarchy, but even in this case, all associations can be properly updated within the already loaded parent objects. This means that all basic operations (delete, copy, move) can be performed in the client on arbitrary large subtrees without loading any new objects or even talking to the server.

The most important inter-model relationship is inheritance, which is significantly more challenging to support than associations. Simple changes in a base type can have an influence on all subtypes, but again we do not want to load all instances just because we modify the base type. Containment and inheritance interact in surprising ways, for example, deletions can have a cascading effect through containment and inheritance. To combat these, we dynamically compute the inheritance, where each object stores internally only those attributes and children that are different than those that are already present in its base type. This logic works even for associations, however, some extra logic is needed to allow the deletion of associations in instances. Currently, moving of objects within basetypes that already have instances are not supported on the client because this would still require the traversal of the inheritance chain. Deletion and coping does not have this restriction, and currently we are investigating ways to be able to support moving of objects in this case, as well.

To support this datamodel and online collaboration, the database backend provides the following two services: 1) respond to load and save requests of objects that are identified by their hash, and 2) store the current hash value of the commit object for each branch and broadcast changes of this hash value to connected clients. Since each object is uniquely identified by its hash value, the load and save requests can be completely reordered and no coordination between clients is required (only hash collisions need to be monitored for safety). On the other hand, updates of the current branch hash has to be serialized and all new objects should already be in the database when the new hash value is updated and broadcasted. Once the branch hash value is broadcasted, the clients update their user interface and download all missing objects from the backend.

With the datamodel described above, users can concurrently delete, copy and move entire subtrees consisting of millions of objects with the same efficiency as operating on a single object. Moreover, all these operations can be performed immediately on the client with no database round trip, so WebGME can provide instant visual feedback even if the network connection is slow or disconnected.

5 Architecture

WebGME is designed from the ground-up as a modern web application, using a *single page* interface and advanced AJAX communication patterns. Figure 6 illustrates the high-level system architecture. By choosing JavaScript for implementing all core components and with a re-configurable stack of data access layers (*database driver, cache, remote access*), WebGME allows for different deployment scenarios tuned for scalable collaboration, offline work and/or for high-performance and high-bandwidth model interpretation. In the most common deployment model, a relatively thin server-side component—running as a Node.JS process—acts as a communication bridge between the model storage (MongoDB) and multiple browser-based clients. Beyond providing serialized read/write data access, it sends broadcast messages to all connected clients after each update—using WebSockets as the transport protocol for both tasks. In a high-performance scenario, a single client can be deployed directly and exclusively on top of the database interface in a Node.JS container on the server. Intermediate layers in the data access stack also enable intentional or accidental (e.g., due to network problems) offline work.

The most critical components (Core and Client) are deployed on top of the data access stack. These layers assemble and maintain consistent in-memory snapshots of the model hierarchy and provide Model API, the common basic interface for all higher level components. This interface is directly used by the visualization stack (Section 6) and by high-performance plug-ins implemented in JavaScript. These plug-ins can target specific domains for a wide spectrum of automated tasks, such as model analysis, simulation, verification, code and report generation, model transformation, and design space exploration. Also, the architecture can be extended by domain-independent plug-ins for providing and integrating new generic tools.

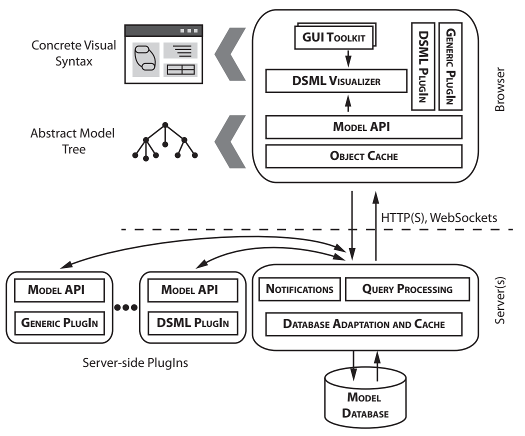


Fig. 6. High-level system architecture

Note that developing new plug-ins against the Model API requires the components to be developed in JavaScript. To overcome this limitation and provide data access for the models for the widest developer community, a REST web services API is also provided by our server. This interface enables language and technology independent access to the same data-model that is available via the Model API. The trade-off of the REST interface is slower access (significantly higher latency and serialization/de-serialization overhead). Note that both *native* JavaScript and REST components can be deployed on client and server side. Although server-side REST components still use the same general infrastructure as those on the client side, these benefit from the physical proximity to the server.

Our experience shows that higher-level domain-specific APIs can dramatically boost the productivity of domain developers. These auto-generated APIs 'speak the language' of the domains, and can significantly reduce the time and effort needed to develop new plug-ins. The definition and generation of these APIs—based on the meta information in the model—is part of our future work.

Another recurring pattern we identified in a large sample of existing third-party tools uses a semi-offline processing approach for the model hierarchy. These tools initially traverse the entire model and store it in an intermediate format for performance and convenience reasons. These types of components can leverage the model export/import facility on our server, which supports full and partial (de-)serialization of the models in JSON format. This capability is integrated with the REST service interface.

Beyond acting as a data bridge, sending broadcast notifications and providing the REST API, the current server-side component is responsible for bootstrapping the browser application with static content and for implementing authentication and authorization tasks. The authentication infrastructure is based on the Passport framework [20], which supports a comprehensive set of strategies and protocols from simple username/password pairs to OpenID and OAuth providers (e.g. Facebook, Google).

6 Visualization

Traditional MIC model editors enable an extremely fast early prototyping phase by providing default visualization and user controls for each element of the DSML language being developed. These editors provide built-in user interface logic for each fundamental (meta-meta) concept. Although, parts of the built-in behavior is customizable—with bitmap and vector images or by providing custom rendering and event handling code plug-ins—many environments make it difficult to implement a fundamentally different user interface experience. Deviating from the default mapping of abstract model elements to visual primitives or implementing radically different visual behavior is becoming exponentially more difficult. Thus, after the early prototyping phase, many DSML designers struggle with developing a refined domain-specific and user-centric editor experience. Typically these custom views include tabular data representation, textual formats (source code, XML, JSON), form-based user interfaces, or complex data visualization with precisely controlled layout and rendering. In all these cases, customization needs to reach beyond the rendering of individual model elements and has to control the overall mapping of abstract data model elements to visual primitives and UI actions to operations on the data.

Hence, WebGME provides a visualization toolkit as opposed to the customizable model editor approach of GME. The key difference is that with the toolkit, the DSML designer has more control over the visualization aspects of the language. Elements of the toolkit include layout managers, line and area rendering primitives, and in-place text editors. These elements handle the rendering and UI interaction tasks. The DSML designer will use the graphical building blocks and provide the mapping between the model database (Model) and the toolkit elements (Views and Controllers).

Visually complex and large models pose scalability challenges for the UI much sooner than for the underlying model database. Graphical model editors typically address the visual scalability problem by either providing flat 'model canvases,' with which the model can be partitioned to multiple sheets with some shared entities, or by using hierarchical decomposition. Examples for the former approach are UML class diagrams, circuit schematics, and Petri nets, while the second approach is more prevalent in modeling signal flow graphs, hierarchical state machines, and design spaces. Both methods have limitations: the model builder has to reason about a 'mentally stitched' model or constantly navigate into a deep model hierarchy while taking extra effort to model cross-cutting relationships across distant model elements. Filtered views (i.e., aspects)—showing only a subset of the elements of the model—is a simple but insufficient feature towards providing a scalable user interface.

Both GME and WebGME supports model canvases, hierarchy and filtered views (aspects), but WebGME takes a significant step beyond these standard techniques with the introduction of crosscuts. Crosscuts decouple visualization from the model hierarchy and provide completely user-defined orthogonal views of the models. The default visualization in crosscuts focus on associations by displaying any existing relationship between members of the crosscut with lines, as well as provide the means to modify them or create news ones. However, any other type of visualization can be implemented as required by the given domain. Elements of the crosscut can be selected by direct user actions or soon user-defined queries can gather model objects and populate the crosscut. We believe that by providing powerful query-based cross-cutting views one can build truly scalable domain specific interfaces.

7 Multi-Paradigm Modeling Support

The best way to show how WebGME supports multi-paradigm modeling is through an example. Consider a simple DSML, a hierarchical signal flow language similar to Simulink called *SignalFlow*. Figure 7 shows WebGME with the SignalFlow DSML inside a Chrome browser. The metamodel is shown on top. The main concepts are *Compound* and *Primitive*, signal flow operators that are the composite and leaf nodes of the model hierarchy, respectively; *Input* and *Output* ports that provide the signal interface for the operators; and *Flow* that are the connections between ports. Parameters (with *DataType* and *Size* attributes) provide configuration parameters to operators.

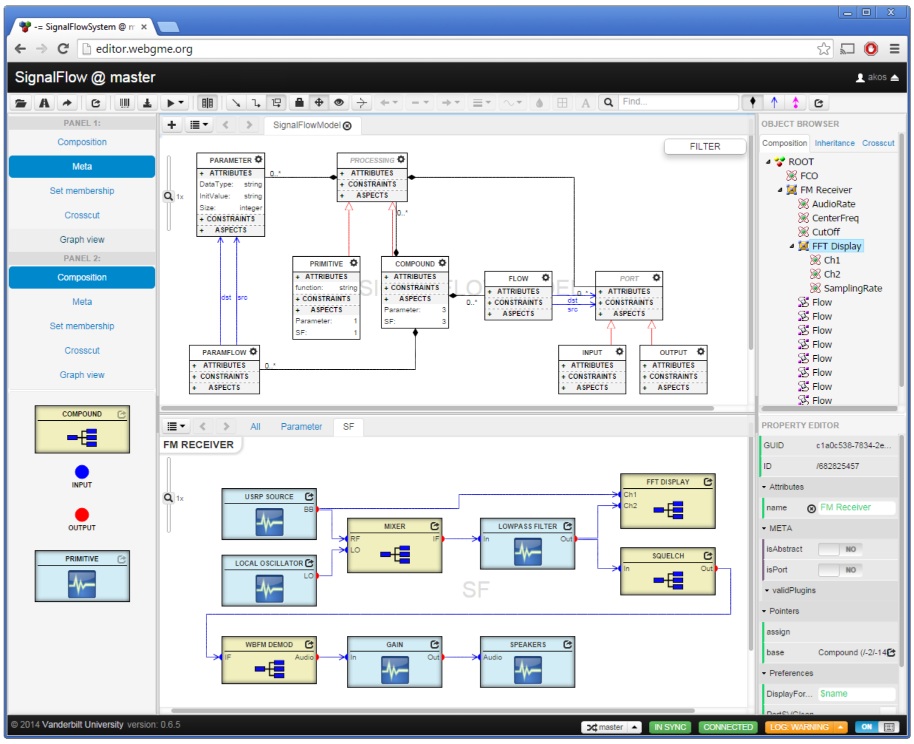


Fig. 7. Example Signal Flow DSML in WebGME

The left side of the screen shows the panel control buttons and below them the Part Browser that displays the models that can be instantiated inside the model loaded in the current panel as defined by the metamodel. Dragging and dropping a model from the Part Browser creates a new instance of the given model. The top right side of the user interface shows the Object Browser that shows the composition hierarchy of the project starting at Root. Below is the Property Editor where attributes, preferences and other properties of the currently selected model can be edited.

