Christophe Jacquet, Daniel Balasubramanian, Edward Jones, Tamás Mészáros, editors

Proceedings

7th International Workshop on Multi-Paradigm Modeling MPM 2013

co-located with Models 2013

Miami, Florida, 30 September 2013

To contact the editors:

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

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

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

Edward Jones
Google, Inc.

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 7th edition took place on 30 September 2013 and was co-located with MODELS'13 in Miami.

Out of the 10 papers submitted and reviewed by at least three members of the program committee, 5 were selected for oral presentation and 2 as posters. About 30 participants attended this edition of the workshop. A vote was cast among the participants of the workshop to elect the best presentation. Markus Voelter's presentation was chosen unanimously.

In addition to the presentation of the selected papers from the technical program, MPM'13 featured an invited presentation by Bernhard Rumpe who talked about compositional multi-paradigm models for software development.

This volume contains versions of the selected papers that the authors had the opportunity to enhance after the workshop and the fruitful discussions that occurred during the whole day. The papers where collected using the EasyChair conference system, formatted according to the LNCS style, and assembled using pdfLaTeX and the pdfpages package.

December 13, 2013
Gif-sur-Yvette

Christophe Jacquet
Daniel Balasubramanian
Edward Jones
Tamás Mészáros

Towards Compositional Domain Specific Languages

Andreas Horst, Bernhard Rumpe

Software Engineering
RWTH Aachen University, Germany
http://www.se-rwth.de/

1 Introduction

The deployment of Domain Specific Languages (DSL) and in particular Domain Specific Modeling Languages (DSML) is becoming more and more prominent in various domains. In order to cope with the complexity of the realization of DSLs, common and well-established methods of software engineering such as modularization and reuse need to be adapted and applied for DSLs. This has already been noted in [2] when the emerging DSL era was still closely akin to compiler theory.

As stated in this work, compositionality of DSLs can take place at several dimensions. Various contributions in this field of ongoing research reflect this and only a brief overview is given below. One form of DSL composition is the syntactic embedding such as embedding DSLs in GPLs as described in [3, 4]. In [5] a family of DSMLs are used for the generation of web information systems. There the composition is carried out via the joint usage of several languages each with their own artifacts and hence no syntactic embedding. Other contributions in the area consider the composition of the models expressed in DSLs as a constructive model transformation [1] and examine the effects of DSL composition at the infrastructure level [9] (e.g., syntax aware editors etc.). The DSL framework and workbench MontiCore [7, 8, 10, 11] was designed and realized particularly with respect to compositionality at various dimensions [6, 12].

Compositionality is of special interest if models of different modeling paradigms - and hence expressed in different languages - need to be combined while at the same time retaining their specific semantics. Whenever the different modeling paradigms are integrated, it can be observed that each paradigm is equipped with its own modeling language and that therefore such a paradigm integration is always also a model language composition. This holds for the composition of structural and behavioral languages as well as for the composition of languages with synchronous or real-time communication and event triggered asynchronous models, etc. In the following the dimensions of such compositions are discussed in more detail.

2 Compositional Language Definition

The major rationale of a DSL is its specificity. One could argue that therefore each DSL has to be defined for the specific use case, i.e., the target domain. However, there usually exist common parts being used in various DSLs. This also holds for DSLs serving different paradigms as usually names, types, variables and often signatures are shared.

Thus the DSL development process benefits from a library based approach. Common language fragments can be provided as a library. The definition of concrete DSL then imports, inherits or embeds the required common language definition components (e.g., in form of grammar nonterminals).

Furthermore, features such as checking of context conditions and especially type correctness (i.e. semantic analysis) and other language infrastructure components (e.g., parser, abstract syntax tree (AST)) of a DSL need to be reusable in a reasonable manner. This requires a thoughtful design of the Application Programming Interface (API) the DSL infrastructure is based upon particularly with respect to compositionality.

3 Compositional Modeling

Often it is necessary or at least helpful to decompose a larger description into several artifacts. This capability is the foundation of modularity and reusability and requires the DSL infrastructure to feature processing of models distributed over individual artifacts just as most GPL compilers can process source files in a rather independent and incremental manner. DSL infrastructures hence have to support model artifact dependencies and thus some sort of model path. This mechanism should also allow to incorporate libraries.

For a simple DSL, this compositional modeling can basically be achieved by splitting models and distributing the resulting fragments over several artifacts. Typically the resulting artifacts each encapsulate a specific part of the overall model and respectively exhibit an explicit interface other artifacts can depend on. Therefore DSLs supporting compositional models necessarily have to provide encapsulation, interfaces and imports. This of course greatly impacts the design of the DSLs.

Apart from this rather straightforward case, the composition encompassing models expressed in various DSLs - potentially even with differing modeling paradigms - yields more complex requirements. For this to work, the aforementioned infrastructure (i.e., context conditions, AST, editors) needs to be capable of being glued together to perform all desired and required tasks. The particular challenges of this complex scenario of compositional modeling across language - and potentially even modeling paradigm - boundaries is based on the following dimensions of composition:

- Syntactic: The syntactic dimension describes how the composition of models - in particular expressed in different DSLs - looks like (e.g., textually embedding, split among artifacts, graphical vs. textual etc.).
- Context Conditions: particularly complicated is the dimension of context conditions that spread across the various languages being deployed together; where e.g. types are shared.
- Semantic: The semantic dimension is about the meaning of the individual model fragments and the meaning of their composition. As an example consider the composition of behavioral models (e.g., Statecharts) with structural models (e.g. object diagrams); what does such a composed model express?
- Technical: The technical dimension deals with the tooling infrastructure of the composition. This for instance determines whether the different models can be processed incrementally and/or individually.

- Methodical: Compositionality provides the ability to decompose a problem and to solve it in parts. A good method can and must take decomposition into consideration.
- Organizational: The decomposition of the problem also yields the possibility to have developers solve particular sub-problems in parallel. This allows to organize the team according to the particular composition of the models. Indeed in conventional software engineering - especially for large projects - the organization of the development team is typically based on the problem/product architecture and component structure.

These considerations show that the possibility to decompose a model into several fragments potentially spread across different artifacts and expressed in different languages with clearly defined interfaces greatly influences the development process.

Typically the model composition boils down to the transport of names and related information in the interfaces between the artifacts each encapsulating a part of the composed model. Names are the primary mechanism to refer to when importing some concept from another artifact. Names come with a lot of related information which includes types, method signatures, etc. But most importantly a name needs to be equipped with the kind of model element it represents, e.g., a method, an attribute, a state, an activity; i.e., the respective concept of the DSL. In behavioral languages it is usually necessary to provide some knowledge about behavioral dependencies, such as order dependencies of messages, maximal waiting time before a timeout is executed within the answer awaiting sender, etc. A different example is the composition two models, one being a class diagram and the other an OCL invariant. There the name of a class used in the invariant determines which attributes are valid to be used in the OCL invariant. This name based dependency is independent from the actual form of syntactic composition, i.e., it does not matter whether the two models are expressed in separated artifacts or combined in one artifact using language embedding.

4 Compositional Generators

In practice it is of interest to defer the actual execution of model composition to a later phase of the development or respectively compilation process. This means that while the semantic composition is well known during the creation of the models, the actual composition takes place in a later phase. This deferring of the composition is a major achievement of modern programming languages. Taking the GPL Java as an example, it is well known how classes are combined together, but the actual technical composition - namely the linking - is conducted later on (i.e., there is no source code being copied into a single monolithic source artifact). Instead each source artifact containing a class definition is being compiled independently and only when starting the program these compiled classes are then linked together.

Transferring this idea to the field of compositional modeling and in particular code generators, this means that models are not composed together directly, but the individually generated code will later be linked together. Especially in the case of heterogeneous modeling languages, it is an obvious consequence that a compiler infrastructure is needed which provides a modular compilation unit for each of the individual languages.

The implementation of such compilation units should be independent of other generators, because only then the composition of DSLs and their paradigms can be carried out in a rather flexible way with respect to code generation.

As an example consider a generator for JPA compatible Java implementations of a class diagram. A second generator creates a graphical web information system out of class diagrams which enables users to explore data structures. A third generator adds state to the objects described by Statecharts. All generators should be usable independently but also easily be composable. Now consider that for example the JPA generator creates Java classes with a specific constructor with parameters while hiding the default constructor. If all three generators are to be used together the two other generators have to take the JPA generator's behavior into account in order to produce valid Java code; i.e., they need to use the JPA classes with the correct constructor and in particular cannot assume the availability of the default constructor. Ideally this dependency is handled in a transparent way not obstructing the independence of each generator individually.

Although partially solved for certain instances, a general solution for the problem of a flexible composition of generators is an ongoing research task. It is to be examined in which way generators can be combined using a suitable interface. In the example above, it would be desirable to have the JPA generator provide the information necessary to use the generated domain model classes which in turn can then be correctly used by the Statecharts and the web information system generators (i.e., by the code generated by these generators).

Of course when composing generators, it is not only necessary to have composable interfaces on the generator level, but also to ensure that the generated results are semantically consistent and thus compositional too. Therefore it absolutely makes sense to first solve composition of multi-paradigm models respectively their languages semantically, before this is implemented in generator tools.

References

1. Bezivin, J., Bouzitouna, S., Fabro, M.D.D., Gervais, M.P., Jouault, F., Kolovos, D.S., Kurtev, I., Paige, R.F.: A Canonical Scheme for Model Composition. In: Verlag, S. (ed.) Proceedings of the Second European Conference on Model-Driven Architecture (EC-MDA) 2006. Bilbao, Spain (July 2006)
2. Bosch, J.: Delegating Compiler Objects: Modularity and Reusability in Language Engineering. Nordic J. of Computing 4, 66–92 (1997)
3. Bravenboer, M., de Groot, R., Visser, E.: MetaBorg in Action: Examples of Domain-specific Language Embedding and Assimilation using Stratego/XT. In: Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05). Braga, Portugal (July 2005), `http://www.cs.uu.nl/~visser/ftp/BGV05.pdf`
4. Bravenboer, M., Visser, E.: Designing Syntax Embeddings and Assimilations for Language Libraries. In: 4th International Workshop on Software Language Engineering (2007)
5. Dukaczewski, M., Reiss, D., Rumpe, B., Stein, M.: MontiWeb - Modular Development of Web Information Systems. In: Rossi, M., Sprinkle, J., Gray, J., Tolvanen, J.P. (eds.) Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09) (2009)
6. Grönniger, H., Rumpe, B.: Modeling Language Variability. In: Calinescu, R., Jackson, E. (eds.) Foundations of Computer Software. No. 6662 in LNCS, Springer, Redmond, Microsoft Research, Mar. 31- Apr. 2 (2011)

7. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Tech. Rep. Informatik-Bericht 2006-04, Software Systems Engineering Institute, Braunschweig University of Technology (2006)
8. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore: a Framework for the Development of Textual Domain Specific Languages. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume. pp. 925–926 (2008)
9. Kats, L.C.L., Kalleberg, K.T., Visser, E.: Domain-Specific Languages for Composable Editor Plugins. In: Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009) (April 2009)
10. Krahn, H.: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering. Ph.D. thesis, RWTH Aachen University (2010)
11. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a Framework for Compositional Development of Domain Specific Languages. International Journal on Software Tools for Technology Transfer (STTT) 12(5), 353–372 (September 2010)
12. Völkel, S.: Kompositionale Entwicklung domänenspezifischer Sprachen. Ph.D. thesis, TU Braunschweig (2011)

A Domain-Specific Language for Dependency Management in Model-Based Systems Engineering

Ahsan Qamar[1], Sebastian Herzig[2], and Christiaan J. J. Paredis[2]

[1] KTH-Royal Institute of Technology, Stockholm, Sweden
{ahsanq}@kth.se

[2] Georgia Institute of Technology, Atlanta, Georgia, USA
{sebastian.herzig,chris.paredis}@me.gatech.edu

Abstract. The varying stakeholder concerns in product development today introduces a number of design challenges. From the perspective of Model-Based Systems Engineering (MBSE), a particular challenge is that multiple views established to address the stakeholder concerns are overlapping with many dependencies in between. The important question is how to adequately manage such dependencies. The primary hypothesis of this paper is that modeling dependencies explicitly adds value to the design process and in addition supports consistency management. We propose a domain-specific language called as the Dependency Modeling Language (DML) to capture the dependencies between multiple views at the appropriate level of abstraction, and utilize this knowledge to support a dependency management process. The approach is illustrated through a dependency model between three views of a robot design example. In addition, we discuss how to analyze dependency graphs for consistency checking, change management, traceability and workflow management.

Keywords: Dependency Modeling Language, Domain Specific Modeling Language, Model Based Systems Engineering, Consistency Management, Change Management.

1 Introduction

Contemporary product development is a complex process. Primarily, this is the case due to a large number of stakeholders being involved, all of which have varying and overlapping concerns. Therefore, adequate methods to manage the consequential conflicts are required. In this sense, the design of mechatronics is a particularly interesting case, since stakeholders from a very diverse set of disciplines are involved. This makes good decision making very challenging. To support a model-based mechatronic design process, different viewpoints are defined, each supported by one or more modeling views. Naturally, multiple viewpoints are supported through multiple modeling languages, where the overlapping stakeholder concerns lead to *dependencies* between the established views. Traditionally, the dependencies are managed in an ad-hoc fashion by mainly relying on the communication between the stakeholders. However, ad-hoc dependency management can prove to be ineffective, especially for complex and large scale systems where there could be a large number of such dependencies.

In earlier work, consistency management was explored in the context of engineering design, and a classification of several distinct type of inconsistencies was identified [1]. It was concluded that no consistency check can ever be complete and that only some inconsistencies can be identified, that too only in the information captured explicitly and formally. Dependencies are interesting because they could be the cause of the arising inconsistency, and hence explicit knowledge of dependencies is vital for consistency management. However, this is not a trivial task since adequate support in terms of a modeling language and a supporting tool for dependency management is currently lacking. In this paper, we present a modeling language to help build a model of dependencies.

The fundamental question to answer is whether it is valuable to model dependencies in contrast to current approaches where dependencies are not captured formally. The work reported in this paper builds on the hypothesis that modeling dependencies adds value to the design process. The value can be measured in terms of support for consistency management, change management, ensuring traceability and managing the design process workflow, each of which can be supported by the dependency modeling approach presented in this paper.

The remainder of this paper is organized as follows: Section 2 builds a notion of dependency. An example use case is described in Section 3. Section 4 introduces a Domain-Specific Modeling Language (DSML) for capturing dependencies, which is illustrated through the example use case in Section 5 and Section 6. Section 7 presents the related work and is followed by a discussion in Section 8. Section 9 presents conclusions and possible future work.

2 Notion of Dependency

Rational Design Theory (RDT) [2] establishes a theoretical foundation for *Artifacts*, *Properties*, *Concept Selection* and *Concept Evaluation*. Based on RDT and on Hazelrigg's decision-based design framework [3], we argue that two types of properties are prevalent in design: one is used to describe constraints (specification), whereas the other is used to communicate the designer's belief regarding the value of the property (a prediction based on a given specification). We call specification properties *Synthesis Properties* (SP) and prediction properties *Analysis Properties* (AP).

To describe an artifact, there could potentially be an infinite number of properties spread across multiple views. In this paper, the term *dependency* refers to a type of model capturing the relationship between the values of input and output properties (regardless of the view they belong to). This is somewhat different to how dependencies are defined in UML [4], where they represent a relation and express the need for a particular element to exist. For example, if an element A depends on element B, and B no longer exists, A is no longer specifiable. In our case, removing a dependency does not invalidate the inputs or the outputs - the dependency is just no longer captured. Causality naturally arises from the fact that a dependency has well defined inputs and outputs.

A dependency is considered to be a model; it is possible that this model is unknown at a given design stage; in this case a dependency can still be specified (along with its input and output properties) in a dependency model. Once the model that specifies a particular dependency is known, references can be created between this model and the dependency model. A natural question to ask is why not utilize currently available languages to model dependencies. In the work reported in [5], different modeling languages were analyzed for dependency modeling (e.g. OMG SysML^{TM}[6]). However none were found to be suitable for capturing dependencies adequately without having to modify them (e.g., profile extension of SysML). In addition, the size of meta-model extensions (when using a general purpose language for many different purposes) adds to the intrinsic complexity of the underlying system [7] and introduces *accidental complexity* [8].

In contrast to a general purpose modeling language (such as SysML), a DSML is restrictive and has a specific purpose, in particular as per the demands of a specific viewpoint [7], and it captures the object of interest at the appropriate level of abstraction and formalism to help minimize the complexity [9]. Based on this motivation, we will - in the following section - introduce a DSML to model dependencies called the *Dependency Modeling Language* (DML).

3 Example Use Case

In order to illustrate the proposals of this paper to the reader, an example use case is considered: a simple two degree of freedom robot. The design problem is formulated as follows: *Design a pick and place robot with Work Space (WS) coverage of $4m^2$, with Close loop Position Accuracy (CPA) of at least 5mm, and with the End-to-End Response Time (EERT) of the robot should not be more than 0.5 seconds.* Three viewpoints (one for each stakeholder) are considered for this example: mechanical design, control design, and Hardware/Software (Hw/Sw) design. The three stakeholders - based on the design specifications for each viewpoint - develop disparate models focusing on different aspects of the robot by utilizing different design and analysis tools, such as a CAD tool for mechanical design, a control design tool and a software design tool. The semantic overlaps between the views results in dependencies, which will be the focus of the illustration in section 5. It is worthwhile to mention that gaining knowledge about properties CPA and EERT requires the combined work of the three stakeholders, making it essential to manage dependencies.

4 A Modeling Language for Capturing Dependencies

This section describes the DML which is currently supported in Eclipse Modeling Framework (EMF) [10]. Figure 1 illustrates the abstract syntax of the DML using a class diagram. Any model that conforms to this meta-model is referred to as a *Dependency Model*. In the following, the semantics of the language constructs of the DML are discussed.

A **Concept** is a description of an artifact and can refer to the actual product to be developed, or to any of its sub-components. We say that concepts are formed by constraining some of the properties associated with it. With the passage of time, more constraints are put on properties, hence leading to further refined concepts. For instance, by constraining the number of arms of a robot to two, a two arm robot *Concept* is created. A concept *Contains* zero to many subordinate concepts - by way of example, here are a few that can be considered for the two arm robot: Arm1, Arm2, Controller, Motor1, Motor 2, Sensor1, and Sensor2. All these Concepts are contained under the main concept *two arm robot*. Concepts are related to each other through an *isPartOf* relationship, which creates the semantic context around each concept. Each *Concept* can be looked at from many *Viewpoints*, and is characterized by a number of *Properties*, which are captured in a *Model*.

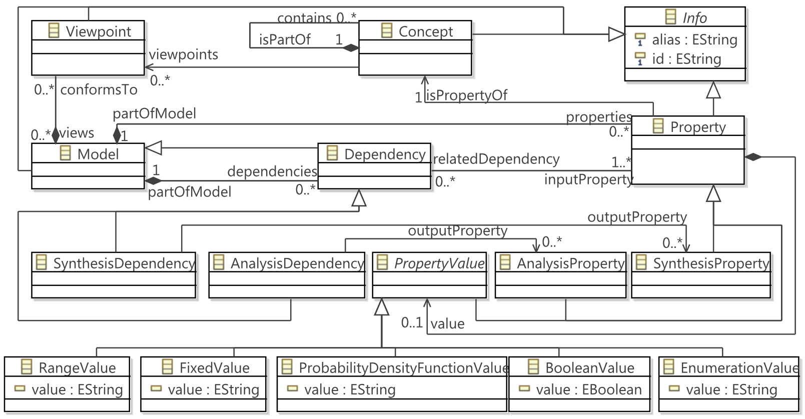


Fig. 1. Abstract syntax (meta-model) of the DML.

A **Viewpoint** refers to the guidelines and conventions used to establish a *View*, where a *View* corresponds to a *Model* or a composition of disparate models: for example, mechanical design viewpoint encompassing a Solid Edge model or the dynamic analysis viewpoint encompassing a Modelica model.

A **Model** is an abstraction of a real-world artifact (described by a *Concept*). Multiple *Viewpoints* may be required to address the stakeholder concerns with respect to an artifact (*Concept*), which can be supported through multiple modeling *Views*. A *Model* contains many *Properties* with multiple *Dependencies* between them.

A **Property** is any descriptor of an artifact. Its value could be numerical, logical, stochastic or an enumeration. In design, two types of properties are used:

properties which are selected or chosen by the designer (Synthesis Properties (SP)) and ones which are predicted through an analysis model or an equation (Analysis Properties (AP)). In order for a property to have an unambiguous meaning, the semantic context around each property should be specified, which is done through *isPropertyOf* relationships to a *Concept*. For example, *EERT isPropertyOf* a *Concept ControlSystem* (see Figure 2), which, in turn, *isPartOf* a *Robot*. A *Property* can influence other properties via a *Dependency*, which is captured through the *relatedDependency* relationship: e.g., SD_4, SD_5 and SD_{13} are related dependencies for the property EERT (see Figure 2).

A **Synthesis Property (SP)** describes the value that a designer has selected for a particular property. SPs are usually defined through a range of values *(RangeValue)*; but they can also be defined through a *FixedValue* or a *BooleanValue*. For example, a load profile could be used as an SP to select the corresponding *actuator power*.

An **Analysis Property (AP)** describes the value predicted as a result of performing an analysis captured in a model, e.g. solving an equation or a constraint. APs are predictions and hence uncertain by definition. Therefore, APs should be specified using a *Probability* value *(ProbabilityDensityFunctionValue)*.

A **Dependency** describes the nature of the relationship between two or more properties. The relationship is assumed to be causal, thereby assuming that some properties are *inputs* while others are *outputs*. The nature of a particular dependency could be known or unknown at a given design stage, and its specifics are described in a number of ways. As per [5], dependencies can be expressed in two forms - a heuristic between two or more properties (*Synthesis Dependency* (SD)), or a constraint, an equation or an analysis model (*Analysis Dependency* (AD)). One particular case is that of an equality binding between two or more properties (e.g., same properties belonging to multiple views). There could be many dependencies within a *Model*, hence a particular *Dependency* can be a part of (i.e. contained within) a particular *Model* (i.e., a model within a model, such as a constraint within a CAD model), or, in other cases, represents a distinct *Model* (e.g., a Simulink model). *Bindings* between properties are captured in the *Dependency Model*.

A **Synthesis Dependency (SD)** refers to the heuristics used in selection of a SP. It is also possible that a modeler uses their experience in making this selection, and overrides the heuristic completely. An SD could have one or more SPs as its output, e.g. SD_4 in Figure 2.

An **Analysis Dependency (AD)** refers to the analysis (present in a model), an equation, or a constraint used to predict the value of an AP. An AD could have one or more APs as its output.

While the dependency models are causal in nature, in many practical scenarios, cycles will be present. For example, an algebraic loop could exist, where a property is both chosen and predicted. From the perspective of structural semantics, cyclic models are valid. However, from the perspective of operational semantics (which are outside the scope of this paper), such loops must be broken during execution - for example, by using the well known tearing algorithm.

5 Illustration: Dependency modeling through the DML

EMF was used to support the DML which we used to construct the dependency model for the example use case. In the following, we will illustrate the equality binding between properties that are part of different views of the robot. Other possible illustrations include (but are not limited to): top-level view of the robot showing the involved *Viewpoints* and *Concepts*, and binding of a *Property* to multiple dependencies. The reader should note that the illustrations we provide are models generated based on the abstract syntax and no concrete syntax was developed at this stage, although graph-based visualizations of the dependency models were built (see Section 6).

Consider the *Synthesis Dependency* SD_4 in the CAD View. Figure 2 shows the dependency SD_4 where Motor A Torque (M_A) is determined based on the information about Inertia of Arm-A (I_A), the requirement for End-to-End Response Time (EERT), and the control system structure (CS).

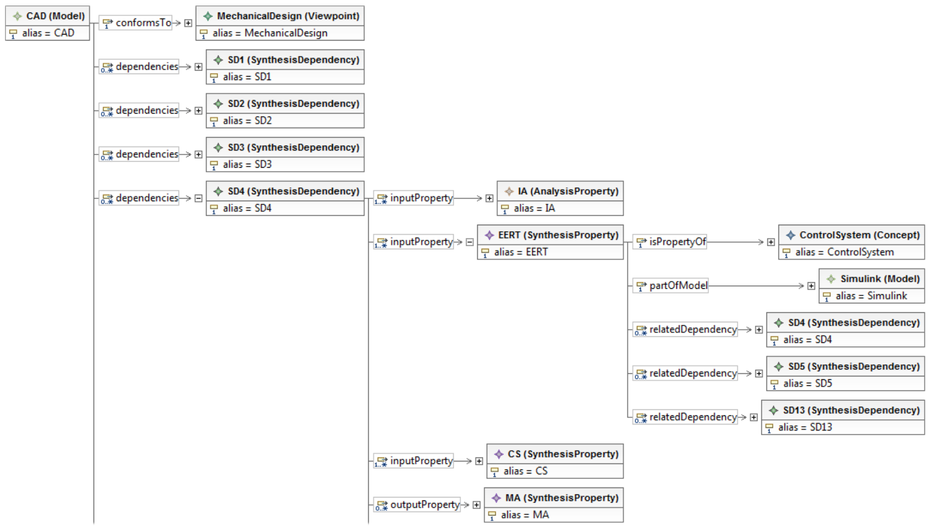


Fig. 2. Contents of the dependency SD_4 (within the CAD View) showing equality binding to the property EERT, which is a property of the Simulink View.

It can be seen that property EERT *isPropertyOf ControlSystem Concept* and is *partOfModel* Simulink, which is a supporting View in the ControlDesign Viewpoint. The resulting property binding relationship is contained under the property EERT, and maintained inside the dependency model. The meta-models of Matlab/Simulink, and MagicDraw SysML (as UML2.1) are available in our Eclipse implementation (Cameo Workbench [11]), and we added the Solid Edge (CAD tool) meta-model to it, hence the models created in these tools can be read as Ecore models and transformations between them can be built.

6 Visualizing dependencies as graphs

The information captured in the dependency model can be visualized by a *dependency graph*. As opposed to a tree-based representation, a graph-based representation is better suited for discussions between different stakeholders (see Figure 2 and Figure 3). We used the tool Graphviz [12], which supports the DOT language [13], to build graphs. Figure 3 shows a directed dependency graph between the three views of the robot. As an example, consider SD_9 which refers to the dependency between inertia of first and second arm of the robot (I_A and I_B in mechanical design view) and the transfer function ($G(s)$) attributes (in the control design view). The figure illustrates that even for a fairly simple robot design example, there are many dependencies between the considered viewpoints, and manual management of such dependencies is either very challenging or often not possible due to a lack of information.

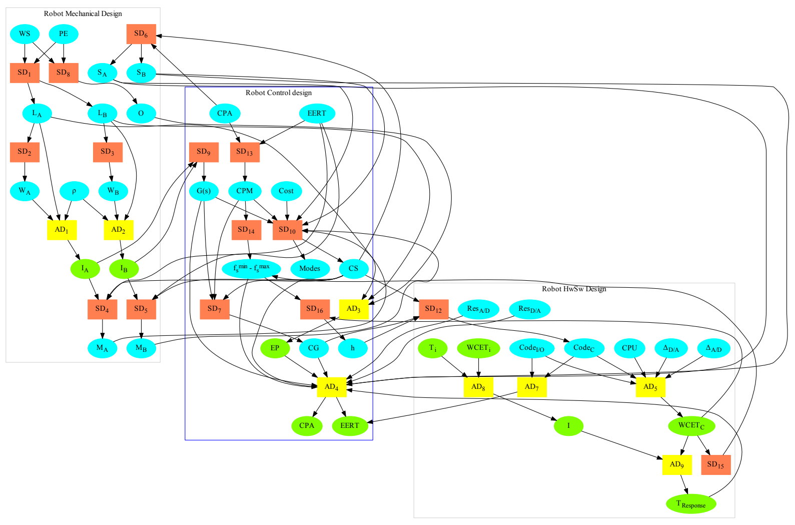


Fig. 3. Dependency graph between three robot views. SPs are shown in Blue, APs in Green, SDs in Orange and ADs in Yellow.

7 Related Work

One popular method of modeling dependencies is the Design Structure Matrix (DSM) [14], which allows relations among properties to be represented in a matrix. DSMs are used in a variety of disciplines - for example, in software engineering [15]. Compared to the DML, DSMs are limited in terms of their expressiveness. For one, it is not possible to differentiate between synthesis and analysis properties, nor between synthesis and analysis dependencies. As discussed in [5], this differentiation is important to keep analysis results separate from selections made by the designer. In addition, this differentiation adds to the semantic richness of a dependency model thereby supporting the change management and inconsistency management scenarios. Furthermore, the semantic context around a property can be shown in a DSM only to a limited degree.

In terms of tools, Product Data Management (PDM) systems are probably among the most widely used systems to manage product-related data. One of the core capabilities of modern PDM systems is allowing users to establish relations between elements that are stored in the repository: for instance, reference and correspondence relationships can be created. Such relationships can typically only be created among files. However, some contemporary PDM systems integrate tool adapters, allowing for certain properties of supported models to be exposed (for example: the part hierarchy in CAD models). While PDM systems implement some of the desired functionality, they are still limited in terms of their capabilities of capturing dependencies. In particular, PDM systems require dependent models to already exist, therefore not enabling one to create a dependency model independently from the corresponding design artifacts.

Modeling dependencies is also supported (at least to some extent) in the Process Integration Design Optimization (PIDO) approach, which is implemented in tools such as ModelCenter [16]. The underlying principle is the integration of disparate models. ModelCenter, for instance, provides several tool connectors, which enable data exchange among disparate models and allow for properties of compatible models to be exposed. As a result, dependencies between properties can be modeled. However, current PIDO tools only provide a black box view for each model and hide most of the semantic context of properties. Furthermore, not all possible properties can be exposed.

To the best of knowledge of the authors, modeling languages intended specifically for the purpose of modeling dependencies have, to the date of writing this paper, not been publicized. While there are some promising methods and tools available, none implement all of the envisioned capabilities. For one, none allow for the definition of a dependency model independent of other domain specific models. Furthermore, of the approaches surveyed, none provided the desired level of depth and access to properties in models.

8 Discussion

The order in which the different views are developed in relation to the dependency model is an important consideration. There are two possibilities here: a bottom-up scenario where the initial design and analysis of design concepts is already captured in multiple views (e.g. a CAD model and a Simulink model), and then the dependency model is built. In this case, the knowledge already present in disparate views can be used to automatically build parts of the dependency model. The other possibility is a top-down scenario where the dependency model is manually created after the requirements and the system architecture are identified, and based on the dependencies captured in the dependency model, other views such as CAD and Simulink models are developed. For the example described in Section 5, we have followed the former approach, where the views supporting mechanical, control and Hw/Sw design of the robot already existed.

Dependency models can be used for more than just one purpose. Given the causal nature, a dependency model can be used for change propagation and consistency management. For example, in Figure 3, a change to the predicted value of $T_{Response}$ triggers the necessity for AD_4 to be refreshed automatically. It can also support traceability in that it is possible to reason about both the existence and nature of certain relationships among models. For example, one useful application is requirements traceability. Dependency models are also useful for the purpose of managing workflow. Given a (causal) network of dependencies, tasks can be parallelized and merge points identified. A dependency model can also be used for the purpose of avoiding certain inconsistencies. Not only can changes be propagated through such a model, but a single source of truth for properties is established. Such is the case because properties in the dependency model are unique, even though these may refer to elements in disparate models.

Modeling dependencies requires additional effort and, hence, additional resources to be allocated. However, any commitment of resources needs to be justified. It is entirely conceivable that in some cases (e.g. very simple or well understood systems) the risk associated with not explicitly capturing dependencies is negligibly low. Similar arguments can be made about the completeness of the dependency model: to what level of detail should dependencies be modeled? As per [5], dependencies can be defined at six levels of detail starting with the level-0 where the dependencies are completely unknown to level-5 where both the dependencies and the transformation models that lead to them are explicitly known. Behind building such transformation models are *dependency patterns* which gather and illustrate known dependencies between specific types of properties under a design context. The use of such patterns would decrease the cost associated with modeling dependencies. Patterns are currently not supported by the introduced DML and their discussion is beyond the scope of this paper.

9 Conclusions

This paper presents a DSML for modeling dependencies between properties. Properties are typically referenced in multiple views on a system. A dependency modeling language allows for the dependencies between these properties to be captured in a single model, as illustrated for a robot example in Section 5.

Future work should includes the provision of additional features, such as supporting modeling at multiple levels of detail and allowing for a variety of stakeholder-specific views to be generated automatically. Furthermore, the operational semantics of the DML should be defined formally. This is particularly important for the purpose of supporting the accompanying dependency management process. For example, an essential task is analyzing how changes propagate.

Since most analysis activities involve some sort of token flow, we suggest to investigate the mapping to the semantic domain of petri-nets as future work.

Acknowledgments. This work was supported in-part by Boeing Research & Technology. The authors would like to thank Michael Christian (Boeing) and Axel Reichwein (Koneksys LLC) for the many valuable discussions.

References

1. Herzig, S.J.I., Qamar, A., Reichwein, A., Paredis, C.J.J.: A Conceptual Framework for Consistency Management in Model-Based Systems Engineering. In: ASME 2011 Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2011, Washington, DC, USA, ASME (2011) 1329–1339
2. Thompson, S.C.: Rational Design Theory: A Decision-Based Foundation for Studying Design Methods. Phd. thesis, Georgia Institute of Technology, Atlanta, Georgia, USA. (2011)
3. Hazelrigg, G.A.: A Framework for Decision Based Engineering Design. Journal of Mechanical Design **120**(4) (1998) 653–658
4. Object Management Group: OMG Unified Modeling Language (UML) Specification V2.4.1 (2011)
5. Qamar, A., Paredis, C.J., Wikander, J., During, C.: Dependency Modeling and Model Management in Mechatronic Design. Journal of Computing and Information Science in Engineering **12**(4) (December 2012) 041009
6. Object Management Group: OMG Systems Modeling Language Specification V1.3 (2012)
7. Vallecillo, A.: On the Combination of Domain Specific Modeling Languages. In: Modeling Foundations and Applications, Lecture Notes in Computer Science. Volume 6138. (2010) 305–320
8. Brooks, F.P.: No Silver Bullet Essence and Accident in Software Engineering. IEEE Computer **20**(4) (1987) 10–19
9. Mosterman, P.J., Vangheluwe, H.: Computer Automated Multi-Paradigm Modeling: An Introduction. Simulation: Transactions of The Society for Modeling and Simulation International **80**(9) (September 2004) 433–450
10. Eclipse Foundation: Eclipse Modeling Framework (EMF) (2009)
11. No Magic: Cameo Work Bench (2011)
12. Ellson, J., Gansner, E.R., Koutsofios, E., North, S.C., Woodhull, G.: Graphviz and Dynagraph - Static and Dynamic Graph Drawing Tools. In: Graph Drawing Software, Springer-Verlag (2003) 127–148
13. Gansner, E.R., Koutsofios, E., North, S.: Drawing Graphs With Dot. Technical report (2009)
14. Eppinger, S.D., Browning, T.R.: Design Structure Matrix Methods and Applications. Engineering Systems. MIT Press (2012)
15. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using Dependency Models to Manage Complex Software Architecture. In: Object Oriented Programming, Systems, Languages & Applications (OOPSLA), San Diego, CA, USA, ACM Press (2005) 167–176
16. Phoenix Integration: ModelCenter (2012)

Integrating Prose as First-Class Citizens with Models and Code

Markus Voelter

independent/itemis, `voelter@acm.org`

Abstract. In programming and modeling we strive to express structures and behaviors as formally as possible to support tool-based processing. However, some aspects of systems cannot be described in a way that is suitable for tool-based consistency checking and analysis. Examples include code comments, requirements and software design documents. Because they can only be analyzed manually, they are often out-of-sync with the code and do not reflect the current state of the system. This paper demonstrates how language engineering based on language workbenches can help solve this problem by seamlessly mixing prose and program nodes. These program nodes can range from simple references to other elements over variables and formulas to embedded program fragments. The paper briefly explains the language engineering technology behind the approach as well as a number of prose-code integrated languages that are part of mbeddr, an integrated language and tool stack for embedded software engineering.

1 Introduction

Even though developers and systems engineers would love to get rid of prose as part of the development process and represent everything with machine-processable languages and formalisms, prose plays an important role.

In *requirements engineering*, prose is the starting point for all subsequent formalizations. Classical requirements engineering uses prose in Word documents or Doors databases, together with tables, figures and the occasional formula. Since these requirements are not versioned together with the code, it is hard to branch and tag them together with the implementation. In safety-critical domains, requirements tracing is used to connect the requirements to implementation artifacts. Traceability across tools is challenging in terms of tool integration.

During the implementation phase, developers add *comments* to the code. These comments must be associated with program elements expressed in various languages. For example, an architecture description language, a state machine modeling language or a business rule language are considered as part of the implementation. Comments also refer to code (for example, a comment that documents a function typically refers to the arguments of that function), and it is hard to keep these code references in sync with the actual code as it evolves.

Depending on the process, various *design documents* must be created during or after the implementation. These are different from code comments in that they look at the bigger picture and ”tell a story”; they are not inlined into the code, they are separate documents. Nonetheless they are tightly integrated with the code, for example, by referring to program elements or by embedding code fragments. Today, such documents are usually written in Latex, Docbook or Word – and synchronized manually with the implementation code.

Problem Prose is is often badly integrated with the artifacts it relates to. It cannot be checked for consistency with implementation artifacts. Mixing prose and code or models is hard: either they reside in separate files, or, if pseudo-code is embedded into a requirements document, it is not checked with regards to syntax and type system rules. No IDE support for the programming or modeling language is available. This leads to a lot of tedious and error-prone manual synchronization work.

Contribution This paper proposes a highly integrated approach for handling prose in the context of model-driven engineering tools that solves the challenges outlined above. The implementation behind the approach relies on language engineering and language workbenches, and an implementation has been developed as part of the mbeddr platform.

2 mbeddr and MPS

mbeddr[1] is an open source project supporting embedded software development based on incremental, modular domain-specific extension of C [7,8]. It also supports languages that address other aspects of software engineering such as requirements or documentation (which is what is discussed in this paper).

mbeddr Overview mbeddr builds on the JetBrains MPS language workbench[2], a tool that supports the definition, composition and use of general purpose or domain-specific languages. MPS uses a projectional editor, which means that, although a syntax may look textual, it is *not* represented as a sequence of characters which are transformed into an abstract syntax tree (AST) by a parser. Instead, a user's editing actions lead *directly* to changes in the AST. Projection rules render a concrete syntax from the AST. Consequently, MPS supports non-textual notations such as tables, and it also supports unconstrained language composition and extension – no parser ambiguities can ever result from combining languages (see [6] for details).

The next layer in mbeddr is an extensible implementation of the C99 programming language in MPS. On top of that, mbeddr ships with a library of reusable extensions relevant to embedded software. As a user writes a program, he can import language extensions from the library into his program. The main extensions include test cases, interfaces and components, state machines, decision tables and data types with physical units[3]. For many of these extensions, mbeddr provides an integration with static verification tools (model checking

[1] `http://mbeddr.com`

[2] `http://jetbrains.com/mps`

[3] I do not distinguish between *models* and *code*. While C99 artifacts would probably be called code, state machines would likely be called models. Since both are

state machines, verifying interface contracts or checking decision tables for consistency and completeness; see also [5]).

Finally, mbeddr supports three important aspects of the software engineering process: requirements engineering and tracing [9], product line variability and documentation. All are implemented in a generic way that makes them reusable with any mbeddr-based language. We discuss the prose aspect of requirements, documentation and code comments in the rest of this paper.

Multiline Text Editing The projectional nature of the MPS editor has important advantages with regards to extensibility of languages. However, it also means that the editor is a bit more rigid than a regular text editor. In particular, until recently, MPS did not support multiline strings with the familiar editing experience where pressing `Enter` creates a line break, pressing ↑ moves the cursor to the line above the current one, or deleting a few words on a line "pulls up" the text from the next line. However, the `mps-multiline`[4] MPS plugin, developed by Sascha Lisson, has enabled this behavior. In addition, an additional plugin[5] supports embedding program nodes into this multiline prose. At any location in the multiline text, a user can press `Ctrl-Space` and select from the code completion menu a language concept. An instance of this concept is then inserted at the current location. The program node "flows" with the rest of the text during edit operations. Other editing gestures can also be used to insert nodes. For example, an existing regular text word can be selected, and, using a quick fix, it can wrapped with an `emph(...)` node, to mark the word as *emphasized*.

The set of language concepts that can be embedded in prose text this way is extensible; the concept simply has to implement the `IWord` interface. For a developer who is familiar with MPS, the implementation takes only a few minutes.

Implementing an Embeddable Word In MPS, language elements (called *concepts*) have children, references and properties. They can also inherit from other concepts and implement concept interfaces such as `IWord`. The multiline prose editor widget works with instances of `IWord`, and by implementing this interface we can "plug in" new language concepts into the multiline editor. An example is `ArgRefWord` which can be embedded into function comments to reference an argument of that function:

```
concept ArgRefWord implements IWord
 references:           concept properties:
   Argument arg 1        transformKey = @arg
```

It states that the concept implements `IWord`, that it references one `Argument` (by the role name `arg`) and it uses the `@arg` transformation key: typing `@arg` in a comment, followed by `Ctrl-Space`, instantiates an `ArgRefWord`.

tightly integrated in mbeddr, the distinction makes no sense and I use the two terms interchangeably.