The user interface of the tool provides quite sophisticated features especially considering the browser-based execution environment. Drag and drop, context menus, search, autorouting, Bezier curves, and various visualizers in addition to what is shown in the figure are all provided. However, a detailed description of the user interface is beyond the scope of this paper.

Suppose we want to create a new DSML, that supports Signal Flow models, but also allows the modeling of simple multiprocessor hardware *and* the assignment of signal flow components to processors. Similarly to GME, WebGME also supports libraries. We can select any model in a project and export it as a library. This takes the composition tree rooted at the given model and generates

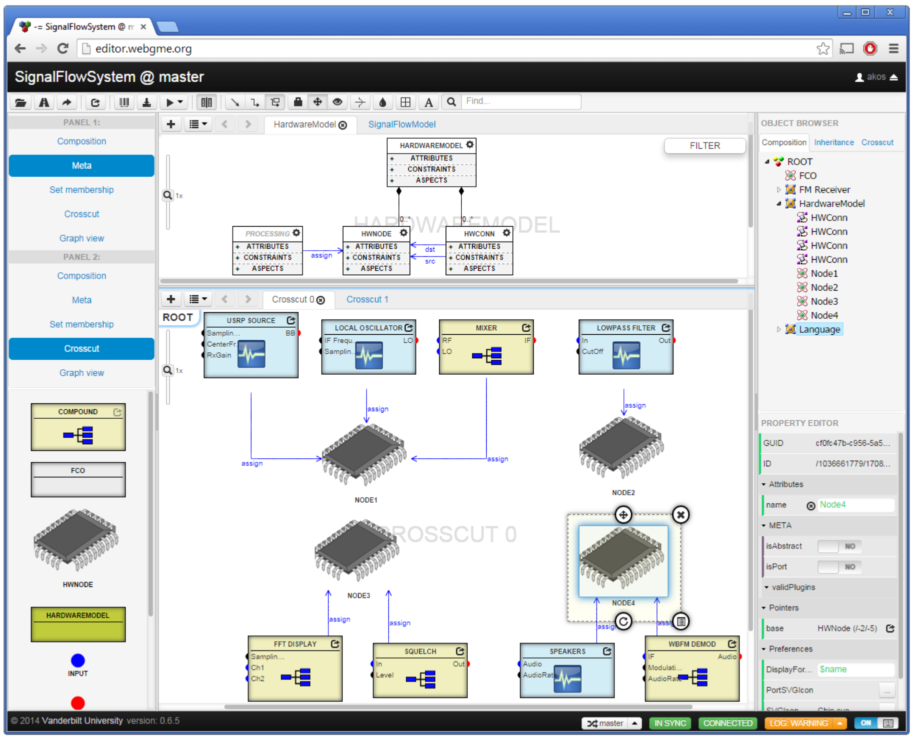


Fig. 8. Processor Assignment for Signal Flow Models

a JSON file from it. In our example, the *SignalFlow* metamodels were stored not directly in *Root*, but in a model contained by *Root* called *Language*. We exported it and then imported it into a new project called *SignalFlowSystem*. Imported libraries are read-only models, but they can be derived and associations can point to any part of them. In this new project, we created a simple computer hardware DSML. Finally, the only thing missing from the metamodels is the specification of the assignment. This is the concept that ties together the two paradigms. This can be done simply by dragging in the *Processing* model (the signal flow base component) in the Hardware metamodel and adding a new pointer to *Processing* called *assign* that can point to *HWNode*s (see top of Figure 8). Note that none of the composition rules were changed, that is, signal flow models cannot contain hardware models and vice versa. The only relation between the two sub-languages is the *assign* pointer, but it cannot be visualized in any of the models nicely. But we can create a crosscut that can contain *Processing* and *HWNode* models and it will show the assignment pointers and also enables the user to edit them as shown in the bottom part of Figure 8.

8 Related Work

There have been promising approaches to (i) collaborative modeling, (ii) web-based modeling environment, and (iii) model versioning. This section reviews the results closest to our solutions.

Collaborative modeling is used in specific domains such as mechanical engineering [21], automotive industry [22], and UML [23]. A collaborative DSML definition process is presented in [24, 25] which could be supported by WebGME. SLIM [26] is a prototype of a collaborative environment executed in a web browser. The Connected Data Objects (CDO) [27] is a model repository and a run-time persistence framework for EMF. It supports locking, offline scenarios, various persistence backends, such as Hibernate, and pluggable fail-over adapters to multiple repositories. As a part of CDO, the Dawn framework supports collaboration on the user interface level with functions such as locking, conflict detection and resolution. It is integrated with multiple graphical editors. In contrast with our approach, CDO supports model integration on the model level, and not on that of the edit operations. If two transactions are trying to modify the same object, CDO signals a conflict for the second transaction as opposed to our approach, where this creates a new branch automatically. CAMEL [28] is also an eclipse plugin that supports collaborative interaction via modeling, drawing, chatting, posterboards, whiteboards, and it is capable of replaying online meetings. Its focus is on collaborative communications rather than versioning and collaborative use of domain-specific languages.

AToMPM [29], a web-based metamodeling and transformation tool for Multi-Paradigm Modeling, is the closest to our work. While many of the authors' architectural decisions are similar to ours, versioned repository, the fusion of metamodeling and prototypical inheritance and crosscuts are the biggest differentiating factors.

A summary of model versioning can be found in [30]. A formal approach is contributed in [31]. [32] describes an extension of AMOR [33] to facilitate model-based merging. [34] describes precise methods for parallel dependent graph manipulations when insertion has priority over deletion. Our philosophy is rather to avoid situations where merge is needed – the fine grained commit cycles serve exactly this purpose. However, especially after offline work, these solutions can extend our approach when two branches need to be merged.

Web technologies have advanced to the point where it is feasible to build user-friendly and visually appealing user interfaces with good performance inside a web browser. Lucid Charts [35] and CircuitLab [36] are excellent examples of what is possible. Some of these even support online collaboration. On the other hand, these tools 1) employ relatively simple, typically flat data models and 2) are very specific to their respective domains. They do not solve the challenges associated with evolutionary language design, configurability, branching, and extensibility.

9 Conclusions

WebGME has a number of novel features and several advantages over desktop-based (meta)modeling tools. The browser-based client is platform-independent and does away with installation and software update issues. The data model and software architecture were designed from the ground up to provide scalability, seamless collaborative modeling and powerful model versioning. The prototypical inheritance and crosscuts are probably the two most unique features of the WebGME meta-metamodel providing DSML and model complexity management.

WebGME is still under development. The two most significant missing pieces are merge support for branches and a constraint manager. The merge operation is critical to enable other modes of collaboration beyond immediate concurrent updates. Constraints also play an important role in DSMLs. GME has an OCL-based constraint manager which proved very useful for people who were willing to learn OCL, but were ignored by most users. What constraint language WebGME will ultimately utilize is still up for debate.

WebGME supports multi-paradigm modeling using inheritance, libraries and crosscuts. Multiple inheritance would be really powerful for metamodeling in general and multi-paradigm modeling in particular. Merge may enable multiple inheritance support, however, it is a really challenging concept because of the complex interplay between composition and inheritance.

9.1 Acknowledgement

This work was sponsored in part by the Defense Advanced Research Project Agency (DARPA) and by the European Union and the European Social Fund via project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013).

References

1. Sztipanovits, J., Karsai, G.: Model-integrated computing. Computer **30**(4) (1997) 110–111
2. Lédeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. Computer **34**(11) (2001)
3. Long, E., Misra, A., Sztipanovits, J.: Increasing productivity at saturn. Computer **31**(8) (1998) 35–43
4. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE **91**(1) (2003) 145–164
5. Mathe, J.L., Ledeczi, A., Nadas, A., Sztipanovits, J., Martin, J.B., Weavind, L.M., Miller, A., Miller, P., Maron, D.J.: A model-integrated, guideline-driven, clinical decision-support system. Software, IEEE **26**(4) (2009) 54–61
6. Lattmann, Z., Nagel, A., Scott, J., Smyth, K., Porter, J., Neema, S., Bapty, T., Sztipanovits, J., Ceisel, J., Mavris, D., et al.: Towards automated evaluation of vehicle dynamics in system-level designs. In: ASME 2012 Computers and Information in Engineering Conference, ASME (2012) 1131–1141
7. Levendovszky, T., Balasubramanian, D., Coglio, A., Dubey, A., Otte, W., Karsai, G., Gokhale, A., Nyako, S., Kumar, P., Emfinger, W.: Drems: A model-driven distributed secure information architecture platform for managed embedded systems. IEEE Software (2014) 1
8. Bagheri, H., Sullivan, K.: Monarch: model-based development of software architectures. Model Driven Engineering Languages and Systems (2010) 376–390
9. Bézivin, J., Brunette, C., Chevrel, R., Jouault, F., Kurtev, I.: Bridging the generic modeling environment (GME) and the eclipse modeling framework (EMF). In: Proceedings of the Best Practices for Model Driven Software Development at OOPSLA. Volume 5., Citeseer (2005)
10. Bunus, P.: A simulation and decision framework for selection of numerical solvers in. In: Proceedings of the 39th annual Symposium on Simulation. ANSS '06, Washington, DC, USA, IEEE Computer Society (2006) 178–187
11. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: An industrial experience report. In: Generative Programming and Component Engineering, Springer (2002) 156–172
12. Stankovic, J., Zhu, R., Poornalingam, R., Lu, C., Yu, Z., Humphrey, M., Ellis, B.: Vest: an aspect-based composition tool for real-time systems. In: Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE. (May) 58–69
13. Thramboulidis, K., Perdikis, D., Kantas, S.: Model driven development of distributed control applications. The International Journal of Advanced Manufacturing Technology **33**(3) (2007) 233–242
14. Mohanty, S., Prasanna, V., Neema, S., Davis, J.: Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. ACM SIGPLAN Notices **37**(7) (2002) 18–27
15. Hemingway, G., Neema, H., Nine, H., Sztipanovits, J., Karsai, G.: Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach. Simulation **88**(2) (2012) 217–232
16. Ledeczi, A., Balogh, G., Molnar, Z., Volgyesi, P., Maroti, M.: Model integrated computing in the large. In: Aerospace Conference, 2005 IEEE, IEEE (2005) 1–8
17. Katz, R.H.: Toward a unified framework for version modeling in engineering databases. ACM Comput. Surv. **22**(4) (December 1990) 375–409

18. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. IJWIS **5**(3) (2009) 271–304
19. : GIT Homepage. `http://git-scm.com` Cited 2013 Mar 14.
20. : Passport, authentication middleware. `http://passportjs.org` Cited 2014 Mar 17.
21. Li, M., Wang, C.C., Gao, S.: Real-time collaborative design with heterogeneous cad systems based on neutral modeling commands. Journal of Computing and Information Science in Engineering **7**(2) (2007) 113–125
22. Kong, S., Noh, S., Han, Y.G., Kim, G., Lee, K.: Internet-based collaboration system: Press-die design process for automobile manufacturer. The International Journal of Advanced Manufacturing Technology **20**(9) (2002) 701–708
23. Boger, M., Graham, E., Köster, M.: Poseidon for uml. Pode ser encontrado em http://gentleware.com/fileadmin/media/archives/userguides/poseidon_users_guide/book1.html (2000)
24. Izquierdo, J.L.C., Cabot, J.: Enabling the collaborative definition of dsmls. In: Advanced Information Systems Engineering, Springer (2013) 272–287
25. Izquierdo, J.L.C., Cabot, J., López-Fernández, J.J., Cuadrado, J.S., Guerra, E., de Lara, J.: Engaging end-users in the collaborative development of domain-specific modelling languages. In: Cooperative Design, Visualization, and Engineering. Springer (2013) 101–110
26. Thum, C., Schwind, M., Schader, M.: Slima lightweight environment for synchronous collaborative modeling. In: Model Driven Engineering Languages and Systems. Springer (2009) 137–151
27. Stepper, E.: Connected data objects (cdo). Website http://www. eclipse. org/cdo/documentation/index. php, seen November (2012)
28. Cataldo, M., Shelton, C., Choi, Y., Huang, Y.Y., Ramesh, V., Saini, D., Wang, L.Y.: Camel: A tool for collaborative distributed software design. In: Global Software Engineering, 2009. ICGSE 2009. Fourth IEEE International Conference on, IEEE (2009) 83–92
29. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Ergin, H.: Atompm: A web-based modeling environment, MODELS (2003)
30. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. International Journal of Web Information Systems **5**(3) (2009) 271–304
31. Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: Algebraic foundations and the tile notation. In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models. (2009) 7–12
32. Brosch, P., Seidl, M., Wieland, K., Wimmer, M., Langer, P.: We can work it out: Collaborative conflict resolution in model versioning. In: ECSCW 2009. Springer (2009) 207–214
33. Altmanninger, K., Kappel, G., Kusel, A., Retschitzegger, W., Seidl, M., Schwinger, W., Wimmer, M.: Amor–towards adaptable model versioning. In: 1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS. Volume 8. (2008) 4–50
34. Ehrig, H., Ermel, C., Taentzer, G.: A formal resolution strategy for operation-based conflicts in model versioning using graph modifications. Springer (2011)
35. : Lucidchart. `http://www.lucidchart.com` Cited 2014 Mar 17.
36. : CircuitLab. `https://www.circuitlab.com` Cited 2014 Mar 17.

Towards an Approach for Orchestrating Design Space Exploration Problems to Fix Multi-Paradigm Inconsistencies

Sebastian J. I. Herzig, Benjamin Kruse, Federico Ciccozzi, Joachim Denil, Rick Salay, and Dániel Varró

sebastian.herzig@gatech.edu,bkruse@ethz.ch, federico.ciccozzi@mdh.se,joachim.denil@uantwerpen.be, rsalay@cs.toronto.edu,varro@mit.bme.hu

Abstract. In model-driven engineering, the aim of design space exploration (DSE) is to generate a set of design candidates that satisfy a given set of constraints and requirements, and are optimal with respect to some criteria. In multi-paradigm modeling it is not uncommon to perform a variety of computationally expensive analyses as part of such an exploration process. However, existing DSE techniques do not take dependencies between solver operations into account, thereby failing to provide a mechanism for pruning infeasible solutions early. Motivated by this source of inefficiency, this paper discusses a conceptual approach to orchestrating solvers and design space exploration problems.

Keywords: design space exploration, solver orchestration

1 Introduction

The design of complex systems necessitates the exploration and evaluation of design alternatives from different perspectives leveraging multi-paradigm modeling languages and tools. Different functionally equivalent design candidates need to be systematically sought out and derived by Design Space Exploration (DSE) techniques [1]. These candidates must simultaneously (i) conform to syntactic constraints and (ii) satisfy requirements defined as semantic properties (uniformly called multi-paradigm consistency criteria). Violating these constraints and consistency criteria can result in *inconsistencies.* Such inconsistencies can be resolved by changing the underlying model through fix operations [2].

Validating static well-formedness constraints, assuring semantic properties (e.g., correctness, completeness, determinacy) and evaluating extra-functional properties (e.g., availability, throughput) requires complex analysis techniques which exploit fundamentally different abstractions and solver technologies. Existing DSE techniques are limited to sequentially calling independently operating solvers. For instance, the generic DSE framework presented in [3] supports arbitrary analysis tools, but requires a predefined workflow. Similarly, Phoenix Integration ModelCenter [4] is a framework enabling DSE of a variety of parameterized models by calling external solvers in a pre-determined sequence. Both approaches are limited in that explicit dependencies between solvers cannot be specified. Therefore, violations to constraints imposed by a previously called solver cannot be detected by subsequently called solvers, thereby potentially carrying forward and further exploring infeasible solutions.

This paper proposes an alternative approach based on performing semantic fixes to resolve diverse, multi-paradigm inconsistencies through semi-automated orchestration of existing checker and solver tools that are driven by a guided DSE process. The paper is a result of a collaborative effort of the authors at the 2014 Computer Automated Multi-Paradigm Modelling (CAMPaM) workshop. Our hypothesis is that focusing on orchestration and decomposition of the overall *design space exploration problem* (DSEP) is a necessary step towards managing the complexity associated with running expensive analyses in scenarios where multiple DSEPs must be solved simultaneously. Our proposed solution is an orchestration strategy that uses heuristics to define the data and control flow between the different analysis and exploration phases to call the appropriate off-the-shelf solvers as needed. Core is the idea of back-tracking solutions automatically by exploiting the dependencies of underlying solvers, the multi-paradigm consistency criteria, and the orchestration strategy.

The paper is organized as follows: we introduce our view on DSE and develop our conceptual idea through application to an example problem in Section 2. Our proposed approach is discussed in further detail in Section 3.

2 Orchestrating Design Space Exploration Problems

DSE is the process of generating and analyzing a set of design alternatives for the purpose of finding the most preferred configuration. In our framework we consider a *guided* incremental approach to exploration, in which potential *design candidates* (alternatives) must meet a set of *constraints*. Constraints can apply to both *numerical* and *structural* attributes [1]. Design candidates are evaluated based on quality metrics such as cost or dependability (*optimality criteria*). *Hints* are provided in the form of heuristics to guide the design process, which mitigates the prominent issue of most DSE approaches: their inefficiency in handling structural constraints and dynamic manipulation of elements due to the use of model checking in conjunction with exhaustive state space exploration [1]. Hints represent expert knowledge (i.e., *heuristics*) that can guide the exploration process by detecting dead ends or unpromising paths early.

2.1 Semantic Inconsistency Fixing as a DSEP

Within the context of our work, we view the constraints typically used for identifying feasible solutions in a DSE as *semantic consistency constraints*. A violation of these is indicative of an inconsistency. We say that semantic consistency constraints belong to either the class of *solution* or *path constraints*. One prime example of solution constraints are constraints used to ensure adherence to language syntax and structural semantics. Path constraints represent a set of conditions that valid exploration paths must fulfill; they are used to determine whether, as a result of applying an operation, the intended model functionality is preserved. Through path constraints, pruning of exploration paths enables to focus on optimizing only valid paths.

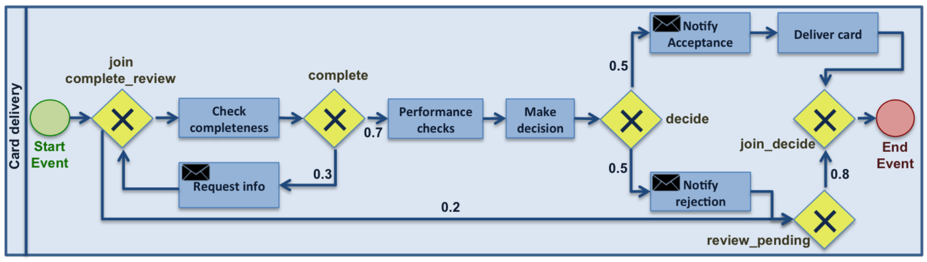


Fig. 1. A BPMN model for a card delivery system

Semantic inconsistencies can be identified by specifying paradigm-specific solution and path constraints. Given a graph-based model representation, *negative graph constraints* can be used for the purpose of checking static semantics [2]. In our framework, we associate these negative constraints with one or more *fixing operations* that can be used to alleviate the respective semantic inconsistency. Each fix is performed in two steps: first, an analysis is performed by a *checker*, which looks for the existence of possible inconsistencies with respect to desired characteristics. In a second step, the identified inconsistencies are *resolved*. The resolution is performed based on a set of given model transformation operations and is guided by i) a given optimality criteria, and ii) heuristics and hints. Dependencies between fixing operations are provided in the form of (formally captured) hints. These dependencies can then be used for early pruning of non-optimal solutions. For instance, if the optimization criteria implies a solution with the smallest difference to the original model, a dependency analysis of the model operations allows an a-priori determination of which combination of operations should be applied in what particular sequence [2].

2.2 Illustrative Example

In the following, we illustrate how different DSEPs can be orchestrated to find and resolve multi-paradigm semantic inconsistencies. For this purpose, a process model – more specifically, a *business process model* (conforming to the *Business Process Model and Notation* (BPMN) [5]) – is used. A business process model describes the set of activities that an organization should implement and execute to provide a particular service or product.

Figure 1 illustrates an example BMPN model for a card delivery process of a service company (only one lane is shown due to space constraints). The company checks if the client filled out the forms correctly and continues to process the order until the card is delivered. Because of production constraints or malformed requests, the order can be rejected.

When modeling this process, it is possible for a number of syntactic and semantic inconsistencies to be introduced. For example, one well-formedness constraint of BPMN2 states that a diverging OR-gate has to be preceded by a single decision activity to specify what gate has to be taken [5]. Additionally, a deadlock can be introduced if two processes in different lanes are each waiting for messages that are only sent after the respective waiting tasks have terminated. Both the detection and the fixing of these types of inconsistencies is non-trivial.