[4] `http://github.com/slisson/mps-multiline`

[5] `http://github.com/slisson/mps-richtext`

A reference to an argument should be rendered as `@arg(argName)`, so we have to define an appropriate editor:

```
[- @arg ( %arg%->{name} ) -]
```

The editor defines a list of cells `[- -]` inside which we define the constant `@arg`, followed by the `name` property of the referenced `Argument`, enclosed in parentheses. To restrict this `IWord` to comments of functions, a constraint is used:

```
can be child constraint for ArgRefWord {
  (node, parent, operationContext)->boolean {
    node<> comment = parent.ancestor<DocumentationComment>;
    node<> owner = comment.parent;
    return owner.isInstanceOf(Function) }
```

We also define the scope for the `arg` reference, since only those arguments owned by the function under which the documentation comment lives are valid targets:

```
link {arg} scope: (refNode, enclosingNode)->sequence<node<Argument>>) {
     enclosingNode.ancestor<Function>.arguments; }
```

Finally, a generator has to be defined that is used when HTML or LaTeXoutput is generated. In this case we simply override a behavior method that returns the text string that should be used:

```
public string toTextString() overrides IWord.toTextString {
  "@arg(" + this.arg.name + ")"; }
```

This completes the implementation. All in all, only 10 lines of code have to be written (the remaining ones shown above are IDE scaffolding)

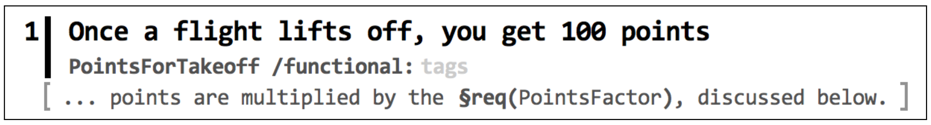
3 Integrating Prose with Models

In this section we look at various examples of integrating prose with code, addressing the challenges discussed in Section 1.

3.1 Requirements Engineering

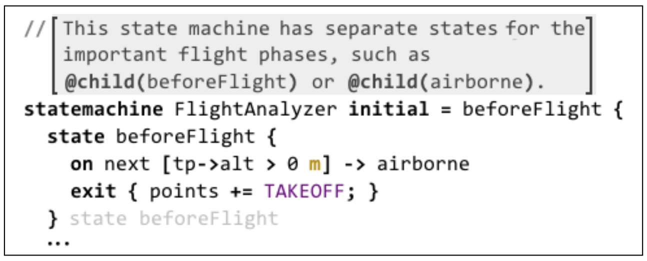
As discussed in [9], mbeddr's requirements engineering support builds on the following three pillars. First, requirements can be collected as part of mbeddr models and they are persisted along with any other code artifact. A requirement has an ID, a prose description, relationships to other requirements (`refines`, `conflicts with`) as well as child requirements. Second, the requirements language is extensible in the sense that arbitrary additional attributes (described with arbitrary DSLs) can be added to a requirement. Examples include business rules or use cases, actors and scenarios. The third pillar is traceability: trace links can be attached to any program element in any language.

In the context of this paper, the interesting aspect is that the prose description can contain additional nodes, such as references to other requirements (the `§req` nodes in Fig. 1). References to actors, use cases and scenarios are also supported. Since these are real references, they are automatically renamed if the target element is renamed. If the target element is deleted, the reference breaks and leads to an error. Referential integrity can easily be maintained.



```
1 Once a flight lifts off, you get 100 points
  PointsForTakeoff /functional: tags
[ ... points are multiplied by the §req(PointsFactor), discussed below. ]
```

Fig. 1. Requirements descriptions can contain references to other requirements (the `§req` node in the text above), as well as references to actors, use cases and scenarios.



```
//[This state machine has separate states for the]
  [important flight phases, such as              ]
  [@child(beforeFlight) or @child(airborne).     ]
statemachine FlightAnalyzer initial = beforeFlight {
  state beforeFlight {
    on next [tp->alt > 0 m] -> airborne
    exit { points += TAKEOFF; }
  } state beforeFlight
  ...
```

Fig. 2. A state machine with a comment attached to it. Inside the comment, we reference two of the states of the state machine.

Note that the mainstream requirements management tool, DOORS, cannot embed references in the requirements description, they can only be added as a separate attribute, which is awkward in terms of the semantic connection between the text and the reference.

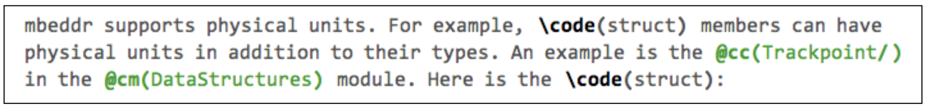
3.2 Code Comments

In classical tools, a comment is just specially marked text in the program code. As part of this text, program elements (such as module names or function arguments) are mentioned. We observe two problems with this approach. First, the association of the comment with the commented element is only by proximity and convention – usually, a comment is located above the commented element (this is true only in textual editors, graphical modeling tools usually do not have this problem). Second, references to other program elements are by name only – if the name changes, the reference is invalid. mbeddr improves on both counts.

First, a comment is not just associated by proximity with the commented program node, it is actually *attached* to it. Structurally the comment is a child of the commented node, even though the editor shows it on top (Fig. 2). If the element is moved, copied, cut, pasted or deleted, the comment always goes along with the commented element.

Second, comments can contain `IWord`s that refer to other program elements. For example, the comment on the state machine in Fig. 2 references two of the states in the state machine. Some of the words that can be used in comments can be used in any comment (such as those that reference other modules or functions), whereas others are restricted to comments for certain language concepts (references to states can only be used in comments on or under a state machine).

Note that some IDEs support real references in comments for a specific language (for example, Eclipse JDT renames argument names in JavaDoc comments



```
mbeddr supports physical units. For example, \code(struct) members can have
physical units in addition to their types. An example is the @cc(Trackpoint/)
in the @cm(DataStructures) module. Here is the \code(struct):
```

Fig. 3. This piece of document code uses `\code` tags to format parts of the text in code font. It also references C program elements (using the `cm` and `cc` tags). The references are actual, refactoring-safe references. In the generated output, these references are also formatted in code font.

for functions if an argument is renamed). mbeddr's support is more generic in that it automatically works for any kind of reference inside an `IWord`. This is important, since a cornerstone of mbeddr is the ability to extend all languages used in it (C, the state machine language or the requirements language). The commenting facility must be similarly generic.

3.3 Design Documents

mbeddr supports a documentation language. Like other languages for writing documents (such as LaTeX or Docbook), it supports nested sections, text paragraphs and images. We use special `IWords` to mark parts of texts as emphasized, code-formatted or bold. Documents expressed in this language live inside MPS models, which means that they can be versioned together with any other mbeddr artifact. The language comes with generators to LaTeX and HTML, new ones (for example, to Docbook) can be added.

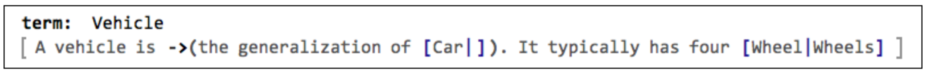
Referencing Code Importantly, the documentation language also supports tight integration with mbeddr languages, i.e. C, exiting C extensions or any other language developed on top of MPS. The simplest case is a reference to a program element. Fig. 3 shows an example.

Embedding Code Code can also be embedded into documents. In the document source, the to-be-embedded piece of code is referenced. When the document is generated to LaTeX or HTML, the actual source code is embedded either as text or as a screenshot of the notation in MPS (since MPS supports non-textual notations such as tables, not every program element can be sensibly embedded as text). Since the code is only embedded when the document is generated, the code is always automatically consistent with the actual implementation.

Visualizations A language concept that implements the `IVisualizable` interface can contribute visualizations, the context menu for instances of the element has a *Visualize* item that users can select to render a diagram in the IDE. The documentation language supports embedding these visualizations. As with embedding code, the document source references a visualizable element. During output generation, the diagram is rendered and embedded in the output.

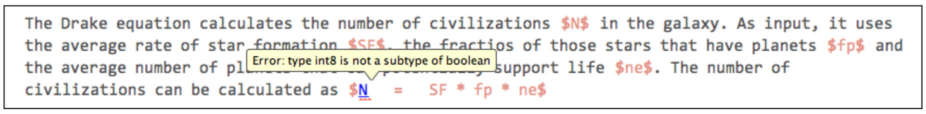
4 Extensibility

A hallmark of mbeddr is that everything can be extended by end users (without invasively changing the extended languages), and the prose-oriented lan-



```
term:  Vehicle
[ A vehicle is ->(the generalization of [Car|]). It typically has four [Wheel|Wheels] ]
```

Fig. 4. A modular extension of the documentation language that supports the definition of glossary terms and the relationships between them. Terms can be referenced from any other prose, for example from comments or requirements.



```
The Drake equation calculates the number of civilizations $N$ in the galaxy. As input, it uses
the average rate of star formation $SF$, the fractios of those stars that have planets $fp$ and
the average number of pl[illegible] support life $ne$. The number of
civilizations can be calculated as $N  =  SF * fp * ne$
```

Error: type int8 is not a subtype of boolean

Fig. 5. An example where variable declarations and equations are integrated directly with prose. Since the expressions are real C expressions, they are type checked. To make this possible, the variables have types; these are specified in the properties view, which is not shown in the figure. To provoke the type error shown above, `boolean` has been defined as the type of the N variable.

guages can be extended as well. The extension mechanism that uses new language concepts that implement the `IWord` interface has already been discussed. This section discusses a few example of further extensions, particularly of the documentation language (Section 3.3).

Glossaries An obvious extension is support for glossaries. A glossary defines terms which can be referenced from other term definitions or from regular text paragraphs or even requirements or code comments. Such term definitions are subconcepts of `AbstractParagraph`, so they can be plugged into regular documents. Fig. 4 shows an example of a term definition.

The term in Fig. 4 also shows how other terms are referenced using the `[Term|Text]` notation (such references, like others, are generated to hyperlinks when outputting HTML). The first argument is a (refactoring-safe) reference to the target term. The optional second argument is the text that should be used when generating the output code; by default, it is the name of the referenced term. Terms can also express relationships to other terms using the `->(...)` notation, which creates a dependency graph between the terms in the glossary. A visualization is available that renders this graph as a diagram.

Formulas Another extension adds variable definitions and formulas to prose paragraphs (Fig. 5) which are exported to the math mode of the respective target formalism. However, the variables are actual referenceable symbols and the equations are C expressions. Because of this, the C type checker performs type checks for the equations (see the red underline under N in Fig. 5). mbeddr' interpreter for C expressions can be plugged in to evaluate the formulas. This way, live test cases could be integrated directly with prose.

Going Meta Section 3.3 has demonstrated how programs written in arbitrary languages can be integrated (by reference or by embedding) with documents written in the documents language. However, sometimes the *language definitions themselves* need to be documented, to explain how to develop languages in MPS/mbeddr. To make this possible, a modular extension of the documentation language can be used to reference or embed language implementation artifacts. Similarly, documentation language documents can be embedded as well, to write documents that explain how to use the documentation language. The user guide for the documentation language[6] has been created this way.

Cross-Cutting Concerns mbeddr supports two cross-cutting concerns that can be applied to any language. Since the documentation language is *just another language*, it can be used together with these cross-cutting languages. In particular, the following two facilities are supported. First, requirements traces can be attached to parts of documents such as sections, figures or paragraphs. This way, requirements traceability can extend into, for example, software design documents. This is an important feature in safety-critical contexts. Second, mbeddr supports product line variability. In particular, static negative variability is supported generically. Using this facility, documents such as user guides, configuration handbooks or software design documents can be made variant-aware in the same way as any other product line implementation artifact.

Generating Documents Documents cannot just be written manually, they can also be generated from other artifacts. For example, mbeddr's requirements language supports generating reports, which contain the requirements themselves, the custom attributes (via specific transformations) and trace information. This feature is implemented by transforming requirements collections to documents, and then using the generators that come with the documentation language to generate the PDFs.

5 Related Work

The idea of more closely integrating code and text is not new. The most prominent example is probably Knuth's literate programming approach [4], where code fragments are embedded directly into documents; the code can be compiled and executed. While we have built a prototype with mbeddr that supports this approach, we have found referencing the code from documents (and generating it into the final PDF) more scalable and useful.

The closest related work is Racket's Scribble [2]. Following their paradigm of *documentation as code*, Scribble supports writing structured documentation (with Latex-style syntax) as part of Racket. Racket is an syntax-extensible version of Scheme, and this extensibility is exploited for Scribble. Scribble supports referencing program elements from prose, embedding scheme expressions (which are evaluated during document generation) and embedding prose into code (for JavaDoc-like comments). The obligatory literate programming example has also been implemented. The main differences between mbeddr's approach and Racket Scribble is that Scribble is implemented as Racket macros, whereas mbeddr's facility are based on projectional editing. Consequently, the range of document styles and syntactic extensions is wider in mbeddr. Also, mbeddr directly supports embedding figures and visualizations.

[6] `http://bit.ly/10gUs0q`

Essentially all mainstream tools (incl. modeling tools, requirements management tools or other engineering tools) treat prose as an opaque sequence of characters. None of the features discussed in this paper are supported. The only exception are Wiki-based tools, such as the Fitnesse tool for acceptance testing[7]. There, executable test cases are embedded in Wiki code. A big limitation is that there is no IDE support for the (formal) test case description language embedded into the Wiki markup. mbeddr provides this support for arbitrary languages.

One exception to the statement made above is Mathematica[8], which supports mixing prose with mathematical expressions. It even supports sophisticated type setting and WYSIWYG. Complete books, such as the Mathematica book itself, are written with Mathematica. mbeddr does not support WYSIWYG. However, mbeddr documents support integration with arbitrary MPS-based languages, whereas Mathematica has a fixed programming language.

One way of integrating program code and prose that is often used in book publishing are custom tool chains, typically based on LaTeX or Docbook. Program files are referenced by name from within the documents, and custom scripts copy in the program code as part of the generation of the output. mbeddr's approach is much more integrated and robust, since, for example, even the references to program fragments are actual references and not just names.

mbeddr's approach to integrating references (to, for example, text sections, figures or program nodes) into documents relies on user-supplied mark up: a reference must be inserted explicitly, either when creating the document, or using a refactoring later. mbeddr makes no attempt at automatically understanding, parsing or checking natural language (in contrast to some approaches in requirements engineering [1,3]). My experience is that such approaches are not yet reliable enough to be used in everyday work. However, it would be possible to add automatic text recognition to the system; an algorithm would examine existing text-only documents and introduce the corresponding nodes. We have built a prototype for the trivial case where a term is referenced from another term in the glossaries extension: by running a quick fix on a glossary document, plain-text references to terms are replaced by actual term references.

mbeddr relies on MPS, whose projectional editor is one of the core enablers for modular language extension. This means that arbitrary language constructs with arbitrary syntax can be embedded into prose blocks. I have seen a prototype of embedding program nodes into comments in Rascal[9]. However, at this point I do not understand in detail the limitations and trade-offs of this approach. However, one limitation is that the syntax is limited to parseable textual notations.

6 Conclusion

mbeddr is a scalable and practically usable tool stack for embedded software development. However, a secondary purpose of mbeddr is to serve as a convincing

[7] `http://fitnesse.org/`
[8] `http://www.wolfram.com/mathematica/`
[9] `http://www.rascal-mpl.org/`

demonstrator for the *generic tools, specific languages* paradigm, which emphasizes language engineering over tool engineering: instead of adapting a tool for a specific domain, this paradigm suggests to use generic language workbench tools and then use language engineering for all domain-specific adaptations.

As this paper shows, this approach can be extended to prose. Through the ability to embed program nodes into prose, prose can be checked for consistency with other artifacts. Of course, this does not address all aspects of prose. For example, consider a program element (such as a function) that is referenced from a prose document that explains the semantics of this program element. If the semantics changes (by, for example, changing the implementation of the function), the *explaining* prose does not automatically change. However, `Find Usages` can always be used to find all locations where in prose a program element is referenced. This simplifies the subsequent manual adaptations significantly.

Since prose is now edited with an IDE, some of the IDE services can be used when editing documents: go-to-definition, find usages, quick fixes, refactorings (to split paragraphs or to introduce term references in prose) or visualizations. Taken together with the direct integration with code artifacts, this leads to a very productive environment for managing requirements or writing documentation.

Acknowledgements I thank the mbeddr and MPS development teams for creating an incredibly powerful platform that can easily accommodate the features described in this paper. We also thank Sascha Lisson for building developing the `multiline` and `richtext` plugins for MPS.

References

1. V. Ambriola and V. Gervasi. Processing natural language requirements. In *Proceedings of the 12th IEEE Intl. Conf. on Automated Software Engineering, 1997.*
2. M. Flatt, E. Barzilay, and R. B. Findler. Scribble: closing the book on ad hoc documentation tools. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, New York, NY, USA, 2009. ACM.
3. V. Gervasi and B. Nuseibeh. Lightweight validation of natural language requirements. *Software: Practice and Experience*, 32(2):113–133, 2002.
4. D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
5. D. Ratiu, M. Voelter, B. Schaetz, and B. Kolb. Language Engineering as Enabler for Incrementally Defined Formal Analyses. In *FORMSERA'12*, 2012.
6. M. Voelter. Language and IDE Development, Modularization and Composition with MPS. In *4th Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2011)*, LNCS. Springer, 2011.
7. M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Journal of Automated Software Engineering*, 2013.
8. M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proc. of the 3rd conf. on Systems, programming, and applications: software for humanity*, SPLASH '12, pages 121–140, New York, NY, USA, 2012. ACM.
9. M. Voelter and F. Tomassetti. Requirements as first-class citizens: Tight integration between requirements and code. In *Proc. of the 2013 Dagstuhl Workshop on Model-Based Development of Embedded Software*, 2013.

Towards a Multi-Domain Model-Driven Traceability Approach

Masoumeh Taromirad, Nicholas Matragkas, and Richard F. Paige

Department of Computer Science, University of York, UK
[mt705,nicholas.matragkas,richard.paige]@york.ac.uk

Abstract. Traceability is an important concern in projects that span different engineering domains. In such projects, traceability can be used across the engineering lifecycle and therefore is *multi-domain*, involving heterogeneous models. We introduce the concept and challenges of *multi-domain traceability* and explain how it can be used to support traceability scenarios. We describe how to build a multi-domain traceability framework using Model-Driven Engineering. The approach is illustrated in the context of the safety-critical systems engineering domain where multi-domain traceability is required to underpin certification arguments.

1 Introduction

Traceability is a key element of any rigorous software development process, providing critical support for many development activities. In some cases, traceability is mandated so as to comply with regulations, e.g., in civil aviation projects. However, there are substantial challenges associated with its use in practice, including identifying the most appropriate artefacts to trace. This makes it difficult to define a generic and effective *traceability framework* – consisting of a Traceability Information Model (TIM), traceability information, and analysis tools – to be used to manage trace information for a specific project; such frameworks are thus still rarely defined and used [1].

In many contexts, such as projects developing high-assurance software systems, different kinds of traceability are mandated [2]. Such projects address multiple engineering domains (e.g., software, mechanics and safety). Each domain has its own stakeholders, artefacts, tools, and goals. Stakeholders of any single domain may be concerned with both intra- and inter-domain traceability. For example, a software developer will be interested in traces from system to software requirements, while a safety engineer will want to trace relationships between fault tree analysis, software requirements and verification artefacts. Considering that traceability may be required throughout the project lifecycle, any traceability framework needs to operate across the project's different domains. In this respect, traceability is a *multi-domain* concern.

The core element of any traceability framework is a traceability information model (TIM) which provides guidance as to which artefacts to collect and which relations to establish in order to support traceability goals [3]. Traceability

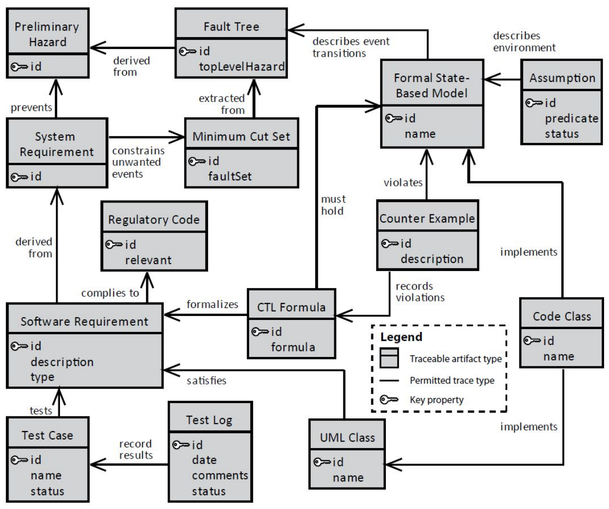


Fig. 1. A traceability information model for a safety-critical system [2]

goals, e.g., ‘traceability of designs against requirements’ and ‘track the allocation of requirements to system components’, specify purposes for accumulating traceability data. A TIM may refer to artefacts (documents, models, databases, project activities context) from different domains or require relationships between multiple domains.