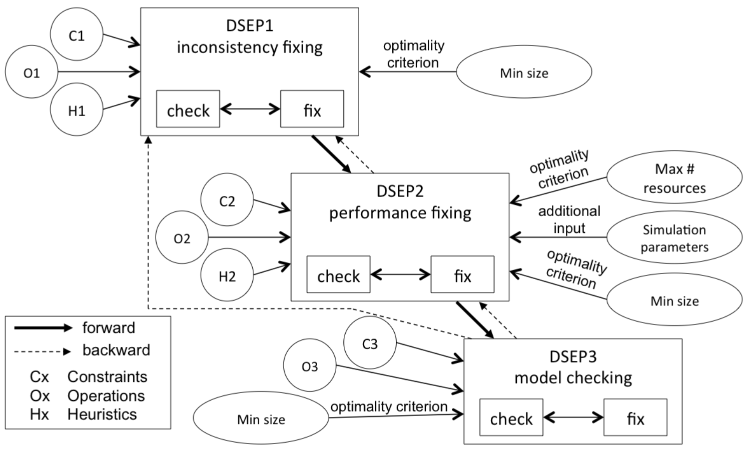


Fig. 2. Example orchestration scenario

In addition to these inconsistencies, it is often desired to optimize business processes. This requires exploration of the design-space of possible alternative processes that result in the same service or product (i.e., that are functionally equivalent). One possible optimization is the redistribution of resources to reduce the expected waiting times at some of the activities.

Exploring the design space requires all of these inconsistencies to be identified and fixed – ideally, in an automated fashion. Figure 2 illustrates one possible orchestration of solvers that check and fix the aforementioned inconsistencies. The first of the three DSEPs manages conformance to language well-formedness constraints, the second DSEP optimizes the performance of the model. Finally, the third DSEP performs a formal verification through model checking to check that deadlocks cannot occur. Inconsistencies are checked for by evaluating constraints (Cx) and fixed using a particular transformation operation (Ox).

In our example, the *performance fixing* DSEP is performed by first transforming the BPMN model to an extended queuing network [6]. The following types of performance fix operations are possible: (1) Resource Distribution: Resources are redistributed to each of the activities resulting in shorter or longer service times. (2) Parallelization: Activities that are executed in sequence can be parallelized. (3) Serialization and Deserialization: An activity is divided into multiple activities or merged into a single activity.

When executing different DSEPs sequentially, inconsistencies may be introduced. For instance, when a *parallelization* rule of the performance fixer is executed, a violation of a well-formedness constraint may occur: e.g., the activities *check performance* and *make decision* should not be parallelized because a diverging OR-node is connected to the decision activity. Therefore, if the rule is applied for either activities, the model violates the constraints checked by the first solver. Capturing such dependencies among fix operations formally as heuristics (Hx) across different solvers enables automated backtracking to previous DSEPs whenever necessary.

3 Discussion

Our illustrative example revealed several general aspects of the DSEP orchestration problem for inconsistency fixing. First, it highlighted the fact that DSE orchestration can be viewed as a decomposition problem: how to split a DSEP into a set of sequential and parallel smaller DSEP steps that, together, are less expensive than the original DSEP? Second, the ability to backtrack within an orchestration is essential. Third, an orchestration can be modeled using a modeling language with specialized semantics.

3.1 Decomposition of a DSEP

There are several dimensions along which a DSEP decomposition can occur.

Submodel decomposition. The model being fixed can be decomposed into submodels and thus the DSEP is split into an orchestration of a set of DSEP's over the submodels. For instance, in our example we could have decomposed the card delivery system model into separate submodels for each lane. Non-overlapping submodel DSEP's could then be executed in parallel in the orchestration. For overlapping models, interface automata [7] could be defined.

Abstraction decomposition. If there is a natural way to create abstractions of the model being fixed, the DSEP can be orchestrated into a sequence of increasingly refined DSEP's. For example, BPMN models can be abstracted by collapsing a portion of the model into a single Activity thus producing a more abstract model. A solution to the DSEP for the abstract model can be used to constrain the DSEP for the original model by limiting its search to a subspace. If no solution is found in the subspace, backtracking occurs to the abstract DSEP to find another abstract solution.

Constraint decomposition. Decomposing the set of solution constraints yields an orchestration into a sequence of DSEP's, each using a subset of the constraints. This is the kind of decomposition we used in the card delivery system example. If a solution is found in one step, it is passed as the initial model to the next DSEP step and so on. This kind decomposition is most effective if the fix operations of each step cannot cause a violation of the constraints in any of the previous steps. If this condition cannot be guaranteed, then the constraints of the first DSEP must be rechecked on a solution to the second DSEP and if a violation occurs, backtracking must be performed. Constraint decomposition is particularly useful when different subsets of constraints require different solvers (as is the case in our example). In this case, the constraints requiring more expensive solvers can be put later in the sequence.

3.2 Backtracking

As discussed above, some decomposition approaches require support for backtracking. This is required to ensure that the orchestration is *complete* - i.e., that every solution of the original DSEP is reachable by the orchestration.

In contrast to the related work discussed in the introduction that define specialized and explicit backtracking mechanisms for their DSE problems, we propose a *general automated and implicit* backtracking strategy. This is possible because we are restricting our attention to the problem of fixing inconsistencies and all DSEPs in an orchestration have a set of fix operations. An automated analysis of the dependencies among fix operations in the different DSEP's can determine the points at which backtracking is necessary. This is similar to what is known as *dependency-directed back-tracking* [8] in artificial intelligence. In our case, this entails the ability to automatically backtrack by exploiting known dependencies among underlying solvers and multi-paradigm consistency criteria.

3.3 Towards an orchestration modeling language

Our goal with this work is to implement automated backtracking semantics on top of an orchestration modeling language used for defining the forward flow of the orchestration. For example, a designer may use a language such as UML activity diagrams to define the forward flow orchestration of a set of DSEP's. Automated dependency analysis would be used to discover dependencies between these DSEP's and then when the orchestration is executed, this would the trigger automatic backtracking where necessary.

4 Conclusions & Future Work

This paper introduces a conceptual basis for orchestrating multi-paradigm design space exploration problems (DSEPs). A core idea is the decomposition of the overall DSEP by exploiting dependencies between model operations. By allowing for backtracking, the overall cost of multi-paradigm DSEPs can be decreased significantly. However, in our approach, this is a conclusion that can only be reached under the assumption that dependencies among model operations can be established – i.e, under the assumption that a common formalism for representing and manipulating models can be identified.

We believe that the semi-automation of the described solution is technically viable, and, for achieving it, further research will be carried out. Future work should include the development of a language for modeling DSEP orchestrations. Additionally, the concepts discussed in this paper could be extended to a *dynamic* orchestration where, similar to co-simulation, a control component manages interactions between DSEPs rather than relying on manually defined backtracking paths.

References

1. Abel Hegedus, Akos Horváth, István Ráth, and Dániel Varró. A Model-Driven Framework for Guided Design Space Exploration. In *Procs of ASE*, pages 173–182. IEEE Computer Society, 2011.
2. A. Hegedus, A. Horváth, I. Ráth, M. C. Branco, and D. Varró. Quick Fix Generation for DSMLs. In *Procs of VL/HCC*, pages 17–24. IEEE, 2011.
3. T. Saxena and G. Karsai. MDE-Based Approach for Generalizing Design Space Exploration. In *Procs of MODELS*, pages 46–60. Springer, 2010.
4. Phoenix Integration ModelCenter. `http://phoenix-int.com`.
5. Business Process Model And Notation (BPMN) Version 2.0, January 2011.
6. P. Bocciarelli and A. D'Ambrogio. Automated Performance Analysis of Business Processes. In *Procs of TMS/DEVS*, 2012.
7. L. De Alfaro and T. A. Henzinger. Interface Automata. *ACM SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.
8. R.M. Stallman and G.J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

Taming Multi-Paradigm Integration in a Software Architecture Description Language

Daniel Balasubramanian, Tihamer Levendovszky, Abhishek Dubey, and Gábor Karsai

Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37235, USA
{daniel,tihamer,dabhishe,gabor}@isis.vanderbilt.edu
http://www.isis.vanderbilt.edu

Abstract. Software architecture description languages offer a convenient way of describing the high-level structure of a software system. Such descriptions facilitate rapid prototyping, code generation and automated analysis. One of the big challenges facing the software community is the design of architecture description languages that are general enough to describe a wide-range of systems, yet detailed enough to capture domain-specific properties and provide a high level of tool automation. This paper presents the multi-paradigm challenges we faced and solutions we built when creating a domain-specific modeling language for software architectures of distributed real-time systems.

1 Introduction

Software architecture description languages offer a convenient way of describing the high-level structure of a software system. An architecture combines the individual pieces of a system, such as software components and communication networks, into an integrated view. Combining what are normally considered "separate" pieces of the system into such an integrated view allows the relationships between the different elements to be explicitly represented.

Software architecture descriptions serve several purposes. Apart from providing documentation and a high-level view of a software system's design, they can serve as the basis for automated code generation. Automated tools can generate skeleton code for the software components based on their interfaces and communication links to other components. This ability to translate a high-level description into lower-level implementation artifacts makes architecture description languages ideal for rapidly prototyping applications. Automated analysis can use an architecture description at design-time to answer questions related to security, schedulability and resource utilization.

One of the big challenges facing the software community is the design of an architecture description language that is both general enough to apply to a wide range of systems, but at the same time expressive enough to describe problems specific to individual systems and provide a high amount of automated tool support. For instance, AUTOSAR [2], a standard for defining software architectures in the automotive domain, is heavily tailored to concepts and standards found in the vehicle design domain. On the other hand, languages like AADL [8] can be too general to model software systems that contain concepts outside of its specification, such as custom security concepts.

One alternative to using standardized architecture languages is to create a custom language. This approach can work especially well when implemented with domain-specific modeling languages (DSMLs), which provide an intuitive syntax and ensure that only the relevant architectural concepts are captured. Metamodeling environments also facilitate rapid language development iterations. However, there are several multi-paradigm modeling challenges involved with this approach: architectures contain elements at different levels of abstraction, multiple formalisms and cross-cutting concerns such as security. A viable architecture description language must provide solutions to these challenges.

This paper presents our solutions to a series of multi-paradigm challenges we faced while building a domain-specific language for describing the software architecture of distributed embedded systems. The first challenge is combining multiple formalisms (textual code and block diagrams) to describe component interfaces. The second challenge is to transform high-level scheduling properties of individual elements into a combined temporal schedule. The third challenge is integrating different types of analyses into the modeling language. Even though our approach is tailored to our architecture language, the solutions are applicable across other architecture languages which face similar challenges on a similar level of abstraction, and thus want to provide a similar level of tool automation.

The rest of this paper is organized as follows. Section 2 provides an overview of the modeling language. Section 3 describes how we integrated a textual code formalism with graphical block diagrams. Section 4 describes how we transform high-level scheduling properties of individual elements into a low-level temporal schedule. Section 5 briefly illustrates the opportunities for design-time analysis. Section 6 presents related work, and we conclude in Section 7.

2 Overview

DREMS, Distributed Real-time Embedded Managed Systems [6], is a software infrastructure for designing, implementing, configuring, deploying and managing distributed real-time embedded systems. It consists of two major subsystems: (1) a design-time toolsuite for modeling, analysis, synthesis, implementation, debugging, testing and maintenance of application software built from reusable components, and (2) a run-time software platform for deploying, managing and operating application software on a network of computing nodes. DREMS is intended for platforms that provide a managed network of computers running distributed applications; in other words, a cluster of networked nodes.

The design-time toolsuite is underpinned by a DSML (an overview is presented in [6]) with tools and generators to automate the tedious parts of software development and provide an analysis framework. The run-time software platform reduces the complexity and increases the reliability of software applications by providing reusable technological building blocks in the form of an operating system, middleware and application management services.

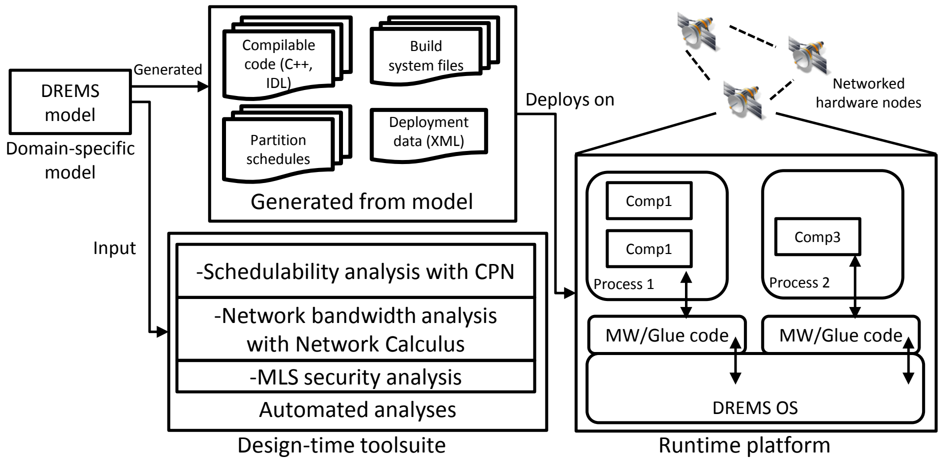


Fig. 1. The design-time toolsuite (left) and the run-time platform (right). The modeling language is used as the basis for both analysis and code generation.

Figure 1 shows a high-level overview of the development process. A model captures several aspects of the system in addition to structure. Models of the software components and interfaces are used to generate compilable skeleton code. Scheduling attributes of processes are used to generate OS schedules. The mapping of software to hardware, along with component interaction descriptions, generates deployment configuration files. Additionally, there exist design-time analysis tools; Section 5 contains more details.

The run-time system, shown on the right-hand side of Figure 1, includes a custom operating system (OS) and middleware (MW). Applications consist of some number of components, hosted inside processes, that communicate through the middleware. Each node in the system is expected to contain the full run-time infrastructure.

3 Integrating textual code with graphical block diagrams

The first integration challenge we describe is integrating textual code inside the graphical modeling language. The use of a textual language was motivated by the fact that the underlying platform uses a component-based software engineering (CBSE) methodology [10]. In CBSE, applications are built from reusable, communicating software components. For our underlying platform, these components are specified using a language called the Interface Definition Language

(IDL), an OMG standardized language to describe component interfaces. From an IDL specification of a component, code generators produce code stubs that are combined with user-written business logic and compiled to produce the actual component.

Thus, inside our DSML, we need abstractions for describing components, which include a suitable representation for IDL. The two main requirements for using IDL inside models were (1) other modeling elements should be able to refer to its properties and attributes, and (2) IDL developed independently from the model should be straightforward to use inside the model. Figure 2 shows the desired workflow.

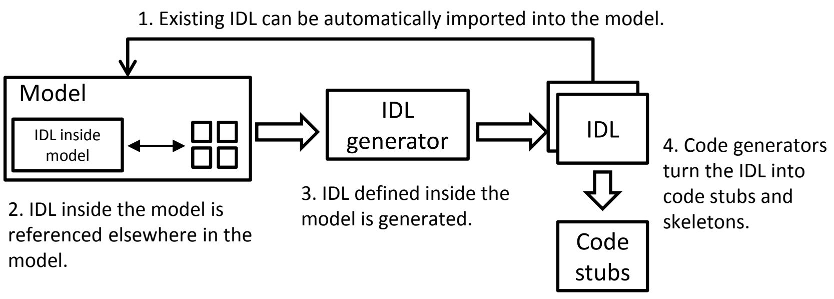

1. Existing IDL can be automatically imported into the model.
Model
IDL inside model
IDL generator
IDL
4. Code generators turn the IDL into code stubs and skeletons.
2. IDL inside the model is referenced elsewhere in the model.
3. IDL defined inside the model is generated.
Code stubs

Fig. 2. Workflow for using IDL in the modeling language.

We had two main choices for how to represent IDL inside our models: a graphical representation or a textual representation. The drawback of using a graphical notation to represent IDL is that it is cumbersome for the user. Essentially, the user builds a graphical representation of the abstract syntax tree (AST) of their IDL. This is especially tedious for designers who already have some familiarity with IDL. The advantage of this approach is that it allows other modeling elements to easily refer to attributes and properties of the IDL.

On the other hand, the advantage of a textual notation is that it is compact, which makes it simple for users to write. It is also easy to import from existing IDL definitions. The drawback of a textual notation is that other elements in the modeling language cannot easily refer to its content. Also, the modeling language itself cannot be used to enforce that syntactically correct IDL is written by the user.

Ultimately, we decided on a combination of a graphical and textual representation. Figure 3 shows the portion of the metamodel (as a UML class diagram) defining the six types of IDL elements represented graphically inside the model; note that they are all subclasses of the abstract class *DataType* and inherit the *Definition* attribute. These are the graphical elements that appear in the modeling language.

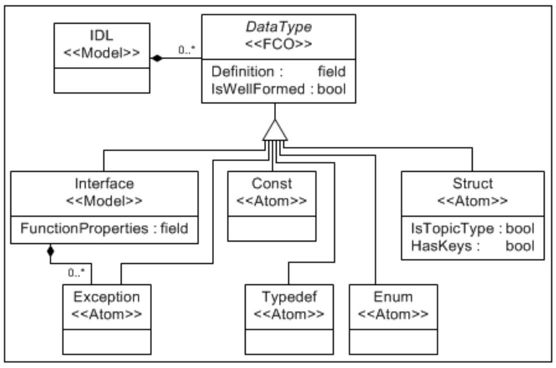
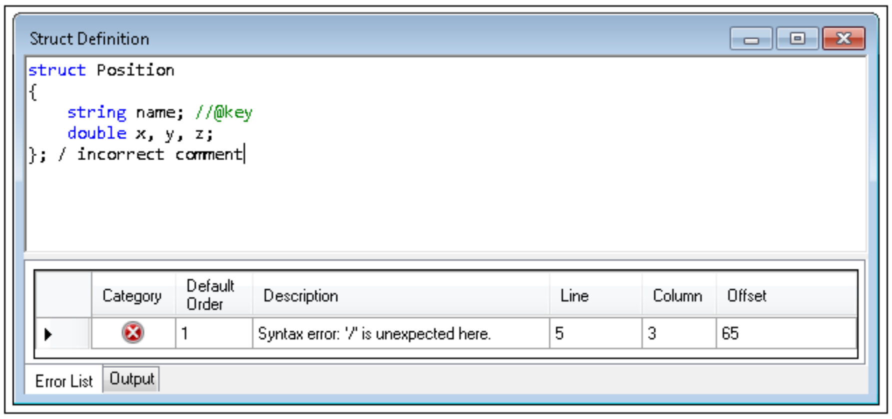


Fig. 3. The types of IDL elements represented graphically inside models.

The integration of the textual IDL language was accomplished by (1) generating an IDL parser using the ANTLR parser generator [12] and (2) creating an add-on to the modeling language using our modeling environment's extension API that uses the generated parser. The overall process is shown in Figure 5. The *Definition* attribute of each graphical IDL element contains its IDL code and is edited using a special code editing window that is provided by the add-on (see Figure 4). The code editor invokes the IDL parser and reports any syntax errors to the user, as shown in Figure 4.


Struct Definition

```
struct Position
{
    string name; //@key
    double x, y, z;
}; / incorrect comment
```

	Category	Default Order	Description	Line	Column	Offset
▶	⊗	1	Syntax error: '/' is unexpected here.	5	3	65

Fig. 4. Screenshot of the IDL code editing window, which was integrated into the modeling language. Errors reported by the parser are at the bottom. In this example, the IDL code is invalid because the comment on the last line begins with a single '/'.

If the IDL code is syntactically correct, then the integrated parser automatically sets attributes of the corresponding graphical IDL element in the model.

The key to the approach is the add-on that is able to (1) parse the textual language, and (2) based on the results of the parser, set multiple attributes on the graphical modeling elements. This is important because it enables attributes defined in the textual language to be integrated into the modeling language.

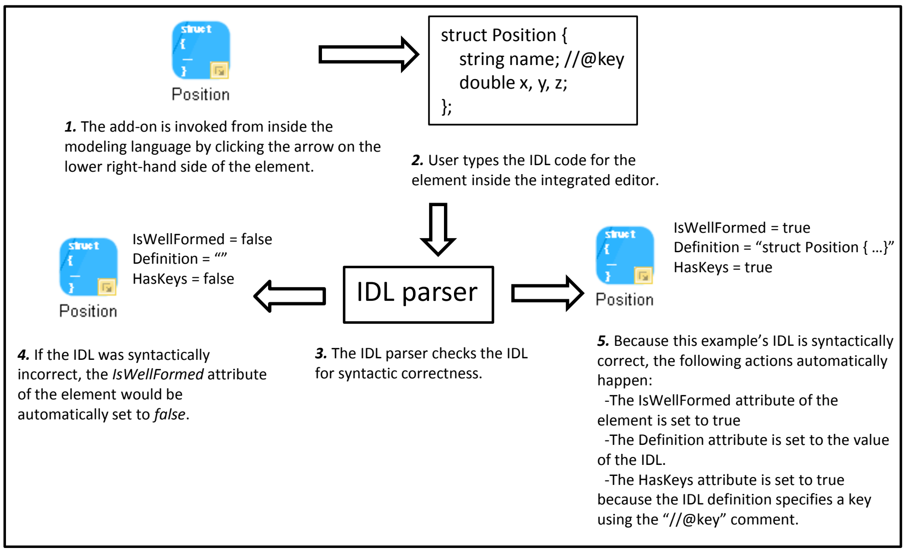

struct {
}
Position
struct Position {
string name; //@key
double x, y, z;
};
1. The add-on is invoked from inside the modeling language by clicking the arrow on the lower right-hand side of the element.
2. User types the IDL code for the element inside the integrated editor.
IsWellFormed = false
Definition = ""
HasKeys = false
Position
IDL parser
IsWellFormed = true
Definition = "struct Position { …}"
HasKeys = true
Position
4. If the IDL was syntactically incorrect, the *IsWellFormed* attribute of the element would be automatically set to *false*.
3. The IDL parser checks the IDL for syntactic correctness.
5. Because this example's IDL is syntactically correct, the following actions automatically happen:
-The IsWellFormed attribute of the element is set to true
-The Definition attribute is set to the value of the IDL.
-The HasKeys attribute is set to true because the IDL definition specifies a key using the "//@key" comment.