The main contribution of this research is a model-driven approach to support multi-domain traceability. It introduces detailed steps that can be used to build a multi-domain traceability framework. We express a TIM as a domain-specific modelling language and use model-driven engineering (MDE) techniques to derive traceability information from sources, record the information in a traceability model (TM), and finally perform traceability analyses, based on traceability goals. Section 3 describes the steps in more detail and gives potential ways in which MDE can help support them.

2 Motivation

To introduce our approach, we give an example from the safety-critical systems domain. We then highlight the open challenges for multi-domain traceability.

2.1 Example

Figure 1 depicts a basic TIM for a safety-critical system using a UML class diagram [2]. The traceable entities include artefacts from two domains: software development and safety engineering.

Typical software development artefacts are seen along the left side of the diagram: for example, software requirements are derived from system requirements; classes are designed according software requirements and implemented by code. Meanwhile, safety engineering requires additional artefacts to be produced and traced to the general software development artefacts; these are shown mainly on the right hand side of the diagram. For example, the preliminary hazard artefact documents hazards that could lead to system failure. Such hazards are modelled in more detail in a fault tree which looks at events that could lead to the hazards. System Requirements are specified to prevent hazards from occurring by preventing the unwanted events documented in the Minimum Cut Sets. The Software Requirements may also have to comply with Regulatory Codes.

Although the TIM captures all the traceable components in one metamodel and ultimately will be instantiated in a single traceability model (TM), most of the traceability information needed to build the TM is available in different domains. For example, the relationship between 'System Requirement' and 'Software Requirement' is an elementary trace link specified in general software development projects (regardless of the type of the project). The link between 'Formal State-Based Model' and 'Assumption' is a link type normally provided in the safety engineering domain. Accordingly, each domain includes traceability information related to that domain.

In this respect, to capture traceability information and generate a traceability model (TM), we need to find ways that (re-)use existing information and minimises rework.

2.2 Challenges

A review of the literature suggests several challenges for multi-domain traceability not fully addressed by existing approaches.

Domain-specific Traceability Information Traceability information is captured and collected based on a TIM. As illustrated in the above example, each domain includes traceability information specific to that domain. For a systems engineering project, each domain (and hence each TIM) can provide part of the information needed to generate a complete traceability model. In this respect, local traceability information is essential to capture and record project-wide traceability information, though there is no guarantee that it will be sufficient to achieve traceability goals.

Heterogeneous Traceability Information Usually, available traceability information is provided in different formats, including documents (plain text or structured languages), models (e.g. UML class diagrams), databases, tools, or XML documents. To collect the traceability information using existing available information, we need to find a systematic approach to extract the required information and integrate them as the ultimate traceability information. In this context, integration of information from various domains which are expressed in different and heterogeneous formats is an important concern.

Missing Inter-domain Traceability Information As mentioned earlier, we cannot just rely on the available information as it normally does not cover inter-domain information. Usually, the relationships between domains are defined informally or incompletely which results in inconsistencies and redundancies among domains. Specifying and recording the inter-domain relations are of the essential needs to accumulate the traceability information.

Separation of Concern (SoC) People are interested in their own domain and usually prefer to work with familiar tools or techniques; the existing tools, models, and techniques for a specific domain which are specialised for that domain. Therefore, it is not reasonable to require all stakeholders to work with the traceability model directly.

Tool Support Tool support for a traceability framework is essential to maximise the return on investment in building a TIM. However, practical guidance on how to define and implement a TIM, and use it in practice, is still a poorly understood issue [1]; the effort needed for specifying and managing a TIM and the tools which let the user implement it are the main concerns of supporting traceability.

3 Multi-Domain Traceability

This section introduces our approach to support multi-domain traceability. It defines a traceability framework, and discusses how a model-driven approach to traceability can help to effectively support the approach. Our approach constructs a modelling language to describe the traceability information model (TIM). We identify and specify the relationships between TIM and existing project information sources. Traceability information is captured and instantiated in a *single* traceability model (TM) that conforms to the TIM and is used to perform traceability analyses. Fig. 2 illustrates the proposed approach.

3.1 Traceability Information Model

The core element of any traceability framework is a TIM, which identifies the information required to support traceability goals, such as which artefacts should be traced, the level of detail of the traces, and how traceability links should be classified regarding their usage, context or semantics [3]. So, the first step to define a traceability framework is to define a suitable TIM that supports project traceability goals.

In our approach, the TIM is described as a modelling language, and is thereafter used to generate traceability models. Fig. 1 shows an example TIM described as a UML class diagram.

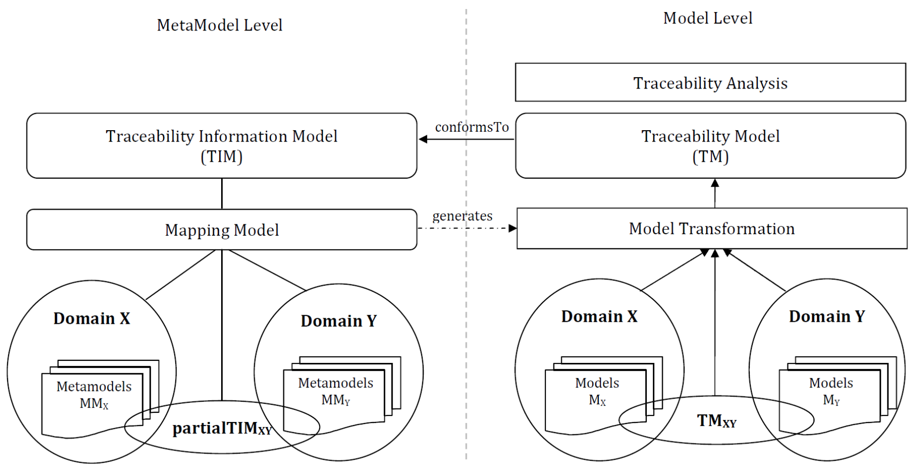


Fig. 2. The proposed model-driven approach to multi-domain traceability

3.2 Traceability Information

Once the TIM has been defined, traceability information is collected and recorded in a traceability model. In our approach, the traceability model is built on top of the other models, generated automatically (by a query), not containing any information that cannot be regenerated automatically. We consider the TM as a *view* similar to 'view' in database context in which view is defined as a dependent object over some tables and theoretically generated on-demand. A single TM provides a coherent view of the traceability information; using a diverse set of traceability information sources (usually represented in heterogeneous formats) is one of the main problems in working with trace links [4]. The TM unifies the way in which the traceability information can be used to perform traceability analyses.

We propose the following steps and activities to build the traceability model:

Step 1: Identify the available information. Based on the TIM, available information sources are gathered to find out how much of the required information is provided and available, in which ways, and how it can be used. As a result, the available information (models, trace link types both within and between domains) and missing information (models, trace link types, . . .) is identified.

Step 2: Add the missing information. Based on the results of step 1, we complete the information sources to provide the missing information. The missing information can be divided into two parts: information limited to one domain and that which relates to multiple domains, such as inter-domain trace link types. We elaborate on this in more detail.

Step 2.1: Add the missing information from one domain. To complete the missing information in one domain, the following options are available:

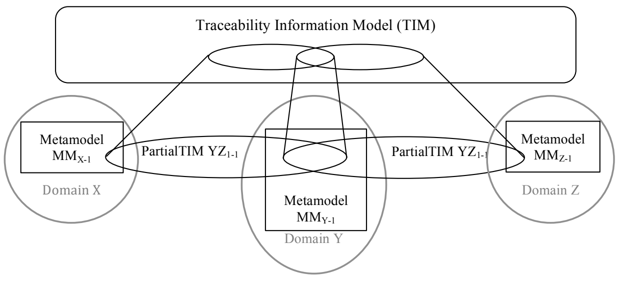


Fig. 3. Partial Traceability Information Models

– complete/extend existing metamodels in each domain by adding missing objects, trace link types, and validation rules, and then update or regenerate the models.
– define new metamodels and create new models within one domain, whenever required.

Step 2.2: Add the inter-domain missing information. After completing the required information for each domain, the inter-domain information are considered. One of the main missing information would be the trace link types defined between domains which can not be easily captured and recorded. The available trace link types are usually limited to one domain. To represent the inter-domain traces, we propose building a traceability model between pairs of domains, wherever required, to provide the missing trace links. To do this, we need to define a traceability information model for each required traceability model. Each traceability information model – a *partial* TIM – is a submodel of the main TIM. Fig. 3 shows the relationship between partial TIMs and the main TIM.

Step 3: Define the mapping between TIM and the information sources. Once all the source models for the traceability information are available, we define the *mapping* between these models and the TIM. The mapping explicitly specifies how each concept (object and trace link type) in the TIM is related to concepts in the source models. The mapping is used to collect information and generate the traceability model. The mapping is also used to interpret the result of traceability analyses, performed on TM, in terms of source models in different domains. The *mapping* is similar to a Correspondence Model (CM) in model composition [5]. A CM is a model that explicitly describes the relationships between elements of different models, but is constructed specifically for model comparison or merging processes.

Step 4: Generate the traceability information. Finally, based on the mapping, the traceability model is generated (as depicted in Fig. 4) The traceability model is created so as to minimise redundancy and inconsistency; it captures the min-

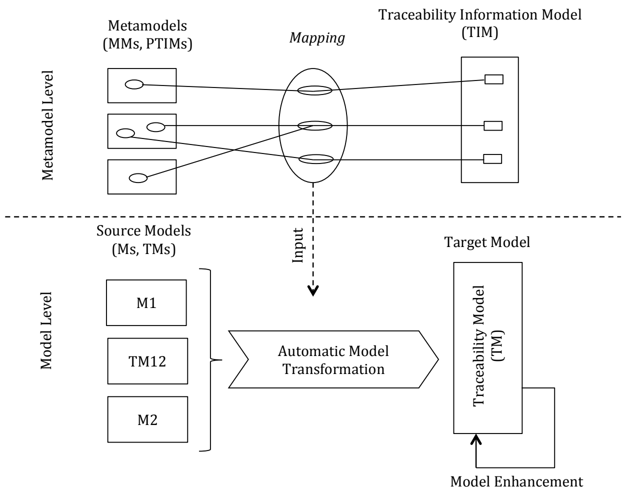


Fig. 4. Generating the traceability model (TM)

imum information needed. For example, it may just contains reference to the source elements in the source models instead of redefining these elements.

As mentioned before, the required traceability information could be represented in various formats (e.g. plain text, XML files) with different underlying structures and metamodels. To support analyses and an overall MDE approach to developing tools, we need to provide a MDE view of the non-model information (wherever it is possible and reasonable) as a prerequisite of building the TM; this can be challenging. However, we focus on those models that are available and that information which can be automatically transformed to models: there are several types of model in different domains (e.g., requirements models, safety analysis models) that provide substantial traceability information. Our approach does not limit the types of model that can be considered in the traceability framework.

3.3 Traceability Analysis

Traceability information is captured and recorded to support traceability goals. Usually, traceability goals are explained in very abstract terms. For example, they can be expressed 'traceability of designs against requirements' or 'track the allocation of requirements to system components'. To be able to support the goals, we need to define concrete traceability analyses which can be applied on the traceability information to determine whether traceability goals are satisfied. In this way, each traceability goal may result in one or more traceability analyses which are defined in terms of traceability and the TIM.

We can use generic query languages (e.g. SQL) and model management languages to express the traceability analyses, which require knowledge of the underlying structures in which the traceability information is stored. As an alternative, we suggest defining a task-specific query language to express traceability analyses precisely. A task-specific query language – bound to the TIM – would hide the underlying complexity and diversity of the underlying information representation.

3.4 Implementation

Typically, an implementation of a TIM will be in the form of a *traceability metamodel*, which will be used to create traceability models. It is these TMs that model the trace links between concepts and artefacts in a project.

In our approach, the TIM is defined as a modelling language; this can be done with any metamodelling technology. Model management tools can then be used to query and manipulate traceability models. For example, the Ecore metamodelling language [6] and Epsilon [7] can be used to describe the traceability metamodel and work with models, respectively. We implemented a basic TIM (for a safety case study) using Ecore, and used Epsilon (specifically EOL, ETL, EVL) to build a TM, query the traceability model, execute constraints on models, and generate analysis reports.

3.5 Discussion

The model-driven approach to TIM definition and implementation enables us to work with arbitrary engineering models and effectively use them. The approach uses the domain-specific traceability information, extract the required information from them regarding the TIM, and generate a TM for the project. The TM is a *view* built automatically through model management operations over the available models in different domains. Throughout the process to generate the TM, missing traceability information, mainly inter-domain traces, is identified and added to the existing information.

The TIM and TM will allow the traceability users to ignore the underlying information complexity, data structures, and information representation format of artefacts, instead allowing them to focus on achieving traceability goals. The proposed approach also supports separation of concerns: different artefacts from different domains that are being traced do not need to be combined (possibly artificially) in one overall description, and can be managed separately while traceability information is defined.

One of the main concerns with traceability implementations and tools is managing *change*, for example, changes in the TIM. The traceability framework that we have developed is also subject to change and evolution. Based on an analysis of the artefacts involved in the traceability framework, and the types of change we can encounter in MDE, we focus on the following changes to the traceability framework: change in TIM, change in domain metamodels, change in domain models. Change in the TIM is the most expensive change as it requires updating most of the involved artefacts (e.g. models, metamodels, and mapping). Change in the metamodels in each domain results in change in the intermediate models and the mapping. Finally for the change in domain models, the traceability model is regenerated automatically based on the mapping and new models.

4 Related Work

There are challenges associated with defining a TIM, such as finding the appropriate level of granularity; as such, TIMs are often considered to be project specific. Researchers agree on the value of a project-level definition of a TIM as it facilitates a consistent and ready-to-analyse set of traceability relations for a project [4]. As discussed in [8], project characteristics are critical in finding the *necessary and sufficient* amount of required information which should be recorded to support the traceability goals.

[9] highlights the importance of a project-specific TIM and suggests a UML-based approach to define, implement, and use it. [2] proposes a usage-centred traceability process which uses UML class diagrams to define traceability strategies for a project. [2, 10–12] each focus on traceability in the safety domain and propose traceability metamodels and queries for that domain.

Another strand of traceability research is on reducing the effort associated with managing traceability. Egyed et al. [8] introduce value-based requirements traceability to balance cost and benefits related to capturing and maintaining traceability. [13] proposes dynamic requirements traceability to minimize the need for creating and maintaining explicit links and reduce the effort required to perform manual trace. In this context, some studies take different approaches, such as improving tools in order to decrease the cost of providing traceability. [14] provides a tool-based approach for agile requirements capture and traceability.

5 Conclusion and Future Work

This paper introduced *multi-domain* traceability and highlighted its fundamental concerns. Traceability is multi-domain as it is often required to capture the artefacts and trace links either within one or across many domains. We presented a model-driven approach to constructing and managing a framework to support traceability activities. We showed how to define a TIM and express it as a traceability metamodel, to be used later to capture and record traceability information.

We observed that there are several available information sources (mainly models) which provide a considerable amount of required traceability information. A traceability model is generated as a *view* over all the other project models, providing a coherent view of traceability, and unifying the way in which information is used (e.g. traceability analyses). The *mapping* between the TM and the other models is specified and used to generate the traceability model.

We identified that change is an important concern and we need to provide more detailed and precise support. Existing model migration and co-evolution techniques could be helpful to cope with change effectively. It may also be fruitful to create TIMs using the traceability metamodelling language approach in [15].

Improved tool support for TIMs and TMs is needed. Currently, the TIM can be implemented as a metamodel in an arbitrary technology, and then existing model management languages can be used. But, as discussed in Section 3.4, providing traceability-specific tools and technical support would improve the traceability framework. A *Traceability Query Language* (TQL) to describe the traceability analyses, a DSL to specify the mapping between the main TIM and the other metamodels, and automatic generation of transformation rules based on the mapping are examples of potential improved support. We plan to work on this next while rolling out detailed case studies in domains that require traceability (particularly safety, health informatics, and security).

References

1. Mäder, P., Gotel, O., Philippow, I.: Motivation Matters in the Traceability Trenches. In: Proc. RE'09. (2009) 143–148
2. Cleland-Huang, J., Heimdahl, M., Hayes, J.H., Lutz, R., Maeder, P.: Trace Queries for Safety Requirements in High Assurance Systems. In: Proc. REFSQ'12. (2012) 179–193
3. Ramesh, B., Jarke, M.: Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering **27** (2001) 58–93
4. Mäder, P., Cleland-Huang, J.: A Visual Traceability Modeling Language. In: Proc. MoDELS'10. (2010) 226–240
5. Bézivin, J., Bouzitouna, S., Fabro, M.D.D., Gervais, M.P., Jouault, F., Kolovos, D., Kurtev, I., Paige, R.F.: A canonical scheme for model composition. In: Proc. ECMDA-FA'06. (2006) 346–360
6. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. 2. edn. Addison-Wesley, Boston, MA (2009)
7. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book. (2010)
8. Egyed, A., Grunbacher, P., Heindl, M., Biffl, S.: Value-Based Requirements Traceability: Lessons Learned. In: Proc. RE'07. (2007) 240–257
9. Mäder, P., Gotel, O., Philippow, I.: Getting Back to Basics: Promoting the Use of a Traceability Information Model in Practice. In: Proc. TEFSE'09, IEEE Computer Society (2009) 21–25
10. Peraldi-Frati, M.A., Albinet, A.: Requirement Traceability in Safety Critical Systems. In: Proc. CARS'10, ACM (2010) 11–14
11. Sanchez, P., Alonso, D., Rosique, F., Alvarez, B., Pastor, J.: Introducing Safety Requirements Traceability Support in Model-Driven Development of Robotic Applications. IEEE Transactions on Computers **60**(8) (2011) 1059 –1071
12. Katta, V., Stlhane, T.: A Conceptual Model of Traceability for Safety Systems. Technical report, Laboratory of Algorithmics, Complexity and Logic (2012)
13. Cleland-Huang, J., Settimi, R., Duan, C., Zou, X.: Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability. In: Proc. RE'05, IEEE Computer Society (2005) 135–144
14. Lee, C., Guadagno, L.: FLUID: Echo Agile Requirements Authoring and Traceability. In: Proc. MWSEC'03. (2003) 50–61
15. Matragkas, N.: Establishing and Maintaining Semantically Rich Traceability: A Metamodelling Approach. PhD thesis, University of York (2011)

A Hyperdense Semantic Domain for Discontinuous Behavior in Physical System Models

Pieter J. Mosterman, Gabor Simko, Justyna Zander

MathWorks, Vanderbilt University, HumanoidWay

Abstract. Multiple time models have been proposed for the formalization of hybrid dynamic system behavior. The superdense notion of time is a well-known time model for describing event-based systems where several events can occur simultaneously. Hyperreals provide a domain for defining the semantics of hybrid models that is elegantly aligned with first principles in physics. This paper discusses the value of both time models and shows how approximating different physical effects is best expressed over different domains. Finally, the formalization and interaction of two types of discontinuities observed in hybrid systems, mythical modes and pinnacles, are explored. This analysis helps specify semantics that combine continuous-time behavior with discontinuities in the computational system.

1 Introduction

In recent history, the complexity of engineered systems has grown by leaps and bounds, largely because of embedded computation. While embedded systems are well understood and supported by Model-Based Design [14], *Cyber-Physical Systems* (CPS) build on a general paradigm of 'openness' [18] that challenges the current paradigm of system design. This openness manifests, for example, by an application that may execute on different platforms or feature functionality that may be provided by distinctly separate systems. Because it is open, such a CPS cannot rely on integration testing (e.g., [20]) as it is part of the traditional paradigm for embedded system design.

Given the delicate interaction between various component and subsystem behaviors in their implementation, addressing system integration challenges with models is not straightforward. In particular, it is essential to create 'good' models of the physics, that is, models that embody correctly the pertinent physical effects while not giving rise to behaviors that have no physical manifestation. The desiderata for a formalism to model physical systems thus require domain-specific models that inherently reflect the laws of physics. Moreover, the models of the physics must be employable in concert with models of various paradigms such as those for computational and networking functionality in the overall system.

For system-level studies, physics models are generally well described by continuous-time behavior (e.g., based on the foundations of thermodynamics [3, 6]).