Fig. 5. Overview of how the textual IDL parser is integrated into the modeling language. Multiple attributes are set based on the value of the IDL in Step 5.

4 Schedule generation

The next challenge we describe deals with transforming high-level scheduling properties of individual elements into a combined, low-level schedule for the target platform. The motivation for this is that the target DREMS platform uses its own scheduler, namely, a temporal partition scheduler [1] to schedule processes. A partition is a logical group of executing processes in which all processes in the same partition are periodically given exclusive access to the CPU. Exclusive access means that no process from another partition is given access to the CPU during this time. Each partition can contain any number of processes and is defined by two attributes: a *duration* and a *period*. For instance, a partition with a duration of 4ms and a period of 10ms will run for 4ms every 10ms, and during these 4ms the processes it contains will have exclusive access to the CPU.

In order for the target platform to schedule the temporal partitions at runtime so that the period and duration constraints of each are satisfied, it must be given a valid partition schedule. This partition schedule is a periodically repeating set of time slices called *hyperperiods*. The partition schedule specifies when to start each partition relative to the start of the hyperperiod. At the end of the hyperperiod, the same schedule is repeated again.

The multi-paradigm challenge with this is how to transform the process-partition assignment graph as well as the high-level scheduling attributes (a period and duration) of individual temporal partitions into a low-level partition schedule that can be used at runtime by the operating system. This schedule should be a part of the architecture description language because users must know at design-time whether a satisfying partition schedule exists and if so, what that schedule is. However, even for seemingly simple combinations of individual partitions, calculating a schedule by hand can be a non-trivial task. Thus the modeling language needs to hold enough information as well as provide a facility to calculate this schedule automatically and present it to users.

Figure 6 shows our solution using a small example. On the left side of Figure 6 are two processes (*P1* and *P2*), defined inside the software part of the modeling language. These two processes are assigned to two different partitions (*P1* is assigned to *T1* and *P2* is assigned to *T2*) in the software deployment portion of the language. The *schedule calculator* is the solution that provides a bridge between the individual temporal partitions and an integrated partition schedule. It is implemented as an add-on to the modeling language using our modeling environment's extension API and provides a bi-directional interface both to and from the modeling language.

When the schedule calculator is invoked, it queries the model and collects the individual temporal partitions. Next, it formulates the scheduling problem as a constraint satisfaction problem [13]. The constraint satisfaction problem is then given to an off-the-shelf solver (the Z3 SMT solver [3]). If the solver cannot find a solution, then it indicates a conflict with the temporal partitions and the schedule calculator then informs the user so that the temporal partitions can be modified.

If a solution to the constraint satisfaction problem is found, then the scheduling calculator must translate this solution into a partition schedule and insert it into the model. Creating the partition schedule from the scheduling calculator is done using the modeling environment's API which provides programmatic access for setting attributes.

5 Design-time analysis

This section describes three automated analyses that we built for the modeling language: security of communications, software component schedulability analysis and network quality of service (QoS) analysis. Due to space constraints, we only briefly describe each.

The operating system provides security to applications through spatial isolation (processes run in separate address spaces) and temporal isolation (temporal partitions guarantee that processes get a guaranteed portion of processor time). However, these two mechanisms alone cannot guarantee information flow iso-

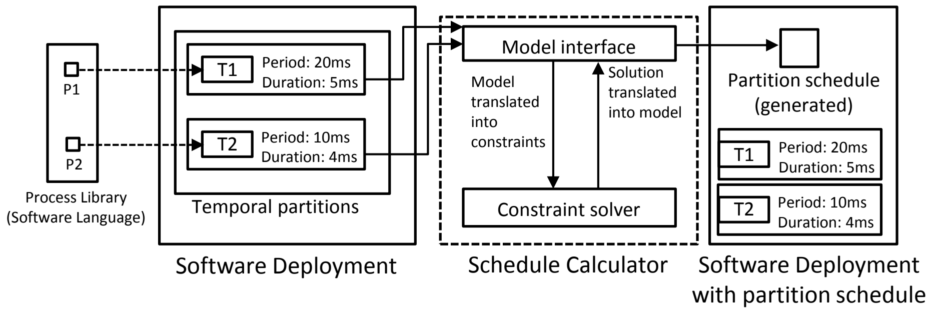


Fig. 6. The schedule calculator transforms individual partition attributes into a combined partition schedule. Processes (P1 and P2) from the software language are assigned to temporal partitions (T1 and T2) in the software deployment. The schedule calculator uses the attributes of T1 and T2 to formulate and solve a set of numerical constraints and generate a valid partition schedule, consisting of repeating sequences of the two partitions, into the model.

lation between applications. This is important because the runtime system is expected to host both trusted and untrusted (i.e., 3rd party) applications. Preventing covert channel communication between applications is also important. Our solution to this problem is a multilevel security (MLS) policy [6] that uses multi-domain *labels*. The modeling language supports the MLS policy with label elements that are attached to the communication endpoints and processes; an automated tool was built to check the compatibility of security labels between communicating processes at design time.

DREMS supports systems whose network quality and connectivity may vary over time. To analyze whether the QoS requirements of processes can be satisfied with time-varying networks, the modeling language supports QoS *profiles* describing the expected network parameters (e.g, bandwidth, latency) over time and QoS *requirements* that specify the network requirements of processes. An automated tool based on the network calculus [14] then analyzes whether the QoS requirements of all processes can be satisfied.

The third analysis relates to the verification of system properties, such as deadline violations by software components. This approach is based on modeling the abstract control flow and timing properties for component operations and then combining them with a formalized model of hierarchical schedulers in the system (the operating system scheduler and the component operation scheduler) to generate an integrated Colored Petri Net (CPN) [11] model. This model can be used to ensure that as long as the assumptions made about the system hold, the behavior of the system lies within the safe regions of operation. For example, the tool can verify that component operations will not cause deadline violations.

Overall, these design time tools provide analyses to both application developers and system integrators to ensure that the system meets the expected requirements and is robust to the runtime constraints imposed by the environment and the infrastructure.

6 Related work

There are several architectural description languages and standards that have similarities to ours, such as AADL [8], OMG's SysML [9], and AUTOSAR [2]. Both AADL and SysML are general-purpose architecture description languages that support abstractions such as software components, hardware and system integration. Our main reason for developing a custom architecture language was the need fo a language closely coupled to the semantics of our target platform to provide a sufficient level of tool automation. This required specific syntax and semantics for modeling language elements like scheduling, component interactions and security; using an AADL annex would have been too complex.

AUTOSAR [2] is a comprehensive standard for vehicle architectures that has a component model similar to ours. The drawback is that the AUTOSAR standard contains very little guidance on design-time analysis and tool support, both of which are crucial for architecture design languages.

There exist various approaches for integrating textual artifacts within graphical modeling languages. XText [7] is a framework integrated into the Eclipse Modeling Framework (EMF) for developing domain-specific languages. From a grammar in the XText grammar format, XText generates a parser that allows a graphical editor and text editor to be simultaneously used to edit a file, with the changes from each reflected in the other. The main difference to our approach is that our add-on sets multiple attributes based on the contents of the text. XText does offer an extension API that could be used to implement similar behavior.

The work in [5] considers the problem of using both textual and graphical notations to modify models of the same language. This is different from our work, which integrates textual code in one formalism with graphical models in another formalism. Language workbenches [15] also offer language composition features that can be used to combine textual and graphical languages for different formalisms.

Our temporal partition scheduler is based on the temporal partitioning method described in the ARINC-653 avionics standard [1]. ARINC-653 systems have been modeled using AADL in [4]. Unlike the ARINC-653 standard, which does not require address separation between processes of the same partition, DREMS allows system integrators to separate components into separate address spaces (called *actors*) within the same partition. The main advantage of our approach is that we can handle faults within related components (within the scope of an actor) with guaranteed isolation. This is not possible with the ARINC-653 standard.

7 Conclusions

This paper presented the multi-paradigm challenges we faced when building an architecture description language for distributed, real-time systems. The first challenge was integrating two different formalisms: textual code and graphical block diagrams. The second was computing a low-level runtime schedule from individual, high-level temporal partitions. The third challenge was integrating design-time analysis.

In cases such as ours, where a significant level of tool automation is required, we believe that a custom architecture language tailored to the semantics of the runtime platform provides a big advantage. The alternative, using a standard architecture language, can require adapations to code generators and automated analysis tools to account for semantic differences between the language and the runtime platform, such as scheduling or security.

Acknowledgments: This work was supported by the DARPA System F6 Program under contract NNA11AC08C and USAF/AFRL under Cooperative Agreement FA8750-13-2-0050. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or USAF/AFRL.

References

1. ARINC Incorporated, Annapolis, Maryland, USA. *Document No. 653: Avionics Application Software Standard Inteface (Draft 15)*, January 1997.
2. Autosar GbR. AUTomotive Open System ARchitecture. `http://www.autosar.org/`.
3. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
4. Julien Delange, Laurent Pautet, Alain Plantec, Mickael Kerboeuf, Frank Singhoff, and Fabrice Kordon. Validate, simulate, and implement ARINC 653 systems using the AADL. In *SIGAda*, SIGAda '09, pages 31–44, New York, NY, USA, 2009. ACM.
5. Luc Engelen and Mark van den Brand. Integrating textual and graphical modelling languages. *Electron. Notes Theor. Comput. Sci.*, 253(7):105–120, September 2010.
6. Tihamer Levendovszky et al. Distributed real-time managed systems: A model-driven distributed secure information architecture platform for managed embedded systems. *IEEE Software*, 31(2):62–69, 2014.
7. Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *SPLASH*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM.
8. P.H. Feiler, Bruce A. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems. In *Computer Aided Control System Design*, pages 1206–1211, 2006.
9. Matthew Hause et al. The SysML Modelling Language. In *Fifteenth European Systems Engineering Conference*, volume 9, 2006.
10. George T. Heineman and Bill T. Councill. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, Reading, Massachusetts, 2001.
11. Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems.* Springer, 2009.
12. Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Bookshelf, 2007.
13. Klaus Schild and Jörg Würtz. Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357, 2000.
14. Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *in ISCAS*, pages 101–104, 2000.
15. Markus Völter and Eelco Visser. Language extension and composition with language workbenches. In *SPLASH*, SPLASH '10, pages 301–304, New York, NY, USA, 2010. ACM.

Integrating Language and Ontology Engineering

Bruno Barroca[‡], Thomas Kühne[*], Hans Vangheluwe[†‡]

[†] University of Antwerp, Belgium
[‡] McGill University, Montréal, Canada
[*] Victoria University of Wellington, New Zealand

Keywords: Ontologies, FTG+PM, Properties, Meta-modeling Environments, Description Logics, Semantic Mapping, Verification spaces

Abstract. Creating new modeling environments has become a relatively low-cost investment thanks to meta modeling environments and language workbenches that can automatically synthesize environments from language specifications. However, the currently existing tools are focused on language syntax and execution/simulation rather than providing means to reason with semantic properties from the real world. It appears that reasoning opportunities as they arise in ontology research (e.g., based on description logics) are currently not exploited in the field of language engineering. In this vision paper, we explore the integration of the hitherto rather isolated areas of language engineering and ontology engineering in order to exploit the potential of using reasoning for models expressed in user-defined languages.