At this level, however, physical phenomena that are often part of actuators on the interface with the information technology domain (e.g., electrical switches, hydraulic valves, clutches) typically operate at a time scale too fast to be captured in continuous detail. Instead, such fast behavior is modeled as discontinuous change.

The formalization of behaviors in physical system models builds on a semantic domain that combines evolution in a continuous domain, extended with an integer domain for sequences of mode changes [5, 13]. The resulting $\mathbb{R}\times\mathbb{N}$ domain, so-called *superdense time* [11], however, is not sufficiently rich to allow a precise mathematical description of the intricate behavior in physical system models around discontinuities.[1] Specifically, an ontology of behavior in hybrid dynamic system models of physical systems developed in previous work [15] includes a class of behaviors called *mythical modes* [16] that maps well onto superdense time. However, another class of behaviors called *pinnacles* requires physical time to advance during discontinuous change, and, therefore, is not amenable to employing superdense time as a semantic domain.

Related work [1, 2, 9] has turned to *hyperreals* from *nonstandard analysis* [10] to define semantics of hybrid dynamic systems. In this paper, the hyperreals are considered as a semantic domain that supports pinnacles. In combination with an integer domain for mythical modes, this leads to a *hyperdense* time domain that supports the various classes of behavior found in hybrid dynamic system models of physical systems. The mathematical formalization is mapped onto a computational implementation that allows for generation of consistent and physically meaningful behavior of interactions between various classes of discontinuities.

Section 2 presents the notion of continuous-time interacting with discontinuities in physical system models. Bond graphs are the formalism to represent these phenomena. Further, pinnacles and mythical modes are introduced in detail and related to the notions of superdense and hyperreal time. Section 3 discusses the interactions among the different modes. Semantics of discontinuous change is explained based on Newton's cradle modeled as bond graphs. Section 4 concludes.

2 Discontinuities in Physical System Models

At a macroscopic level, physical systems are well modeled as continuous-time systems [7]. Continuous phenomena that occur at a time scale much faster than the behavior of interest can be approximated by discontinuities. This section first introduces *bond graphs* [17] as a formalism to model the continuous-time behavior of physical systems. Next, an ideal switching element is added to represent the discontinuity and form *hybrid bond graphs* [12].

[1] Note that superdense time is typical in a computational approximation of the mathematical representation. However, floating point numbers then represent the continuous domain and the approximation of the continuous domain is in fact not *dense*.

2.1 Bond Graphs

Across physics domains (e.g., electrical, hydraulic, thermal, chemical, etc.) thermodynamics identifies two types of variables subject to dynamic behavior representing either: (i) *extensive* quantities or (ii) *intensive* quantities. The dynamics of extensities and intensities are related by *conduction*, that is, when there is a difference in intensities, a change in extensity follows. For example, a difference in velocities between two bodies results in a force acting between them that causes a change in momentum ($F = m\frac{dv}{dt} = \frac{dp}{dt}$). The change of energy, *power*, as the product of the intensity difference, *effort*, and its corresponding change in extensity *flow* then provides a general notion of dynamics across physics domains. For example, $v \cdot F$ equates power much like in the electrical domain the product of the intensity difference (voltage, v) and change of extensity (current i) equates power. Consequently, any change in dynamic variable values is the result of an effort, e, and a flow, f, acting. Moreover, there are two basic energy-based phenomena: (i) storage of either effort (C) or flow (I) and (ii) dissipation (R). Finally, ideal sources of effort (Se) and of flow (Sf) define the model context.

Behavior of the connections then relates the efforts and the flows of all interacting phenomena such that the sum of their product equates 0 (so there is neither dissipation nor storage), $\sum_i e_i \cdot f_i = 0$. The two orthogonal implementations of this are that either all efforts are the same while the flows sum to zero, or the converse. In the electrical domain, this corresponds to either Kirchhoff's current law or Kirchhoff's voltage law. In bond graph terminology these connections are represented by *junctions*, the former by a 0 junction ($\forall_{i \neq j} e_i = e_j$ and $\sum_i f_i = 0$) and the latter by a 1 junction ($\forall_{i \neq j} f_i = f_j$ and $\sum_i e_i = 0$).

Introducing discontinuities into bond graphs requires an idealized form of discontinuous change in dynamic behavior, which is well represented by a reconfiguration of the junction structure because this structure is ideal. This idealized reconfiguration amounts to a junction between phenomena being active or not [19]. In other words, a 0 junction can be active ($\forall_{i \neq j} e_i = e_j$ and $\sum_i f_i = 0$) or not ($\forall_i e_i = 0$) and a 1 junction exhibits the dual behavior when active ($\forall_{i \neq j} f_i = f_j$ and $\sum_i e_i = 0$) or not ($\forall_i f_i = 0$). Note that when a junction is not active, indeed no power flows across it. These junctions that can change their mode from active (*on*) to inactive (*off*) are called *controlled junctions.*

2.2 The Logic of Discontinuities in Physics Models

A controlled junction is equipped with a finite state machine (FSM) that determines the junction *on* or *off* mode, which involves capturing: (i) how the state of the FSM maps onto the *on* and *off* mode of the junction and (ii) how the physical quantities map onto transition conditions of the FSM. Continuity of power implies that discontinuities in physical quantities result from a lack of detail in modeled phenomena, which come in two classes: (i) storage and (ii) dissipation. The discontinuous behavior that emerges in turn for each of these is discussed next.

Pinnacles Multibody collisions are often modeled by discontinuous velocity changes. In a hybrid bond graph model, a collision between two bodies, m_1 and m_2, can be modeled as depicted in Fig. 1. The two bodies are modeled as inertias, I, connected to a common velocity, 1, junction. These junctions represent the respective velocities, v_1 and v_2, which are connected via a common force, 0, junction. This 0 junction is controlled and when *off* it exerts force 0 on both bodies. Upon collision, the 0 junction turns *on* and it now enforces a velocity balance such that $v_1 - v_2 + \Delta v = 0$, where Δv is computed by an ideal flow source, Sf, as $\Delta v = v_1^- - v_2^-$, with the '-' superscript referring to signals immediately preceding the collision.

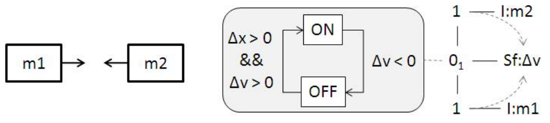


Fig. 1. Ideally plastic collision

The FSM controlling the *on*/*off* mode of the 0 junction switches from *off* to *on* when the bodies make contact ($\Delta x > 0$) and when they are moving toward one another ($\Delta v > 0$). Here the $\Delta v > 0$ is essential to model that there is a collision as opposed to the bodies only being in contact. As soon as the bodies move away from one another ($\Delta v < 0$), the 0 junction switches to *off*, irrespective of whether the bodies are touching.

During behavior generation, when $\Delta x > 0$ && $\Delta v > 0$ holds, a collision occurs and the flow source enforcing the velocity difference Δv^- becomes active. Based on this velocity difference and conservation of momentum ($\sum_i m_i v_i^- = \sum_i m_i v_i$), the velocities upon collision can be computed. The state of the velocity of the bodies is then reinitialized and this leads to the condition $\Delta v < 0$ being satisfied. Thus, a consecutive mode change occurs where the FSM moves to the *off* mode again. In the *off* mode the bodies behave as independent masses, and, therefore, no further changes in the physical state occur. Since the discrete mode changes have thus converged, the system proceeds to evolve in continuous time.

The end result is that the bodies m_1 and m_2 evolve according to a mode of continuous evolution. With a point in time at which two mode changes occur: (i) first, a collision mode occurs that necessitates a reinitialization (discontinuous change) and (ii) second, the system changes back to a mode of continuous evolution. The collision mode that is active only as a reinitialization of physical state is referred to as a *pinnacle* [13].

Mythical Mode Change Now, consider two bodies m_1 and m_2 with m_2 at rest on top of m_1. When at a point in time a large enough external force is exerted on m_1, m_1 will start moving with a corresponding velocity. However, if the force is sufficiently large that the breakaway friction force $F_{breakaway}$ between m_1 and m_2 is exceeded, m_2 may remain at rest.

A hybrid bond graph model of such a system is depicted in Fig. 2. An ideal source of effort exerts a force on m_1 because connected to the common velocity 1 junction that represents the velocity of m_1. When *on*, a controlled 0 junction connects the 1 junction that represents the velocity of m_2, which forces m_1 and m_2 to move with the same velocity. The FSM for the controlled 0 junction shows that the junction changes to its *off* mode when the force between m_1 and m_2 exceeds the breakaway force, $F > F_{breakaway}$. In the *off* mode, the 0 junction exerts 0 force on both m_1 and m_2, and so they move independently. The FSM also shows that if the velocity difference between m_1 and m_2 falls below a threshold velocity ($\Delta v < v_{th}$) the two bodies 'stick' to each other again.

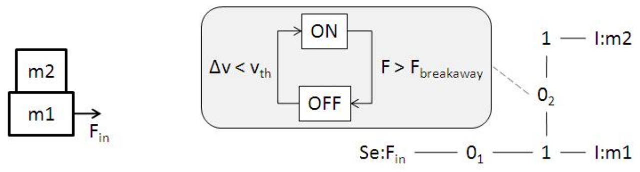


Fig. 2. Two bodies with a breakaway force

During behavior generation, initially the 0 junction is in its *on* mode because the bodies are at rest with one atop the other and the system evolves in continuous time. Now, at the point in time where F_{in} changes discontinuously new velocities for both m_1 and m_2 are computed. These velocities, however, may require a force to be exerted on m_2 that causes the condition $F > F_{breakaway}$ to be satisfied and the 0 junction changes to its *off* mode. In the *off* mode, if the velocity difference is sufficiently large, no further mode changes occur and the system proceeds to evolve in continuous time.

At the point in time at which a discontinuous force is exerted the corresponding velocities and forces are computed and based on the newly computed values the connection between the two bodies changes mode such that they are dynamically independent. Since there is no effect of the external force on the velocity of m_2, in order to arrive at the proper values for reinitialization of v_1 and v_2 the mode where the external force becomes active while m_1 and m_2 are still connected is considered to have no effect on the physical state, which is referred to as a *mythical mode* [13].

2.3 Introduction to Superdense Time

Time-event sequence is a semantic domain for describing event-based models. Intuitively, time-event sequences are instantaneous events separated by non-negative real numbers that describe time durations between the events. Events separated by zero duration are simultaneous, but have a well-defined causal ordering.

Superdense time was introduced to represent time-event sequences as functions of time [11]. Superdense time is a totally ordered subset of $\mathbb{R}_+ \times \mathbb{N}$, where the non-negative real number represents the real time and the natural number represents the causal ordering. Simultaneous events at time t are mapped to $(t, 0), (t, 1), \ldots$ superdense time instants such that the ordering of the events is preserved.

The (total) ordering of superdense time is given by the following definitions: $(t, n) = (t', n') \Leftrightarrow t = t' \wedge n = n'$, and $(t, n) < (t', n') \Leftrightarrow t < t' \vee (t = t' \wedge n < n')$. Therefore, superdense time is a time model that can be used to describe simultaneous events as functions of time, while retaining the causality of events.

Mythical modes emerge as an artifact of logical inference to determine a new mode in which physical state can change. As such, mythical modes do not affect the dynamic state of a physical system. Moreover, different logic formulations may traverse different mythical modes yet still arrive at the same resulting mode where physical state changes can occur. Consequently, the logical evaluation has no corresponding manifestation in the dynamic state of a physical system and occurs at a single point in time along a logical inferencing dimension. This behavior corresponds to the superdense semantic domain.

2.4 Introduction to Hyperreal Time

Calculus comprises two different approaches to capturing infinitely small values: either through the use of limits, or by the extension of the field of reals with infinitesimals. An infinitesimal ϵ is any number, such that $|\epsilon| < \frac{1}{n}$, for any $n \in \mathbb{N}$.

Intuitively, the idea behind hyperreals is to extend the dense field of $\mathbb{R}$ with infinitely many points around each real number such that any real sentence that holds for one or more real functions also holds for the hyperreal natural extensions of these functions [10] (transfer principle).

In the ultrapower construction [8], hyperreals are represented as sequences of real numbers $u_1, u_2, \ldots u_n \in \mathbb{R}^n$ with real numbers embedded as constant sequences (i.e., a real number r is the sequence of $r, r, \ldots r \in \mathbb{R}^n$). These sequences, together with elementwise addition and multiplication operations, form a commutative ring but not a field (since the multiplication of two non-zero numbers could result in zero: $0, 1, 0, \ldots \times 1, 0, 1, \ldots = 0, 0, \ldots$). This issue is remedied by considering equivalence classes of $\mathbb{R}^n$ defined by a free ultrafilter U of $\mathbb{N}$.

Let J be a non-empty set. An ultrafilter on J is a nonempty collection U of subsets of J having the following properties: $\emptyset \notin U$; $A \in U$ and $B \in U$ implies $A \cap B \in U$; $A \in U$ and $A \subseteq B \subseteq J$ implies $B \in U$; for all $A \subseteq J$, either $A \in U$ or $J \setminus A \in U$. For any $x \in J$ there is a principal ultrafilter $\{A \subseteq J \mid x \in A\}$. Finally, any non-principal ultrafilter is called a free ultrafilter.

Given an ultrafilter U, an equivalence relation $=_U$ can be defined over $\mathbb{R}^n$: $u =_U v$ holds for sequences $u = u_1, \ldots, u_n$ and $v = v_1, \ldots, v_n$ if and only if $\{i \mid u_i < v_i\} \in U$. The hyperreals are then defined as the quotient of $\mathbb{R}^n$ by U, ${}^*\mathbb{R} = \mathbb{R}^n/U$. Now, ${}^*\mathbb{R}$ is an ordered field for which the transfer principle holds.

As a semantic domain, hyperreals have the advantage that between any *real* time instant there are many ordered time instants. Such extension of time greatly simplifies the semantic specification of discontinuities, in particular, the description of pinnacles that represent fast physical behaviors where the dynamic state changes discontinuously. As a result, a pinnacle corresponds to a distinct state of physical behavior. In physics, such a distinct state corresponds to a distinct point in time. Because the continuous behavior represented by a pinnacle is considered to occur infinitely fast, time is considered to advance by an infinitesimal amount for a pinnacle to implement the physical state change. This behavior corresponds to the hyperreal domain.

It is a straightforward extension to introduce a hyperdense time model as a "combination" of the super-dense and hyperreal time models. We define the hyperdense time $^*\mathbb{R}_+ \times \mathbb{N}$ as the product of the non-negative hyperreals and natural numbers. Such a time model can be used for representing both infinitesimal time advancements, as well as establishing a causal ordering at any hyperreal time instant.

3 Semantics of Discontinuity Behavior

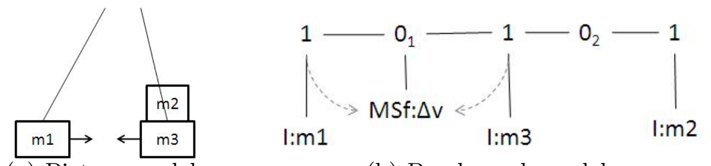
The formalized models of time provide the ingredients for a semantic domain that is sufficiently rich to formalize the pinnacle and mythical mode behavior at discontinuities as well as combinations.

3.1 Interacting Pinnacles and Mythical Modes

With superdense time as a semantic domain for mythical modes and hyperreals for pinnacles, models that engender both build on a combined hyperdense semantic domain. The particular value of such a precise semantic description lies in the ability to develop consistent computational behavior generation algorithms. Because of the discreteness of computational values, the semantic domain of values in computational models can represent neither superdense nor hyperreal domains. Therefore, the behavior generation algorithms must include sophistication that addresses the differences between superdense and hyperreal semantic domains. The computational implementation of each of the semantic domains and their interaction is described based on an illustrative example.

In Fig. 3(a), a variant of Newton's Cradle is shown. One of the bodies, m_3, has another body, m_2, positioned on top of it. Stiction effects between m_2 and m_3 cause them to behave as one body with combined mass as long as the breakaway force between them, $F_{breakaway}$, is not exceeded. A body, m_1, may collide with m_3 according to a perfectly elastic collision, $\Delta v_{32} = -\epsilon \Delta v_{32}^{-}$, where Δv is the difference in velocities $(v_3 - v_2)$ after the collision and Δv_{32}^{-} is the difference in velocities before the collision.

The bond graph model in Fig. 3(b) shows the three masses each connected to a common velocity junction, 1, with the velocity of the directly connected mass on all ports. Common force junctions, 0, connect the 1 junctions and are



(a) Picture model (b) Bond graph model

Fig. 3. Newton's Cradle for advanced maneuvers

controlled junctions such that a finite state machine determines their *on* or *off* state. A modulated flow source *MSf* models the collision with restitution $\epsilon = 0.8$. If the controlled junction 0_1 is in its *on* state, this flow source enforces a difference in velocities of m_1 and m_3, possibly accounting for the rigidly connected mass m_2. If the controlled junction 0_1 is in its *off* state, a 0 force is exerted on both m_1 and m_3 (possibly accounting for m_2). In its *off* state, the controlled junction 0_2 exerts a 0 force as well, which is when the force at the contact point between m_2 and m_3 exceeds the breakaway force. Note that for the case when m_2 and m_3 move independently in continuous time no viscous friction is modeled for clarity purposes. If the difference in velocities between m_2 and m_3 falls below a threshold level, the stiction effect becomes active, modeled by 0_2 changing to its *on* state. In the *on* state, a difference in velocities of m_2 and m_3 of 0 is enforced.

Upon collision of m_1 and m_3, if the difference in velocities v_2 and v_3, Δv_{23}, is less than the threshold velocity v_{th}, stiction is active and m_2 and m_3 behave as one body with mass m_2+m_3. When $m_2+m_3 > m_1$, m_1 will have a return velocity and start moving in the opposite direction compared to the velocity before the collision. However, the momentum of m_1 may be such that an impulsive force [4] arises between m_2 and m_3 that triggers the $F_{breakaway}$ transition, causing the two bodies to move independently. In this case, if $m_1 = m_3$, there is no return velocity of m_1 but instead it acts as in the case of Newton's Cradle where m_3 assumes all of the momentum of m_1 while m_1 comes to rest. In this case the velocity of m_2 is not affected by the collision.

The importance of semantic domain that combines both superdense as well as hyperreals is clearly illustrated by this example. While the condition for 0_2 to switch from *on* to *off* occurs in 0 time, the condition for 0_1 to switch from *on* to *off* occurs in infinitesimal, ϵ, time. A critical consequence of this phenomenon is that, although in reasoning about the system, 0_1 first changes its state to *on*, after which the change of state in 0_2 to *off* is determined, the change of state in 0_1 back to *off* is not effected until *after* the change of state in 0_2 to *off*.

In Table 1 and in Table 2, 0_{F13} and 0_{F23} are the junctions at a select force impact, while p_{m1}, p_{m2}, and p_{m3} are the momenta for each of the masses. The sequences of mode changes are depicted in Table 2, which shows clearly the difference in effects as the model evolves in superdense and in hyperreal time. The ability to differentiate between $t = \langle t_{collide}, 1\rangle$ and $t = \langle t_{collide} + \epsilon, 0\rangle$ makes it possible to distinguish the pinnacle effect of 0_1 from the mythical mode effect of 0_2. Otherwise, 0_1 would have switched back *off* simultaneously with 0_2 switching

time	0_{F13}	0_{F23}	p_{m1}	p_{m2}	p_{m3}
$t = \langle t_{collide} - \epsilon, 0\rangle$	*off*	*on*	1	0	0
$t = \langle t_{collide}, 0\rangle$	*on*	*on*	-0.2	0.6	0.6
$t = \langle t_{collide} + \epsilon, 0\rangle$	*off*	*on*	-0.2	0.6	0.6

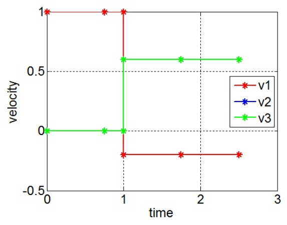


Table 1. Mode changes for $v_{th}^2 = 0.1$ and $F_{th} = 0.95$

time	0_{F13}	0_{F23}	p_{m1}	p_{m2}	p_{m3}
$t = \langle t_{collide} - \epsilon, 0\rangle$	*off*	*on*	1	0	0
$t = \langle t_{collide}, 0\rangle$	*on*	*on*	-0.2	0.6	0.6
$t = \langle t_{collide}, 1\rangle$	*on*	*off*	0.1	0.9	0.0
$t = \langle t_{collide} + \epsilon, 0\rangle$	*off*	*off*	0.1	0.9	0.0

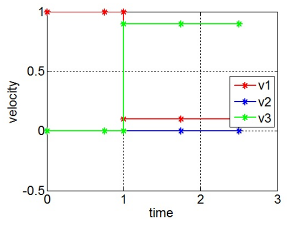


Table 2. Mode changes for $v_{th}^2 = 0.1$ and $F_{th} = 0.5$

off. This would either: (i) not allow modeling of inferencing (mythical) modes or (ii) have the collision effect (incorrectly) computed for m_2 and m_3 comprising a combined mass of $m_2 + m_3$.

4 Conclusions

Superdense and hyperreal time notions provide a comprehensive basis for formalizing a computational semantics. The interplay between them enables the design of models that include continuous-time behavior interacting with discontinuities of physical system models. Such formalization is of great value, in particular in simulation technologies, because of the benefits in a consistent projection of behavior according to the laws of physics into a computational representation.

Based on the bond graph modeling formalism, a formalization is developed for combining continuous-time with discontinuities. The combination provides a theoretical reference for the computational integration of different effects of discontinuities observed across multiple domains. Moreover, the work allows for a consistent mapping onto corresponding algorithms that lack hyperreals as execution domain. Most prominently, the research addresses how to combine behavior because of logical switching with physics-based switching behavior.

Formalization in this domain often foregoes the collision mode by reinitializing velocities as a transition action in a state machine. Though there is no fundamental difference in behavior generation, such a representation makes it exceedingly complicated to attribute a sound theory of physics to discontinuous behavior. Instead, this paper relates pinnacles and mythical modes to different notions of time so as to formalize their interaction in a computational sense.

References

1. Albert Benveniste, Timothy Bourke, Benôit Caillaud, and Marc Pouzet. Non-standard semantics of hybrid systems modelers. *Journal of Computer and System Sciences*, 78(3):877–910, May 2012.
2. Simon Bliudze and Daniel Krob. Modelling of complex systems: Systems as dataflow machines. *Fundamenta Informaticae*, 91:1–24, 2009.
3. Peter C. Breedveld. *Physical Systems Theory in Terms of Bond Graphs.* PhD dissertation, University of Twente, Enschede, Netherlands, 1984.
4. Bernard Brogliato. *Nonsmooth Mechanics.* Springer-Verlag, London, 1999. ISBN 1-85233-143-7.
5. Krister Edström. *Switched Bond Graphs: Simulation and Analysis.* PhD dissertation, Linköping University, Sweden, 1999.
6. Gottfried Falk and Wolfgang Ruppel. *Energie und Entropie: Eine Einführung in die Thermodynamik.* Springer-Verlag, Berlin, 1976. ISBN 3-540-07814-2.
7. Oliver Heaviside. On the forces, stresses, and fluxes of energy in the electromagnetic field. *Proceedings of the Royal Society of London*, 50:126–129, 1891.
8. Albert E Hurd and Peter A Loeb. *An introduction to nonstandard real analysis.* Academic Press, 1985.
9. Yumi Iwasaki, Adam Farquhar, Vijay Saraswat, Daniel Bobrow, and Vineet Gupta. Modeling time in hybrid systems: How fast is "instantaneous"? In *Intl. Conf. on Qualitative Reasoning*, pp. 94–103, Amsterdam, May 1995.
10. H. Jerome Keisler. *Elementary Calculus: An Infinitesimal Approach.* Prindle, Weber and Schmidt, Dover, 3 edition, 2012.
11. Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In *Real-Time: Theory in Practice*, LNCS, pp. 447–484. Springer, 1992.
12. Pieter J. Mosterman and Gautam Biswas. Behavior generation using model switching a hybrid bond graph modeling technique. In *Intl. Conf. on Bond Graph Modeling and Simulation* (ICBGM '95), pp. 177–182, Las Vegas, January 1995.
13. Pieter J. Mosterman and Gautam Biswas. A theory of discontinuities in dynamic physical systems. *Journal of the Franklin Institute*, 335B(3):401–439, January 1998.
14. Pieter J. Mosterman, Sameer Prabhu, and Tom Erkkinen. An industrial embedded control system design process. In *Proceedings of The Inaugural CDEN Design Conference (CDEN'04)*, Montreal, Canada, July 2004. CD-ROM: 02B6.
15. Pieter J. Mosterman, Feng Zhao, and Gautam Biswas. An ontology for transitions in physical dynamic systems. In *AAAI98*, pp. 219–224, July 1998.
16. T. Nishida and S. Doshita. Reasoning about discontinuous change. In *Proceedings AAAI-87*, pp. 643–648, Seattle, Washington, 1987.
17. Henry M. Paynter. *Analysis and Design of Engineering Systems.* The M.I.T. Press, Cambridge, Massachusetts, 1961.
18. Steering Committee. Foundations for Innovation: Strategic Opportunities for the 21st Century Cyber-Physical Systems—Connecting computer and information systems with the physical world. Technical report, NIST, March 2013.
19. Jan-Erik Strömberg, Jan Top, and Ulf Söderman. Variable causality in bond graphs caused by discrete effects. In *Proc. of the Intl. Conf. on Bond Graph Modeling* (ICBGM'93), pp. 115–119, San Diego, California, 1993.
20. Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman, editors. *Model-Based Testing for Embedded Systems.* CRC Press, Boca Raton, FL, 2011. ISBN: 9781439818459.

A DSL for Explicit Semantic Adaptation [⋆]

Bart Meyers[1], Joachim Denil[3], Frédéric Boulanger[2],
Cécile Hardebolle[2], Christophe Jacquet[2], and Hans Vangheluwe[1,3]

[1] MSDL, Department of Mathematics and Computer Science
University of Antwerp, Belgium
[2] Supélec E3S, Department of Computer Science, France
[3] MSDL, School of Computer Science, McGill University, Canada

Abstract. In the domain of heterogeneous model composition, semantic adaptation is the "glue" that is necessary to assemble heterogeneous models so that the resulting composed model has well-defined semantics. In this paper, we present an execution model for a semantic adaptation interface between heterogeneous models. We introduce a Domain-Specific Language (DSL) for specifying such an interface explicitly using rules, and a transformation toward the ModHel'X framework. The DSL enables the modeller to easily customise interfaces that fit the heterogeneous model at hand in a modular way, as involved models are left untouched. We illustrate the use of our DSL on a power window case study and demonstrate the importance of defining semantic adaptation explicitly by comparing with the results obtained with Ptolemy II.

1 Introduction

The growing power of modelling tools allows the design and verification of complex systems. This complexity leads to Multi-Paradigm Modelling [1], because different parts of the system belong to different technical domains, but also because different abstraction levels, different aspects of the system, and different phases in the design require different modelling techniques and tools. It is therefore necessary to be able to cope with multi-paradigm, heterogeneous models, and to give them a semantics which is precise enough for simulating the behaviour and validating properties of the system, and to generate as much as possible of the realisation of the system from the model.

A major challenge in model composition is to obtain a meaningful result from models with heterogeneous semantics. Tackling this issue requires that the semantics of the modelling languages used in the composed models is described in a way such that it can be processed and, more importantly, such that it can be composed. In this article, we focus on the definition of the "composition laws" for heterogeneous parts of a model, which have to be explicitly described. A key point of our approach is modularity: we should be able to compose heterogeneous models without modifying them.

To illustrate what these composition laws, which we call *semantic adaptation*, have to deal with, we first introduce the example of a power window and

[⋆] Thanks to Dr. Thomas Feng for giving us insights into Ptolemy II's discrete events semantics show how it can be modelled and simulated using existing tools for heterogeneous modelling. We introduce a meta-model for modelling semantic adaptation, founded on an execution model. We then present a Domain-Specific Language (DSL) that we designed to specify semantic adaptation in an abstract and compact way. We then discuss the advantages and drawbacks of this approach and conclude, giving some perspective for future work.

2 The Power Window Case Study

The system we model to illustrate our work is a car power window, composed of a switch, a controller board and an electromechanical part. These components communicate through the car's bus. The controller receives commands from the switch and information from the end stops of the electromechanical part, and sends commands to switch the motor off, up or down. If the user presses the switch for a short time (less than 500 ms), the controller fully closes or opens the window until it reaches an end stop.

For simplicity, we don't model the bus as a component of the system, and we consider that the components of the model communicate through discrete events (DE). The controller is modelled as a state machine, with timed transitions to distinguish between short and long presses on the switch. The dynamics of the electromechanical part could be modelled using differential equations, but for making the example easier to understand, we discretise its behaviour and represent it using difference equations, for which a synchronous dataflow model (SDF) is suitable. As a result, the global model of the power window system is a heterogeneous model involving three different modelling paradigms: discrete events, timed finite state machines and synchronous dataflow.

In the following, we show how this system is modelled using different tools for heterogeneous modelling and we illustrate how these different tools deal with the necessary semantic adaptation among the heterogeneous parts of this model.

3 Related Work and Semantic Adaptation

We chose to focus our study of the state of the art on three different tools for heterogeneous modelling and simulation: Ptolemy II [2], Simulink/Stateflow[4] and ModHel'X [3]. All of them support the joint use of different modelling paradigms in a model and they all use hierarchy as a mechanism for composing the heterogeneous parts of a model. Other types of approaches are described in [4].

In Simulink/Stateflow, the power window system has been studied at different levels of detail and the resulting models are available as demos in the tool[5]. We have modelled the power window system using Ptolemy II and ModHel'X, and the resulting models are available online[6].

[4] `http://www.mathworks.fr/products/simulink/`

[5] See the online documentation at `http://www.mathworks.fr/fr/help/simulink/examples/simulink-power-window-controller-specification.html`.

[6] Ptolemy II model: `http://wwwdi.supelec.fr/software/downloads/ModHelX/power_window_fulladapt.moml`
ModHel'X model: `http://wwwdi.supelec.fr/software/ModHelX/PowerWindow`

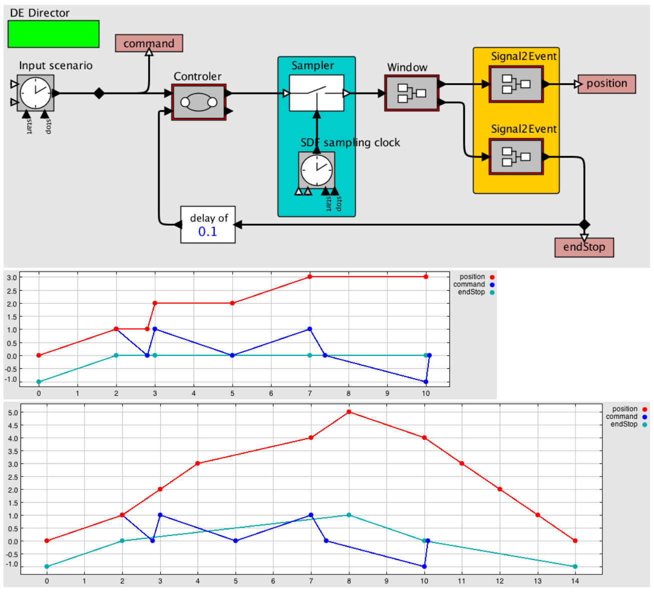


Fig. 1. Ptolemy II model of the power window

Using Ptolemy II for modelling the power window system, we obtain the model shown on Figure 1. In this model, the DE Director box tells that this model is built according to the Discrete Events *model of computation*, or MoC. A MoC defines the execution rules obeyed by the components of a model. The DE MoC considers each component as a process triggered by input events and producing events on the output. The Input scenario block models a simulation scenario for the output of the switch. The controller is modelled as a state machine (FSM MoC), and the window as a data-flow block diagram (Synchronous Data Flow MoC). Due to space constraints, these models are not shown in the paper.

Ptolemy II does not provide a default semantic adaptation between heterogeneous parts of a model. For instance, the Window component, which is modelled using SDF, is considered as a DE component by the DE director. It is therefore activated each time it receives an input event, and each data sample it produces is considered as a DE event on its outputs. This model corresponds to what is shown in Figure 1 without the two large rounded boxes (which deal with semantic adaptation and will be presented below). The result of the simulation without semantic adaptation (i.e., without the two large rounded boxes) is shown just below, in the middle part of the figure. The red line on top shows the position of the window (from 0 to 5), the green line at the bottom shows the end-stop signal (-1 is open, 1 is closed, 0 is in between), and the middle, blue line shows the command played by the scenario (1 is for closing the window, -1 for opening it, and 0 for stopping it). However, the behaviour is not as expected: the window model (using difference equations) is not sampled periodically, meaning that the window is not going up or down with a constant speed, and the model produces consecutive events with the same value.

The problem is that the SDF nature of the model of the window makes it behave inconsistently in a DE environment. Semantic adaptation is needed to sample it periodically, to provide it with correct inputs (as achieved with the left blue box) and to filter its outputs (as achieved with the right orange box). The lowest graph in Figure 1 shows the result with the adaptations. In this case the behaviour is as expected. The window model is sampled at a periodic rate and does not produce duplicate events.

The two highlighted areas were added to the DE model to specify the semantic adaptation with the SDF model of the behaviour of the window. However, their contents depends on the internals of the window model, so the global model is not modular. Moreover, without the highlighting we put on the figure, it is not easy to make a difference between a block which is part of the model and a block which is there for semantic adaptation. Semantic adaptation is therefore diffuse and difficult to specify and understand.

In Simulink/Stateflow, which is a powerful and efficient simulation tool for heterogeneous models, semantic adaptation is even more diffuse. The global semantics of a heterogeneous model (for instance a Simulink model including a Stateflow submodel) is given by one solver which is used to compute its execution. The solver uses parameters of the blocks and their ports, in particular a parameter called "sample time"[7], to determine when to compute the value of the different signals. The resulting execution of the model depends on the value of these parameters and on the type and parameters of the solver itself. Even if default values for these parameters are determined by the simulation engine, to adapt the resulting execution semantics to his needs, the user must have a deep understanding of the solver's mechanisms and fine tune different simulation parameters. A part of the semantic adaptation can also be performed using blocks like in Ptolemy II or truth tables or functions, but with the same drawbacks.

ModHel'X is a framework for modelling and executing heterogeneous models [3] which we developed in previous works. ModHel'X allows for modular semantic adaptation by explicitly defining an *interface block* between the parent model and the embedded model. This block adapts the data, control and time between the different models [5]. In the power window example, we saw examples of the adaptation of data with the filtering of the output signals of the window model, and of control with the periodic sampling. The adaptation of time is necessary when two models use different time scales. Interface blocks are similar to tag conversion actors as presented in [9], but can perform more complex operations. In ModHel'X, control and time adaptation rely on a model of time which is inspired from the MARTE UML profile [6, 7], and can be considered as a restriction of the Clock Constraints Specification Language [7], extended with time tags [8]. An issue with ModHel'X is that the semantic adaptation is specified using calls to a Java API in different methods of an interface block. Therefore, constructing an interface block is a tedious and error prone process. To aid developers in defining interface blocks, we propose a Domain Specific Language for modelling explicitly the adaptation of data, time and control.

[7] http://www.mathworks.fr/fr/help/simulink/ug/types-of-sample-time.html

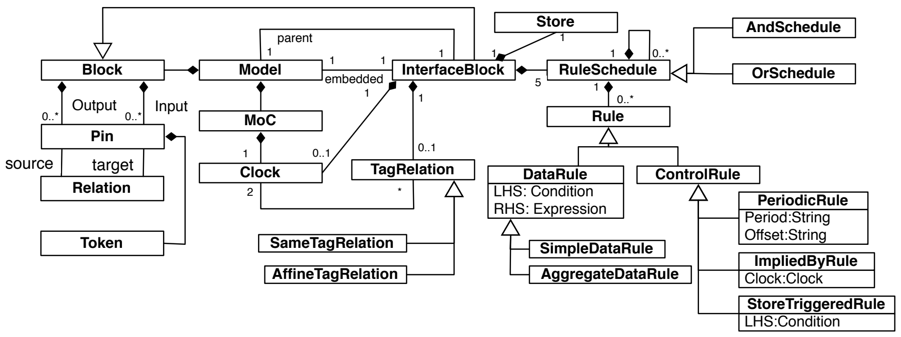


Fig. 2. Partial meta-model of ModHel'X with a focus on the interface block

4 A DSL and Execution Semantics for Semantic Adaptation

The DSL for semantic adaptation provides the modeller with the concepts that are necessary for specifying the glue between heterogeneous models. By insulating the modeller from platform specific issues, it allows him to focus on the problem at stake, and it also reduces the semantic gap between what should be specified and how it is realised. However, the current implementation of the DSL generates code for ModHel'X, so we have to give some background about it.

4.1 The Interface Block Meta-Model

Figure 2 shows a simplified, partial meta-model of ModHel'X with a focus on the interface block. The structure of models in ModHel'X is similar to Ptolemy II's: Models contain Blocks with input and output Pins that can be connected. During execution, values flow through models as Tokens on Pins. A Model has a model of computation (MoC) which defines its execution semantics, similar to a director in Ptolemy II. In ModHel'X a Block can be an InterfaceBlock, containing an embedded Model with any MoC, thus allowing heterogeneous modelling. In an InterfaceBlock, there can be adaptation Relations between input Pins of the InterfaceBlock and input Pins of the embedded Model, and between output Pins of the embedded Model and output Pins of the InterfaceBlock. An InterfaceBlock and a MoC can have a Clock, with its own time scale. An InterfaceBlock has a Store that can hold data, thus adding the concept of state.

Our DSL for describing the behaviour of an interface block is based on rules, which are evaluated and return true when they match. DataRules and ControlRules are used to match conditions for the semantic adaptation of data and control. The order in which rules are evaluated is controlled by a RuleSchedule. Two different scheduling policies (which are hierarchical) are defined: the AndSchedule evaluates the rules in order as long as they return true, and it returns true if all rules returned true; the OrSchedule evaluates the rules in order until one returns true, in which case true is returned.

Data rules are responsible for the adaptation of the data from the parent model to the embedded model and vice versa. Data rules have a left-hand side (LHS), denoting pre-condition for the applicability of the rule and a right-hand side (RHS) providing the needed action when the LHS is matched. Data rules

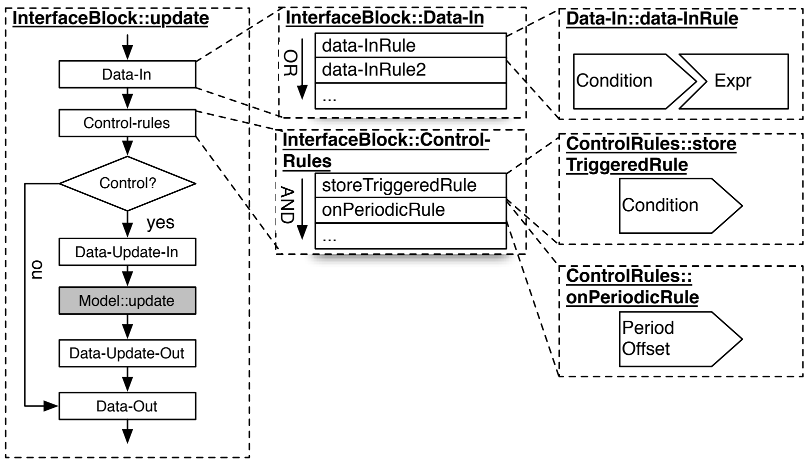


Fig. 3. Execution of an interface block

have access to pins and a store of the interface block which serves as a map of variables (variable name - value pairs). Control rules are responsible for executing the embedded model (also called *giving control*) at the current time instant. Control rules have access to the store to create complex trigger events but they also access the interface block clock to create an observation request in the future. For the semantic adaptation of time, an interface block can contain TagRelations between clocks, which allows to convert dates between their time scales.

4.2 Execution Semantics in Five Phases

As shown in Figure 3, the interface block adapts data and control in five different phases. Each phase has its own schedule (containing a set of rules), and each phase is provided with a set of pins as parameters so that, besides the store, token values on pins become accessible in the rules. In case of the Data-in, Control-rules and Data-update-in phases, parameter pins are the input pins of the interface block. In case of the Data-update-out or Data-out phases the parameter pins are the output pins of the embedded model. The phases are executed in the following order when control has been given to the interface block:

Data-in: (data rules) when the interface block receives control, rules are executed to update the store of the block according to the incoming data tokens. Complex operations can transform the data for further processing;

Control: (control rules) Control rules decide if control should be passed to the embedded model at the current instant according to the store of the block and the data on the input pins. Control rules can also create new observation request on the clock of the interface block so that it receives control later;

Data-update-in: (data rules) if the control rules give control to the embedded model, the Data-update-in schedule is executed before control is passed to the embedded model. These rules provide the internal model with correct inputs according to the values in the store and the data on the input pins of the interface block;

Data-update-out: (data rules) after the execution of the embedded model, the Data-update-out rules are used to update the store with the data available on the output pins of the embedded model;

Data-out: (data rules) finally, the Data-out rules are in charge of producing the outputs of the interface block from the data available in the store. This phase is executed even when control is not given to the embedded model because the interface block may have to produce some outputs any way.