1 Introduction

The automated generation of modeling and programming environments can look back at a long history – starting in the 1980's [1] – of providing means to define the syntax and semantics of languages. Recently, influences from model-driven development made it easier to define semantics more flexibly by means of model-to-model transformations. However, most semantics definitions are still concerned with execution or simulation.

Despite the common trend of increasing the return on investment by improving languages and tools, it appears that the basic 'grammar-ware' principles of meta modeling environments has not changed in over 30 years. However, we can observe that the ability to chain transformations has enabled modelers to learn more about their models – and thus the systems under study – than would have been economically feasible in former days by defining a single transformation towards a particular platform: i.e., generating a compiler. The Formalism Transformation Graph + Process Modeling (FTG+PM) [2] approach from the area of multi-paradigm modeling is the most explicit example for this style of modeling in which the output of one transformation is used as the input for another one.

The crucial aspect of this approach is that from an initial set of models, we can have orthogonal transformations with totally different intents. In particular, each path in the transformation graph is designed to, from the initial set of models, illuminate a particular set of properties of the system under study/development, e.g., regarding termination, liveliness or even safety [3].

In this context, the main trend from the language engineering area, is to establish a trace between the property checking results (in the leaves of a given transformation chain) and the original model so that, for instance, the results can be suitably interpreted by the modeler in a particular domain. In this paper, we test yet another approach on this problem: we argue that one should explicitly associate the properties of interest as an ontological type (i.e., introduce ontological types explicitly as first-class citizens), and then test/check its conformance by means of transformation(s) to the respective semantic domains w.r.t. those properties.

To this end, we explore at the conceptual level, the integration of two hitherto unrelated areas: First, language engineering, with its support for model representation and transformation, and second, ontologies with their rich tool set for describing and inferring properties. In Section 2 we introduce a railway system, that we will use as a running example to illustrate both challenges and our proposed solution. In section 3, we present the context of the problem and the ingredients that contribute to our proposed solution which is presented in Section 4. Finally, we conclude in Section 5.

2 Case Study: The Railway System

Railway transportation is only one of the many sectors that have benefited from the application of computer automation techniques ranging from early automated railway control to train scheduling [4].

2.1 General Requirements

First and foremost a railway system needs to support transport of people or goods between endpoints. This basic function, however, is accompanied by additional constraints and boundary conditions. For example, a railway system must be affordable by end users and profitable for operators, placing certain constraints on energy consumption, personnel cost, etc. Moreover, as with any system where lives and or loss of large monetary investments are at stake, safety is a critical concern. Preventing trains from derailing or colliding is crucial for a railway system.

We observe that it is possible to separate the requirements into what happens, i.e., trains moving between endpoints, and how it happens, e.g., without collisions. Generally speaking, we can distinguish between the mechanics of a system (i.e., its execution semantics), and properties that can be associated with the mechanics (with verification semantics).

2.2 Execution Semantics

In the following, we will use a language to define a railway network (c.f. [5]) to describe the system mechanics. A railway network comprises a collection of railway tracks organized into sections and connecting different train stations. In order to avoid collisions, only one train is allowed in one section at a time. Signals control when trains may enter and leave sections. The combination of signals, their control logic, and track switches is called the interlocking system and its purpose is to prevent conflicting train movements.

The behavior of such an interlocking system is specified in a control table which embodies information about conflicting train routes. A formal description of the control table in combination with information about the physical railway layout could be regarded as a prescriptive model for the execution behavior of the interlocking system.

2.3 Verification Semantics

While one could use the execution behavior of an interlocking system to simulate the latter, this would only enable the detection of errors in the interlocking system through testing. In order to gain certainty regarding the absence of any potential conflicting train movements, it is necessary to formalize the respective safety requirement and ensure that it is satisfied by the interlocking system under study.

Somsak achieves the automated verification of an interlocking system by translating it into a colored petri net (CPN) which can then be automatically verified using CPN tools, a Petri Net model checker [5]. Each railway track represents a resource that should only be used by one train in order to avoid collisions, and can therefore be directly mapped to a place in the CPN formalism. Despite the fact that Somsak only verified three scenarios, this work is a good example of how domain models (such as interlocking systems) can be automatically and systematically verified by reusing effective algorithms developed in other domain such as CPNs.

Note that safety concerns are just one example for the utility of verification semantics. The railway operators may also be interested in the economics of running the railway system and thus may want to convince themselves that certain track layouts will achieve a desired throughput. In general, many such properties will have to hold for a given system and will be required to occur in conjunction, i.e., the system should be both economical to run and be safe to use.

3 Linguistic Models versus Ontologies

It is worthwhile pointing out that both execution and verification represent two different interpretations of one and the same model. Two different semantic domains are used in order to answer different sets of questions regarding the same model. The execution semantics tells us what may happen for a given interlocking system, while the verification semantics tells us whether the entirety of all that may happen for a given interlocking system satisfies a particular property.

3.1 Natural vs. Verification Semantics

Intuitively, it appears that the execution semantics could be regarded as the natural "*meaning*" of an interlocking system, while the verification semantics appears to subject the interlocking system to a test. However, intuition is not reliable and in the following we thus strive to identify intrinsic differences between these two kinds of semantics.

Both kinds of semantics obviously fulfill the minimal criterion of being mappings that are functional and total [6]. At first sight, it may furthermore appear as if the two different kinds of semantics were on the same footing and thus interchangeable. In our example, there appears to be a symmetric relationship between

- the subset of safe interlocking systems,
- the subset of executions of safe interlocking systems, and
- the subset of elements in the verification space that satisfy the property "safe".

On closer examination, however, it becomes apparent that the roles of the two different semantics cannot be swapped as there is an asymmetric dependency. On the one hand, the space of all possible execution behaviors, as defined by the execution semantics, can be *partitioned* by the *properties* it may satisfy. In our example, the set of safe interlocking systems can be derived from the subset of "safe" elements in the verification space and only then find a subset of safe interlocking system executions in the overall set of all possible executions of interlocking systems. On the other hand, it is not true that the verification space can be usefully partitioned from the execution semantics. In our example, this is trivial, since following our assumptions, we have no other way to decide about the "safety" of a given interlocking system execution.

This asymmetric dependency may constitute an obstacle for purposes of our conceptual integration. Fortunately, there exist technologies are a good fit in order to

(a) enable the anchoring of natural semantics to models, and
(b) support the definition and organization of semantic properties

3.2 Language Engineering

A linguistic type model – often referred to as "metamodel" – is ideal for enabling the attachment of natural semantics to models. Following our example, if we define a "Railway DSL" grammar and well-formedness constraints combined in the form of a linguistic type model, then we can determine the structure of all elements that may occur in an interlocking system model. The linguistic type model is thus ideally suited to be used as the basis for transformation definitions such as semantic mappings.

Existing metamodeling environments can synthesize complete environments for configuring scenarios – such as the topology of the railway network, positioning of signaling devices, the control table logic, etc. – and even use the linguistic type model as the basis for subsequent transformation (language) definitions [7].

Conformance of a model to a linguistic type model is typically granted by construction. In special cases of "freehand" (as opposed to syntax directed) editing and language evolution it may be necessary to check whether a model conforms to a linguistic type model, but in most cases the model is the result of using the linguistic type model as a generator, e.g., by structured editing or model generation.

3.3 Ontology Engineering

An ontology comprises and organizes a number of concepts and may use description logic to express concept properties and relationships. While ontologies in general contain very different kinds of concepts – of which so-called moment types (properties) are only a particular subset – the technology associated with them appears to be eminently suited to accommodate a taxonomy of properties derived from the various verification semantics.

The most popular ontology language is the Web Ontology Language (OWL), which along Common Logic (CL), and the Resource Description Framework (RDF) is included in the Ontology Definition MetaModel (ODM) proposed by the Object Management Group (OMG). The most advanced kind of reasoning in ontology engineering is achieved by means of description logics (DLs). DLs is a family of knowledge representation languages, which are used as logical formalisms to support reasoning, e.g., in the context of the Semantic Web. DLs are less expressive than first order logics, hence they are amenable to decidable and efficient reasoning mechanisms.

The most interesting feature of these languages is their ability to infer new knowledge, i.e., make implicit knowledge explicit. DLs language extensions span from introducing the notion of time as partial-order relations such as in temporal extensions [8]; introducing vagueness or incomplete concepts, such as in fuzzy logics extensions [9]; introducing the notion of concepts with probability values [10]; to introducing the ability to express possibility of event occurrence, such as on possibilistic extension [11].

Conformance of a model to an ontological type, in contrast to linguistic type model conformance, is never granted by construction. A conformance check always requires the application of a certain interpretation – i.e., a semantic mapping whose choice depends on the specific property that is to be validated against – and then the subsequent ascertainment of whether or not the property (or properties) associated with an ontological type hold(s) for the element that is the result of the semantic mapping.

In general, ontological types are agnostic to the particular domain they are applied to. For instance, in our example the safety property essentially embodies "*absence of collisions*" and could also be applied to a assembly line scenario in which workpieces are transported and merged by conveyor belts. A different semantic mapping would be required from an assembly line model into the verification space, but ascertaining whether the collision-free property holds based on the respective element in the verification space by using a Petri Net model checker would be identical to the railway example. For that, just consider that instead of trains to Petri Net tokens, we would now have products to Petri Net tokens; and instead of railway tracks to Petri Net Places, we would now have conveyor belts to Petri Net Places; and the property to check is whether there exist some situation in the entire set of possible situations where there are two Tokens in the same Place.

While the above described language agnosticism allows ontological types and their definition to be reused, it also means that it is not straightforward to associated further ontological types to a model: for instance, consider the situation where the only classification information known for a given model is a single ontological type. The latter can classify models from a whole range of languages so it is not clear which semantic mapping (of many potentially applicable) would have to be applied in order to validate it (the model) against another ontological type. In contrast, if the linguistic type of a model is known (typically by-construction in most modeling environments) then it is trivial to determine which semantic mappings are available for it.

4 Integrating Languages with Ontologies

So far we have identified a particular subkind of semantics with associated properties that gives rise to ontological types (i.e., verification semantics), and furthermore observed that linguistic types complement the latter. In the following, we describe what a complete framework that encompasses language engineering and ontology engineering could look like. Such an integrated approach is clearly desirable since it allows us to not execute, simulate, and test models w.r.t. the respective referents in the real world, but also to go beyond and reason about their properties.

For instance, in our example, we could specify a railway ontology based on notions concerning the economic efficiency and/or safety of railway topologies. We could then infer the most economical, but also safe, train schedules using for instance design space exploration techniques.