To ease the creation of rules, we identified some common rule patterns, shown as subclasses of DataRule and ControlRule in Figure 2:

– SimpleDataRule: this data rule is evaluated for every parameter pin. If its LHS evaluates to true, its RHS is executed;
– AggregateDataRule: this data rule allows for more complex patterns over multiple parameter pins, and will only be evaluated once. If its LHS evaluates to true, its RHS is executed;
– Periodic: this control rule periodically gives control to the interface block, and variable names are given for period and offset so that they can be set when the interface block is used in a model. The clock calculus in ModHel'X will make sure that the interface block is updated at the specified moments;
– StoreTriggeredRule: this control rule gives control to the embedded model if its LHS (with access to the store) evaluates to true;
– ImpliedByRule: this control rule gives control to the embedded model whenever a given clock is triggered.

A DSL implementing this meta-model, the presented semantics and a concrete textual syntax are defined in metaDepth [10]. The solution includes an ANTLR-based parser, an EOL script for generating code for the rules, and an EGL script [11] for generating Java code for ModHel'X. These steps ensure that the transformation from DSL code to Java code is entirely automated. Due to space constraints, the meta-model, models or scripts in metaDepth are not shown.[8] The concrete syntax of the DSL will be illustrated in the next section.

5 A Semantic Adaptation DSL in Practice

In this section, we reconstruct the two different semantic adaptations between DE and SDF for the power window system presented in section 3, and of which traces are shown in Figure 1. The models presented in this section are transformed using the metaDepth framework into ModHel'X models in Java code.

Listing 1 and 2 show interface blocks for both behaviours as a DSL model. The first 8 lines state the structural properties of the interface block and are the same for both behaviours. Line 1 presents the model name. On line 2-3, the Java class and package are specified for the code generator. Line 4 presents the clock of the interface. In this case the interface has a timed clock, meaning that events occur at specific dates on a time scale (as opposed to an untimed clock, where events have no date). Line 5-6 and 7-8 respectively present the external (parent) model and internal (embedded) model. The parent model is called `de`

[8] The fully operational solution presented in this paper can however be downloaded from `http://msdl.cs.mcgill.ca/people/bart/PowerWindow.zip`

and adheres the ModHel'X `AbstractDEMoC` with a timed clock named `deClock`. Similarly, the embedded model uses SDF, which is by nature untimed.

Listing 1. Model of the default Ptolemy II behaviour

```
 1 InterfaceBlock {
 2   IMPLEMENTS "DE_SDF_InterfaceBlock"
 3          IN "tests.powerwindow"
 4   TIMED CLOCK ibClock
 5   EXTERNAL MODEL de (
 6     "AbstractDEMoC" WITH TIMED CLOCK deClock)
 7   INTERNAL MODEL sdf (
 8     "SDFMoC" WITH UNTIMED CLOCK sdfClock)
 9   RULES:
10     IN
11     CONTROL
12     UPDATEIN
13       FORALLPINS ALWAYS ->
14                     FORWARD VALUE TO SUCCESSORS;
15     UPDATEOUT
16       FORALLPINS ALWAYS ->
17                     FORWARD VALUE TO SUCCESSORS;
18     OUT
19 }
```

Listing 2. Model of the interface block with semantic adaptation

```
 1 InterfaceBlock {
 2   IMPLEMENTS "DE_SDF_InterfaceBlock"
 3          IN "tests.powerwindow"
 4   TIMED CLOCK ibClock
 5   EXTERNAL MODEL de (
 6     "AbstractDEMoC" WITH TIMED CLOCK deClock)
 7   INTERNAL MODEL sdf (
 8     "SDFMoC" WITH UNTIMED CLOCK sdfClock)
 9   RULES:
10     IN
11       FORALLPINS ON DATA: ALWAYS -> TOSTORE(VALUE);
12     CONTROL
13       PERIODIC AT "init_time" EVERY "period"
14     UPDATEIN
15       FORALLPINS ALWAYS ->
16                     FORWARD FROMSTORE TO SUCCESSORS;
17     UPDATEOUT
18       FORALLPINS ON DATA:
19         VALUE != FROMSTORE ->
20                     TOSTORE(VALUE);
21                     FORWARD VALUE TO SUCCESSORS;
22     OUT
23   TAG RELATION deClock = ibClock
24 }
```

From line 9 onward, the rules that model the execution semantics are specified. These differ in both models. The five rule phases, in their execution order, can be recognised from line 10 onward.

The default Ptolemy II behaviour that results in the upper trace of Figure 1 does little semantic adaptation. This behaviour is defined in Listing 1. No particular Data-in rules are given, since the store is not used. There is no Control rule, because whenever the interface block receives control, it will immediately give control to the embedded model. On Data-update-in, the tokens on the input ports are forwarded with no adaptation to the embedded model. This is modelled as the data rule on line 13-14, which should be read as *LHS* → *RHS*. As a SimpleDataRule (denoted by `FORALLPINS`), it is executed for every input pin of the interface block. The LHS is `ALWAYS`, as there are no restrictions on when to forward data. The RHS specifies that the current token value (`VALUE`) should be forwarded to the embedded model's input pins to which the interface input pins are connected to (`SUCCESSORS`). In the Data-update-out phase on line 16-17, the tokens generated by the embedded model are similarly forwarded in another SimpleDataRule. As mentioned before, in this phase `VALUE` and `SUCCESSORS` – which depend on the parameter pins – refer to the token values and the successors of the embedded model's output pins. In summary, the basic Ptolemy model forwards all data instantaneously.

The intended behaviour of the power window's interface that results in the bottom trace of Figure 1 is shown in Listing 2. The interface block periodically samples the SDF model. Data is provided to the embedded model by applying zero-order hold (ZOH) from the input pins of the parent model to the input pins of the embedded model, meaning that the last known value should be used. The Data-in rule on line 11 saves the data from the input pins in the store, each time a token is received (`ON DATA`). The Periodic Control rule on line 13 specifies the periodic sampling of the embedded model, starting at time "init_time", with a step of "period", which can be set in the instance model. The Data-update-in rule on line 15-16 implements the ZOH functionality by providing the input pins of the embedded model with data from the store. Note that first-order hold (linear extrapolation) could also be modelled by calculating new values based on the previous ones (stored the store) every time the interface block is updated. A Data-update-out rule (line 18-21) checks for every pin whether the embedded model's output value changed, by comparing it to the previous one, which was stored. Initially the value in the store is *null* making sure the LHS evaluates to true. So only when the output of the embedded model changes, tokens are forwarded to the parent model. No event is consequently produced when the window is inactive, as seen in the trace. Finally a SameTagRelation on line 23 specifies that time is measured the same way in the interface block and in the embedding model. Note that in both models we never had to explicitly schedule rules, as every phase contained at most a single rule.

6 Discussion

Our DSL for semantic adaptation allows one to *explicitly* specify the adaptation behaviour intended at the boundary between two heterogeneous parts of a model. This is an improvement in two respects. First, compared to the *implicit* adaptation mechanisms embedded in Ptolemy or Simulink, the semantics is clearly stated. Second, compared to the solution introduced in Section 3 that relies on the modification of the models to introduce new blocks devoted to adaptation, our approach leaves the models unchanged. This improves modularity, as a given model may be embedded as is in different contexts; what needs to be changed each time is just the adaptation layer defined using our DSL.

When embedding a timed finite state machine (TFSM) into a DE model in Ptolemy, one has to weave adaptation mechanisms in the guards and actions of the state machine in order to translate between TFSM events and DE values. Even if this does not introduce new blocks, this is indeed a variant of the second issue outlined above: one has to modify the model to include adaptation, which hinders modularity. We used the proposed DSL to model the adaptation of the DE model to the TFSM model of the power window system. The DSL is able to model all the needed constructs for the interface block and generate Java code. Due to space constraints, the model is not included in the paper. Still, some constructs necessary for the adaptation of a wide variety of MoCs may not be available in the DSL yet. Future enhancements of our DSL will remedy this.

The improvements mentioned here (specifying adaptation between models explicitly while keeping the modularity of models) have been available in ModHel'X for a few years. However in ModHel'X adaptation is coded in Java, so there is a semantic gap between high-level adaptation mechanisms and the way it is actually written as low-level code that makes minute manipulations of data structures. Moreover, an adapter in ModHel'X is scattered in several methods, which makes its behavior hard to follow. The five phased execution semantics and the DSL are focused and concise: they bridge this semantic gap. More than 300 lines of Java code are reduced to less than 25 lines of DSL code.

7 Conclusion

In the domain of Multi-Paradigm Modelling, semantic adaptation is the "glue" that gives well-defined semantics to the composition of heterogeneous models. In this paper we proposed a DSL with execution semantics to bridge the cognitive gap between the implementation and specification of a semantic interface block. The model of an interface block explicitly specifies the adaptation of data, control and time using a set of rules. Furthermore, a model-to-text transformation is defined to generate the interface block code for the ModHel'X framework. The DSL enables the modeller to easily customise interface blocks that fit the heterogeneous models in a modular way, as involved models are left untouched.

The approach was illustrated on the power window case study by reconstructing in ModHel'X both the implicit Ptolemy II adaptation and the specific one needed between DE and SDF to obtain the expected behaviour. We believe that the DSL and the execution semantics are, as a principle, expressive enough to model different semantic adaptations of heterogeneous models, though additional constructs might be needed for additional behaviours.

References

1. P. J. Mosterman and H. Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *SIMULATION*, 80(9):433–450, 2004.
2. J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming Heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.
3. Frédéric Boulanger and Cécile Hardebolle. Simulation of Multi-Formalism Models with ModHel'X. In *ICST 2008*, pages 318–327, 2008.
4. Cécile Hardebolle and Frédéric Boulanger. Exploring multi-paradigm modeling techniques. *SIMULATION* , 85(11/12):688–708, 2009.
5. F. Boulanger, C. Hardebolle, C. Jacquet, and D. Marcadet. Semantic Adaptation for Models of Computation. In *ACSD'11*, pages 153–162, 2011.
6. Charles André, Frédéric Mallet, and Robert De Simone. Modeling Time(s). In *MoDELS'07*, volume LNCS 4735, pages pp. 559–573. Springer, 2007.
7. R. Gascon, F. Mallet, and J. Deantoni. Logical time and temporal logics: comparing UML MARTE/CCSL and PSL. In *TIME'11*, pages 141–148, Lubeck, Germany, 2011.
8. F. Boulanger, C. Hardebolle, C. Jacquet, and I. Prodan. Modeling time for the execution of heterogeneous models. Technical report 2013-09-03-DI-FBO, Supélec E3S, 2012.
9. P. Caspi, A. Benveniste, R. Lublinerman, and S. Tripakis. Actors without directors: A kahnian view of heterogeneous systems. In R. Majumdar and P. Tabuada, editors, *Hybrid Systems: Computation and Control*, volume 5469 of *LNCS*, pages 46–60. Springer Berlin Heidelberg, 2009.
10. Juan de Lara and Esther Guerra. Deep meta-modelling with metadepth. In *TOOLS (48)*, volume 6141 of *LNCS*. Springer Berlin Heidelberg, 2010.
11. Juan de Lara and Esther Guerra. Domain-specific textual meta-modelling languages for model driven engineering. In *ECMFA'12*, pages 259–274, 2012.

A Multiparadigm Approach to Integrate Gestures and Sound in the Modeling Framework

Vasco Amaral[1], Antonio Cicchetti[2], Romuald Deshayes[3]

[1] Universidade Nova de Lisboa, Portugal vasco.amaral@fct.unl.pt
[2] Malardalen Research and Technology Centre (MRTC), Vasteras, Sweden antonio.cicchetti@mdh.se
[3] Software Engineering Lab, Université de Mons-Hainaut, Belgium romuald.deshayes@umons.ac.be

Abstract. One of the essential means of supporting Human-Machine Interaction is a (software) language, exploited to input commands and receive corresponding outputs in a well-defined manner. In the past, language creation and customization used to be accessible to software developers only. But today, as software applications gain more ubiquity, these features tend to be more accessible to application users themselves. However, current language development techniques are still based on traditional concepts of human-machine interaction, i.e. manipulating text and/or diagrams by means of more or less sophisticated keypads (e.g. mouse and keyboard).

In this paper we propose to enhance the typical approach for dealing with language intensive applications by widening available human-machine interactions to multiple modalities, including sounds, gestures, and their combination. In particular, we adopt a Multi-Paradigm Modelling approach in which the forms of interaction can be specified by means of appropriate modelling techniques. The aim is to provide a more advanced human-machine interaction support for language intensive applications.

1 Introduction

The mean of supporting Human-Machine interaction are languages: a well-defined set of concepts that can be exploited by the user to compose more or less complex commands to be input to the computing device. Given the dramatic growth of software applications and their utilization in more and more complex scenarios, there has been a contemporary need to improve the form of interaction in order to reduce users' effort. Notably, in software development, it has been introduced different programming language generations, programming paradigms, and modelling techniques aiming at raising the level of abstraction at which the problem is faced. In other words, abstraction layers have been added to close the gap between machine language and domain-specific concepts, keeping them interconnected through automated mechanisms.

While the level of abstraction of domain concepts has remarkably evolved, the forms of interaction with the language itself indubitably did not. In particular, language expressions are mainly text-based or a combination of text and diagrams. The underlying motivation is that, historically, keyboard and mouse have been exploited as standard input devices. In this respect, other forms of interaction like gestures and sound have been scarcely considered. In this paper we discuss the motivations underlying the need of enhanced forms of interaction and propose a solution to integrate gestures and sound in modeling frameworks. In particular, we define *language intensive* applicative domains the cases where either the language evolves rapidly, or language customizations are part of the core features of the application itself.

In order to better grasp the previously mentioned problem, we consider a sample case study in the Home Automation Domain, where a language has to be provided as supporting different automation facilities for a house, ranging from everyday life operations to maintenance and security. This is a typical language intensive scenario since there is a need to customize the language depending on the customer's building characteristics. Even more important, the language has to provide setting features enabling a customer to create users' profiles: notably, children may command TV and lights but they shall not access kitchen equipment. Likewise, the cleaning operator might have access to a limited amount of rooms and/or shall not be able to deactivate the alarm.

In the previous and other language intensive applicative domains it is inconceivable to force users to exploit the "usual" forms of interaction for at least two reasons: i) if they have to digit a command to switch on the lights they could use the light switch instead and hence would not invest money in these type of systems. Moreover, some users could be unable to exploit such interaction techniques, notably disabled, children, and so forth; ii) the home automation language should be easily customizable, without requiring programming skills. It is worth noting that language customization becomes a user's feature in our application domain, rather than a pure developer's facility. If we widen our reasoning to the general case, the arguments mentioned so far can be referred to as the need of facing accidental complexity. Whenever a new technology is proposed, it is of paramount importance to ensure that it introduces new features and/or enhances existing ones, without making it more complex to use, otherwise it would not be worth to be exploited.

Our solution is based on Multi-Paradigm Modelling (MPM) principles, i.e. every aspect of the system has to be appropriately modeled and specified, combining different points of view of the system being then possible to derive the concrete application. In this respect, we propose to precisely specify both the actions and the forms of language interactions, in particular gestures and sound, by means of models. In this way, flexible interaction modes with a language are possible as well as languages accepting new input modalities such as sounds and gestures.

The remaining of the paper is organized as follows: sec. 2 discusses the state-of-the-art; sec. 3 presents the communication between a human and a machine through a case study related to home automation; sec. 4 introduces an interaction metamodel and discusses how it can be used to generate advanced concrete syntaxes relying on multiple modalities; finally, sec. 5 concludes.

2 State of the Art

This section describes basic concepts and state-of-the-art techniques typically exploited in sound/speech and gesture recognition together with their combinations created to provide advanced forms of interaction. Our aim is to illustrate the set of concepts usually faced in this domain and hence to elicit the requirements for the interaction language we will introduce in Section 4.

Sound and speech recognition

Sound recognition is usually used for command-like actions; a word has to be recognized before the corresponding action can be triggered. With the Vocal Joystick [1] it is possible to use acoustic phonetic parameters to continuously control tasks. For example, in a WIMP (Windows, Icons, Menus, Pointing device) application, the type of vowel can be used to give a direction to the mouse cursor, and the loudness can be used to control its velocity.

In the context of environmental sounds, [2, 3] proposed different techniques based on Support Vector Machines and Hidden Markov Models to detect and classify acoustic events such as foot steps, a moving chair or human cough. Detecting these different events help to better understand the human and social activities in smart-room environments. Moreover, an early detection of non-speech sounds can help to improve the robustness of automatic speech recognition algorithms.

Gesture recognition

Typically, gesture recognition systems resort to various hardware devices such as data glove or markers [4], but more recent hardware such as the Kinect or other 3D sensors enable unconstrained gestural interaction [5]. According to a survey of gestural interaction [6], gestures can be of 3 types :

- hand and arm gestures: recognition of hand poses or signs (such as recognition of sign language);
- head and face gestures: shaking head, direction of eye gaze, opening the mouth to speak, happiness, fear, etc;
- body gestures: tracking movements of two people interacting, analyzing movement of a dancer, or body poses for athletic training.

The most widely used techniques for dynamic gestural recognition usually involve hidden Markov models [7], particle filtering [8] or finite state machines [9].

Multimodal systems

The "Put-that-there" [10] system, developed in the 80's, is considered to be the origin of human-computer interaction regarding the use of voice and sound. Vocal commands such as "delete this elements" while pointing at one object displayed on the screen can be correctly processed by the system. According to [11], using multiple modalities, such as sound and gestures, helps to make the system more robust and maintainable. They also proposed to split audio sounds in two categories: human speech and environmental sounds.

Multimodal interfaces involving speech and gestures have been widely used for text input, where gestures are usually used to choose between multiple possible utterances or correct recognition errors [12, 13]. Some other techniques propose to use gestures on a touchscreen device in addition to speech recognition to correct recognition errors [14]. Both modalities can also be used in an asynchronous way to disambiguate between the possible utterances [15].

More recently, the SpeeG system [16] has been proposed. It is a multimodal interface for text input and is based on the Kinect sensor, a speech recognizer and the Dasher [17] user interface. The contribution lies in the fact that, unlike the aforementioned techniques, the user can perform speech correction in real-time, while speaking, instead of doing it in a post processing fashion.

Human-computer interaction modeling

In the literature, many modeling techniques have been used to represent interaction with traditional WIMP user interfaces. For example, statecharts have been dedicated to the specification and design of new interaction objects or widgets [18].

Targetting virtual reality environment, Flownets [19] is a modeling tool, relying on high-level Petri nets, based on a combination of discrete and continuous behavior to specify the interaction with virtual environments. Also based on high-level Petri nets for dynamic aspects and an object oriented framework, the Interactive Cooperative Objects (ICO) formalism [20] has been used to model WIMP interfaces as well as multimodal interactions in virtual environments. In [4], a virtual chess game was developed in which the user can use a data glove to manipulate virtual chess pieces. In [21], ICO has been used to create a framework for describing gestural interaction with 3D objects has been proposed.

Providing a UML based generic framework for modeling interaction modalities such as speech or gestures enables software engineers to easily integrate multimodal HCI in their applications. That's the point defended by [22]. In their work, the authors propose a metamodel which focuses on the aspects of an abstract modality. They distinguish between simple and complex modalities. The first one represents a primitive form of interaction while the second integrates other modalities and uses them simultaneously. With the reference point being the computer, input and output modalities are defined as a specification of simple modality. Input modalities can be event-based (e.g. performing a gesture or sending a vocal command) or streaming based (e.g. drawing a circle or inputting text using speech recognition) and output modalities are used to provide static (e.g. a picture) or dynamic (e.g. speech) information to the user.

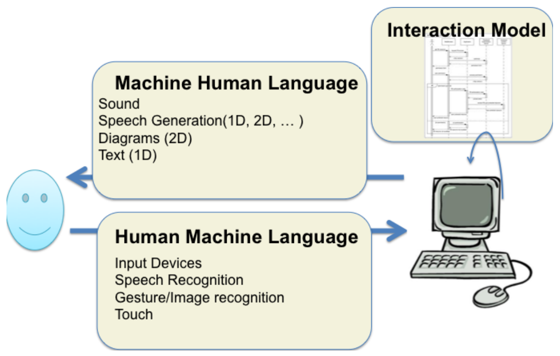


Fig. 1. A more advanced support of Human-Machine Interaction

3 Human-Machine Communication

As discussed so far, different techniques supporting more advanced forms of interaction between humans and machines have already been proposed. Nonetheless, their exploitation in current software languages has been noticeably limited. This work proposes to widen the modalities of human-machine interaction as depicted in Fig. 1. In general, a human could input information by means of speech, gestures, texts, drawings, and so forth. The admitted ways of interaction are defined in an interaction model, which also maps human inputs into corresponding machine readable formats. Once the machine has completed its work, it outputs the results to the user through sounds, diagrams, texts, etc.; also in this case the interaction model prescribes how performed computations should be rendered to a human comprehensible format.