A complete framework for modeling with both linguistic and ontological types, would typically use multiple formalism integration – in the style of FTG+PM – and would have a number of desirable features:

(a) While transforming models across different formalisms, we should be able to trace the properties that are being lost, preserved or created during such transformations;
(b) It is rather likely that ontological types will be reusable across domains (e.g., safety in the context of rail transportation and collision-free schedules

in the context of assembly lines). This will greatly increase the return on investment for developing the respective analysis approaches.

4.1 Conceptual Framework

We start our conceptualization from the well known model developed in [12] that explores how linguistic symbols are related to the objects they represent.

Based on this conceptualization, our first attempt is presented in Figure 1 on the left, where 'System' replaces the real world object, the 'Ontological Type' replaces thoughts (also called reference or Concept), and finally the 'Linguistic Type' replaces symbol (also called 'Sign'). Moreover, we present in the center of the triangle, the model element, which in the modeling world is the first-class entity that relates the System under study/development with both Ontological and Linguistic types.

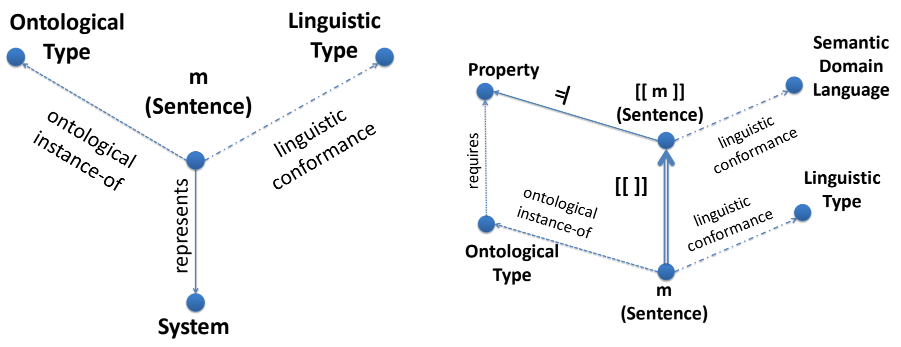


Fig. 1. A base conceptual framework (on the left), and a refined version (on the right).

The Figure 1 on the right, extends our first attempt, where we unfold the 'ontological instance-of' relation between a given model m and its ontological type (denoted in the Figure as 'Ontological Type'). Notice that we start from a situation where although linguistic types are taken for granted (i.e., by-construction) in modeling environments, the same is not true for ontological types. The unfolding of this relation is therefore done by means of a semantic mapping $[\![ ]\!]$ to a verification platform where a set of properties (denoted in the Figure as 'Property') relevant w.r.t. that given 'Ontological Type' can be verified. This relevance is stressed in the Figure with the relation 'requires', which means that a given 'Ontological Type' depends or includes as part of its intension, a given set of properties.

One way of transversing the left side of this diagram is therefore, by assuming that we know which Properties are required by a given Ontological Type, so that we can choose the most appropriate verification platform and devise a semantic mapping $[\![ ]\!]$ which can be realized by means of a model transformation. This model transformation is then able to automatically transform arbitrary models m given that they do conform (in the linguistic sense) to a given meta-model (or grammar) depicted in the Figure as 'Linguistic Type'. The resulting transformed model denoted as $[\![m]\!]$ by construction of the transformation itself should also conform linguistically to the language from the verification platform (denoted as 'Semantic Domain Language'). The circle in this diagram is closed by the satisfaction relationship (denoted as $\models$) between $[\![m]\!]$ and the Property is established, which means that we can now conclude that the given model m is indeed an ontological instance of the given Ontological Type.

It is important to mention here that in practice, there might be several different properties that a given Ontological Type may require. Therefore, we can expect to have several different orthogonal semantic mappings $[\![]\!]$ using possibly different kinds of verification mechanisms (and their associated semantics) in order to be able to finally establish that ontological instance relationship between a model and an ontological type.

4.2 Framework Instantiation

After describing the general terms of our conceptual framework, we will now look at its instantiation in the particular case of our railway transportation example.

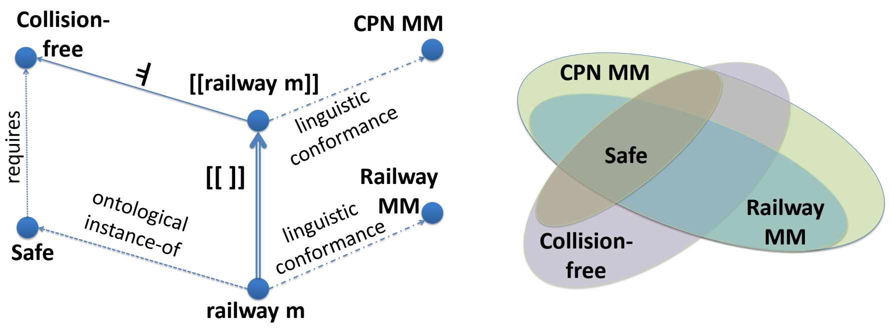


Fig. 2. Instantiation of the conceptual framework on the railway transportation example (on the left), and the respective solution-partition space (on the right).

At the bottom left of Figure 2 (on the left), we see that the system under study is an interlocking system in conjunction with a train scheduling system. The model *railway m* at the bottom right, represents this system. The linguistic type of *railway m* is the meta-model "Railway MM" to which it conforms syntactically. At the top right is an interpretation of *railway m* that was chosen in order to enable a reachability analysis. Such an interpretation can be achieved by means of a model transformations of model *railway m* into a corresponding Colored Petri Net $[\![railway\ m]\!]$. The latter also has its own linguistic type (the meta-model "CPN MM") to which it conforms syntactically.

The analysis of $[\![railway\ m]\!]$ is performed by unfolding the complete state-space (e.g., by using the model-checking engine of a CPN tool). This state-space is then queried for a property "$Collision - free$", e.g., checking whether any collision scenarios exist. This property "$Collision - free$" is depicted at the top-left. Model $railway\ m$ can be said to be an ontological instance of the ontological type "Safe" (i.e., be called "safe") if and only if its interpretation $[\![railway\ m]\!]$ satisfies property $Collision - free$.

Finally, we show in Figure 2 (on the right) the solution space partitioning complements the commuting diagram on the left. Notice that solutions are systems in the real world. On the one hand, given the 'requires' relation, the set of safe solutions is a subset of collision-free solutions: in other words, 'safe' is a stronger concept which may depend not only on being collision-free but also from other concepts. On the other hand, given the semantic mapping $[\![]\!]$, the set of solutions represented by models conforming to the Railway MM will always be a subset of the total set of the solutions represented by models conforming to the Colored Petri Nets CPN MM. Finally, given the above sets, we conclude that the set defined by the commuting diagram on the left is defined by the intersection of all of the sets defined on the right: i.e., a solution which has both a Railway model representation and its respective CPN representation, which is proven to be collision-free, and therefore an ontological instance of type Safe.

5 Conclusions

In this vision paper, we have proposed to integrate the technological spaces of language engineering and ontology engineering in a manner that strengthens language engineering to include concepts and techniques from ontology engineering. While modelers have already been successfully checking models for semantic properties with various approaches in the past, to the best of our knowledge our approach is the first to introduce ontological types with semantic properties as first-class citizens and proposes to arrange them in taxonomies, thus exploiting semantic relationships.

We have identified differences between ordinary, "natural" semantics and verification semantics, observing that semantics of the latter kind are agnostic to languages and partition sentences in a language. We have chosen to refer to types that achieve such partitions as "ontological types", referencing the fact that they are based on semantic properties, rather than on syntactic conformance.

We could only touch upon the potential of using model exploration and inference engines to leverage a taxonomy of ontological types to a tool that supports the identification of models that satisfy properties in multiple dimensions.

Nevertheless, in this paper we have contributed towards finding optimal ways of combining linguistic type models with ontologies w.r.t. previous attempts [13, 14], which are rather more focused (with again) more syntactic issues than conceptual ones. We also believe that our proposal is suitable to shed further light on the most precise characterization of ontological vs linguistic classification as presented in [15]. However, our use of the prefix "ontological" should not be construed as meaning that our notion of "ontological types" is exactly the same as the "ontological types" discussed in [15]. While there is certainly large overlap, it remains to be seen whether our notion full subsumes the other, or rather represent as subset of that may be best characterized as "moment ontological types".

References

1. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A syntax-directed programming environment," *Commun. ACM*, vol. 24, pp. 563–573, Sept. 1981.
2. S. Mustafiz, J. Denil, L. Lúcio, and H. Vangheluwe, "The FTG+PM framework for multi-paradigm modelling: An automotive case study," in *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, MPM '12, (New York, NY, USA), pp. 13–18, ACM, 2012.
3. L. Lucio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukšs, "FTG+PM: An integrated framework for investigating model transformation chains," in *SDL Forum*, pp. 182–202, 2013.
4. J. Pachl, *Railway Operation and Control.* VTD Rail Publishing, 2nd ed., 2009.
5. S. Vanit-Anunchai, "Modelling railway interlocking tables using coloured petri nets," in *Coordination Models and Languages* (D. Clarke and G. Agha, eds.), vol. 6116 of *Lecture Notes in Computer Science*, pp. 137–151, Springer Berlin Heidelberg, 2010.
6. D. Harel and B. Rumpe, "Meaningful modeling: What's the semantics of "semantics"?," *IEEE Computer*, vol. 37, no. 10, pp. 64–72, 2004.
7. T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Systematic transformation development," *Electronic Communications of the EASST*, vol. 21, 2009.
8. A. Artale, E. Franconi, M. Mosurovic, F. Wolter, and M. Zakharyaschev, "Temporal description logic," in *Handbook of Time and Temporal Reasoning in Artificial Intelligence*, pp. 96–105, MIT Press, 2001.
9. U. Straccia, "A fuzzy description logic for the semantic web," in *Fuzzy Logic and the Semantic Web. Capturing Intelligence, Elsevier (2006) 73–90.*
10. Z. Ding and Y. Peng, "A probabilistic extension to ontology language OWL," in *In Proceedings of the 37th Hawaii International Conference On System Sciences (HICSS-37), Big Island*, 2004.
11. G. Qi, J. Z. Pan, and Q. Ji, "A possibilistic extension of description logics," in *In Proc. of DL'07, 2007.*
12. C. Ogden and I. A. Richards, "The meaning of meaning: A study of the influence of language upon thought and of the science of symbolism.," *8th ed. 1923. Reprint New York: Harcourt Brace Jovanovich*, 1923.
13. B. Henderson-Sellers, "Bridging metamodels and ontologies in software engineering," *J. Syst. Softw.*, vol. 84, pp. 301–313, Feb. 2011.
14. E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, and M. Wimmer, "Lifting metamodels to ontologies - a step to the semantic integration of modeling languages," in *In Proceedings of the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006*, pp. 528–542, Springer, 2006.
15. T. Kühne, "Matters of (meta-) modeling," *Software and Systems Modeling*, vol. 5, no. 4, pp. 369–385, 2006.