A software language supports the communication between humans and machines by providing a set of well-defined concepts that typically abstract real-life concepts. Hence, software language engineering involves the definition of three main aspects: i) the internal representation of the selected concepts, understandable by the machine and typically referred to as *abstract syntax*; ii) how the concepts are rendered to the users in order to close the gap with the applicative domain, called *concrete syntax*; iii) how the concepts can be interpreted to get/provide domain-specific information, referred to as *semantics*. Practically, a metamodel serves as a base for defining the structural arrangement of concepts and their relationships (abstract syntax), the concrete syntax is “hooked” on appropriate groups of its elements, and the semantics is generically defined as computations over elements.

In order to better understand the role of these three aspects, Fig. 2 and 1 illustrate excerpts of the abstract and concrete syntaxes, respectively, of a sample Home Automation DSL. In particular, a home automation system manages a `House` (see Fig. 2 right-hand side) that can have several rooms under domotic control (i.e. `Room` and `DomoticControl` elements in the metamodel, respectively).

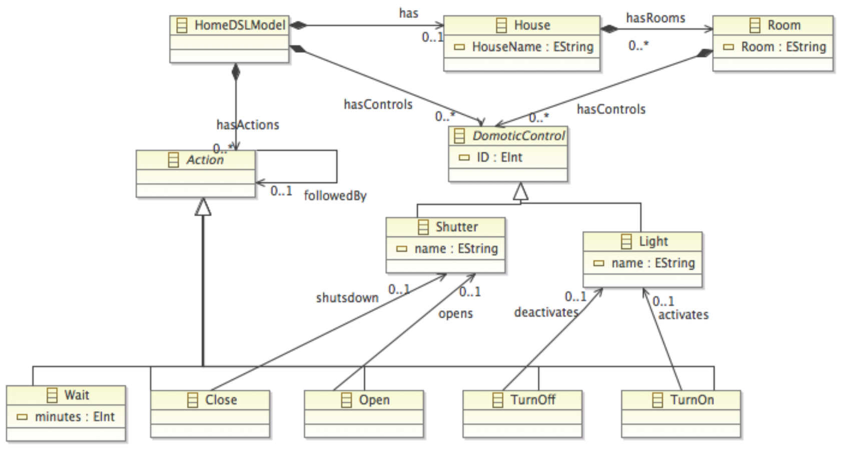


Fig. 2. Metamodel (abstract syntax) of a simple illustration DSL for Home Automation

The control is composed by several devices and actions that can be performed with them. Notably, a `Light` can be turned on and off, while a `Shutter` can be opened and closed.

It is easy to notice that the abstract syntax representation would not be user-friendly in general, hence a corresponding concrete syntax can be defined in order to provide the user with easy ways of interaction. In particular, Prog. 1 shows an excerpt of the concrete syntax definition for the home automation DSL using the Eugenia tool[4]. The script prescribes to depict a `House` element as a graph node showing the rooms defined for the house taken into account.

Prog. 1 Concrete Syntax mapping of the DSL for Home Automation with Eugenia

```
@gmf.node(label="HouseName", color="255,150,150", style="dash")
class House {
  @gmf.compartment(foo="bar")
  val Room[*] hasRooms;
  attr String HouseName;
}
```

Despite the remarkable improvements in language usability thanks to the addition of a concrete syntax, the malleability of interaction modalities provided by Eugenia and other tools (e.g. GMF[5]) is limited to the standard typing and/or

[4] http://www.eclipse.org/epsilon/doc/eugenia/
[5] http://www.eclipse.org/modeling/gmp/

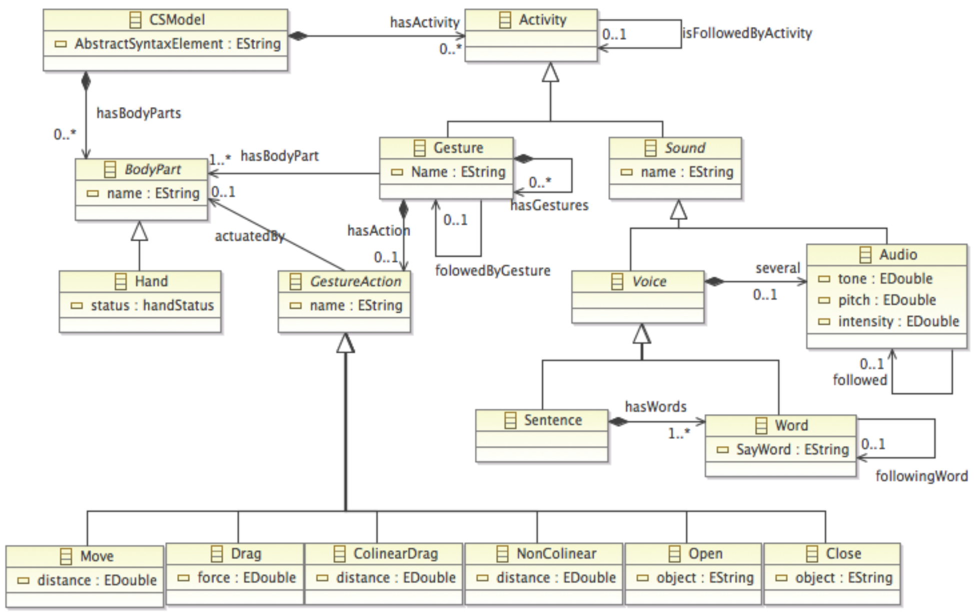


Fig. 3. A possible language to define enhanced concrete syntaxes for DSL

drawing graphs. Such a limitation becomes evident when needing to provide users with extended ways of interaction, notably defining concrete syntaxes as sounds and/or gestures. For instance, a desirable concrete syntax for the home automation metamodel depicted in Fig. 2 would define voice commands for turning lights on and off, or alternatively prescribe certain gestures to do the same operations.

By embracing the MPM vision, which prescribes to define any aspect of the modeling activity as a model, next Section introduces a language for defining advanced human-machine interactions. In turn, such a language can be combined with abstract syntax specifications to provide DSLs with enhanced concrete syntaxes.

4 Interaction Modeling

The proposed language tailored to enhanced human-machine interaction concrete syntax definition is shown in Fig. 3. In particular, it depicts the metamodel to define advanced concrete syntaxes, encompassing sound and gestures, while the usual texts writing and diagrams drawing are treated as particular forms of gestures. Going deeper, elements of the abstract syntax can be linked to (sequences of) activities (see `Activity` on the bottom-left part of Fig. 3). An activity, in turn, can be classified as a `Gesture` or a `Sound`.

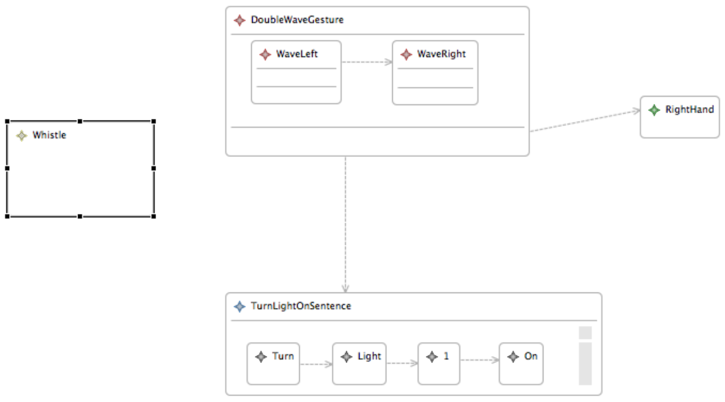


Fig. 4. A possible instance model defining the Concrete Syntax of a concept

Regarding gestures, the language supports the definition of different types of primitive actions typically exploited in gesture recognition applications. A `Gesture` is linked to the `BodyPart` that is expected to perform the gesture. This way we can distinguish between performed actions, for example, by a hand or a head. `Move` is the simplest action a body part can perform, it is triggered for each displacement of the body part. Dragging (`Drag`) can only be triggered by a hand a represents a displacement of a closed hand. `ColinearDrag` and `NonColinearDrag` represent a movement of both hands going in either the same or opposite directions while being colinear or not colinear, respectively. `Open` and `Close` are triggered when the user opens or closes the hand.

Sounds can be separated in two categories: i) `Voice` represents human speech, which is composed of `Sentences` and/or `Words`. ii) `Audio` that relates to all non speech sounds that can be encountered, such as knocking on a door, a guitar chord or even someone screaming. As it is very generic, it is characterized by fundamental aspects as `tone`, `pitch`, and `intensity`.

Complex activities can be created by combining multiple utterances of sound and gestures. For example one could define an activity to close a shutter by closing the left hand and dragging it from top to bottom while pointing at the shutter and saying “close shutters”.

In order to better understand the usage of the proposed language, Fig. 4 shows a simple example defining the available concrete syntaxes for specifying the turning the lights on command. In particular, it is possible to wave the right hand, first left and then right to switch on light 1 (see the upper part of the picture). Alternatively, it is possible to give a voice command made up of the sound sequence “Turn Light 1 On”, as depicted in the bottom part of the figure.

It is worth noting that, given the purpose of the provided interaction forms, the system can be taught to recognize particular patterns as sounds, speeches, gestures, or their combinations. In this respect, the technical problems related to recognition can be alleviated. Even more important, the teaching process discloses the possibility to extend the concrete syntax of the language itself, since additional multimodal commands can be introduced as alternative ways of interaction.

5 Conclusions

The ubiquity of software applications is widening the modeling possibilities of end users who may need to define their own languages. In this paper, we defined these contexts as language-intensive applications given the evolutionary pressure the languages are subject to. In this respect, we illustrated the needs of having enhanced ways of supporting human-machine interactions and demonstrated them by means of a small home automation example. We noticed that in general concrete syntaxes usually refer to traditional texts writing and diagrams drawing, while more complex forms of interaction are largely neglected. Therefore, we proposed to extend the current concrete syntax definition approaches by adding sounds and gestures, but also possibilities to compose them with traditional interaction modalities. In this respect, by adhering to the MPM methodology we defined an appropriate modeling language for illustrating concrete syntaxes that can be exploited later on to generate corresponding support for implementing the specified interaction modalities.

As next steps we plan to extend the Eugenia concrete syntax engine in order to be able to automatically generate the support for the extended human-machine interactions declared through the proposed language. This phase will also help in the validation of the proposed concrete syntax metamodel shown in Fig. 3. In particular, we aim at verifying the adequacy of the expressive power provided by the language and extend it with additional interaction means.

This work constitutes the base to build-up advanced modeling tools relying on enhanced forms of interaction. Such improvements could be remarkably important to widen tools accessibility to disabled developers as well as to reduce the accidental complexity of dealing with big models.

References

1. J. Bilmes, X. Li, J. Malkin, K. Kilanski, R. Wright, K. Kirchhoff, A. Subramanya, S. Harada, J. Landay, P. Dowden, and H. Chizeck, "The vocal joystick: A voice-based human-computer interface for individuals with motor impairments," in *HLT/EMNLP*. The Association for Computational Linguistics, 2005.
2. A. Temko, R. Malkin, C. Zieger, D. Macho, and C. Nadeu, "Acoustic event detection and classification in smart-room environments: Evaluation of chil project systems," *Cough*, vol. 65, p. 6, 2006.
3. S. Nakamura, K. Hiyane, F. Asano, T. Nishiura, and T. Yamada, "Acoustical sound database in real environments for sound scene understanding and hands-free speech recognition," in *LREC*. European Language Resources Association, 2000.

4. D. Navarre, P. Palanque, R. Bastide, A. Schyn, M. Winckler, L. Nedel, and C. Freitas, “A formal description of multimodal interaction techniques for immersive virtual reality applications,” in *INTERACT*, ser. Lecture Notes in Computer Science, M. F. Costabile and F. Paternò, Eds., vol. 3585. Springer, 2005, pp. 170–183.
5. Z. Ren, J. Meng, J. Yuan, and Z. Zhang, “Robust hand gesture recognition with kinect sensor,” in *ACM Multimedia*, 2011, pp. 759–760.
6. S. Mitra and T. Acharya, “Gesture recognition: A survey,” *IEEE Trans. on Systems, Man and Cybernetics - part C*, vol. 37, no. 3, pp. 311–324, 2007.
7. J. Yamato, J. Ohya, and K. Ishii, “Recognizing human action in time-sequential images using hidden Markov model,” in *Proceed. IEEE Conf. Computer Vision and Pattern Recognition*, 1992, pp. 379–385.
8. M. Isard and A. Blake, “Condensation - conditional density propagation for visual tracking,” *International Journal of Computer Vision*, vol. 29, no. 1, pp. 5–28, 1998.
9. P. Hong, T. S. Huang, and M. Turk, “Gesture modeling and recognition using finite state machines,” in *FG*. IEEE Computer Society, 2000, pp. 410–415.
10. R. Bolt, “Put-that-there: Voice and gesture at the graphics interface,” in *Proceed. 7th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH ’80. New York, NY, USA: ACM, 1980, pp. 262–270.
11. B. Demiroz, I. Ar, A. Ronzhin, A. Coban, H. Yalcn, A. Karpov, and L. Akarun, “Multimodal assisted living environment,” in *eNTERFACE 2011, The Summer Workshop on Multimodal Interfaces*, 2011.
12. N. Osawa and Y. Y. Sugimoto, “Multimodal text input in an immersive environment,” in *ICAT 2002, 12th International Conference on Articial Reality and Telexistence*, 2002, pp. 85–92.
13. K. Vertanen, “Efficient computer interfaces using continuous gestures, language models, and speech,” http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-627.pdf, Computer Laboratory, University of Cambridge, Tech. Rep. UCAM-CL-TR-627, 2005.
14. D. Huggins-Daines and A. I. Rudnicky, “Interactive asr error correction for touchscreen devices,” in *ACL (Demo Papers)*. The Association for Computer Linguistics, 2008, pp. 17–19.
15. P. O. Kristensson and K. Vertanen, “Asynchronous multimodal text entry using speech and gesture keyboards,” in *INTERSPEECH*. ISCA, 2011, pp. 581–584.
16. L. Hoste, B. Dumas, and B. Signer, “Speeg: a multimodal speech- and gesture-based text input solution,” in *AVI*, G. Tortora, S. Levialdi, and M. Tucci, Eds. ACM, 2012, pp. 156–163.
17. D. J. Ward, A. F. Blackwell, and D. J. C. MacKay, “Dasher - a data entry interface using continuous gestures and language models,” in *UIST*, 2000, pp. 129–137.
18. D. A. Carr, “Specification of interface interaction objects,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’94. New York, NY, USA: ACM, 1994, pp. 372–378. [Online]. Available: http://doi.acm.org/10.1145/191666.191793
19. S. Smith and D. Duke, “Virtual environments as hybrid systems,” in *Proceedings of Eurographics UK 17th Annual Conference (EG-UK99)*, E. U. K. Chapter, Ed., United Kingdom, 1999.
20. P. A. Palanque and R. Bastide, “Petri net based design of user-driven interfaces using the interactive cooperative objects formalism,” in *DSV-IS*, 1994, pp. 383–400.
21. R. Deshayes, T. Mens, and P. Palanque, “A generic framework for executable gestural interaction models,” in *Proc. VL/HCC*, 2013.
22. Z. Obrenovic and D. Starcevic, “Modeling multimodal human-computer interaction,” *IEEE Computer*, vol. 37, no. 9, pp. 65–72, 2004.

Towards Bidirectional Engineering of Satellite Control Procedures Using Triple Graph Grammars

Susann Gottmann[1], Frank Hermann[1], Claudia Ermel[2], Thomas Engel[1], and Gianluigi Morelli[3]

[1] Interdisciplinary Centre for Security, Reliability and Trust,
Université du Luxembourg, Luxembourg
`firstname.lastname@uni.lu`
[2] Technische Universität Berlin, Germany
`firstname.lastname@tu-berlin.de`
[3] SES, Luxembourg
`firstname.lastname@ses.com`

Abstract. The development and maintenance of satellite control software are very complex, mission-critical and cost-intensive tasks that require expertise from different domains. In order to adequately address these challenges, we propose to use visual views of the software to provide concise abstractions of the system from different perspectives.
This paper introduces a visual language for process flow models of satellite control procedures that we developed in cooperation with the industrial partner SES for the satellite control language SPELL. Furthermore, we present a general and formal bidirectional engineering approach for automatically translating satellite control procedures into corresponding process flow visualisations. The bidirectional engineering framework is supported by a visual editor based on Eclipse GMF, the transformation tool HenshinTGG, and additional extensions to meet requirements set up by the specific application area of satellite control languages.

Keywords: model transformation, model synchronisation, triple graph grammars, bidirectional engineering, Eclipse Modeling Framework (EMF)

1 Introduction

Development and maintenance of satellite control software are very complex, mission-critical and cost-intensive tasks demanding expertise from different domains. We address these challenges by a general approach we developed in a joint research project with the industrial partner SES (Société Européenne des Satellites, `http://www.ses.com/`). SES is a world-leading satellite operator currently operating a fleet of 53 satellites of different vendors. The satellite control programming language SPELL (Satellite Procedure Execution Language & Library) [23] was initiated by SES to become a new standard. It is an open-source package based on Python for the development and operation of satellite control procedures.

The main goal of the research project is to develop the visual modelling language *SPELLFlow*, which represents the control flow of SPELL procedures. Satellite engineers and operators at SES are currently working with the SPELL source code. In the development of SPELL source code, engineers already work with a visual representation of the desired source code (as printout), but it is completely uncoupled from the SPELL development and execution environment. In order to enhance the daily work and reduce cost-intensive errors, a visualisation, related to the one used by satellite engineers, is desired with further improvements: It abstracts from the source code and highlights important commands for providing a more intuitive way of input. However, it shall not lose detailed information, which will be hidden at the beginning and can be shown, if the user desires. So, we developed a layered concept (c.f. Sec. 3). SPELLFlow is adapted to the following domain specific requirements set up by SES: **(1)** Provide a hierarchical visual model defining different layers of abstraction for highlighting more important information, but without losing detailed information. **(2)** Emphasise certain SPELL statements, e.g., commands for sending and receiving telemetry data. **(3)** The bidirectional engineered model will be used as a concise visual view on the source code. Hence, the engineering process between SPELL and SPELLFlow has to yield correct visual models and has to retain functional behaviour, i.e., the translation terminates and yields the same result for identical inputs.

For the translation we use triple graph grammars (TGGs) [21,22], a bidirectional formal technique in the field of graph transformation. Several correctness properties are ensured due to the usage of TGGs (syntactical correctness of translation results, functional behaviour, completeness of translation, i.e., every input graph can be translated).

In the former successful joint research project PIL2SPELL [16] with SES, we developed an automated translation based on TGGs from satellite procedures written in the proprietary satellite operation language PIL of the satellite manufacturer ASTRIUM into SPELL. In the current project, we will reuse results, like the SPELL grammar for the conversion from concrete syntax of SPELL into the abstract syntax graph. At a later stage, we will provide an extension to a model synchronisation framework based on [15], which will be integrated at SES into the daily work of satellite developers and controllers. The bidirectional engineering framework is supported by the transformation tool HenshinTGG [8,13]. A visual editor *SPELLFlowEditor* based on Eclipse GMF [2] was generated and extended to the specific requirements of SES.

Sec. 2 introduces the running example. Sec. 3 presents the visual language SPELLFlow and the approach for the translation. Sec. 4 summarises the applied formal techniques. In Sec. 5 we discuss related work and conclude in Sec. 6.

2 Running Example

```
def fib(n):
  a = 0
  b = 1
  for i in range(n):
    sum = a + b
    b = a
    a = sum
  #ENDFOR
  Prompt('Result: ' + a, OK)
  return
#ENDDEF

Step('1', 'User Input')
nr = Prompt('Number: ', NUM)
fib(nr)
if(Prompt('Restart?',YES_NO) == YES):
  Goto('1')
#ENDIF
```

Fig. 1. Example Procedure in *SPELL* syntax

Throughout the paper, we use the SPELL source code in Fig. 1 as running example. It is not satellite-specific but well-known and complex enough to explain all details of the approach. The program prints the Fibonacci number of a given user input. Lines 1 - 11 depict the subroutine `fib(n)`, which determines the Fibonacci number for the given parameter `n` and outputs the result. In line 13, the main program starts with the SPELL-specific command `Step`. This command indicates a label (first parameter) used for jumps and provides a description (second parameter). Line 14 asks for user input to be given as parameter to the subroutine in line 15. Afterwards, the user is able to decide, whether she wants to restart the procedure. If the user answers the prompt with `YES`, then the application jumps back to line 13 using the Goto command.

3 Methodology

This section describes the general approach for the bidirectional engineering of satellite control procedures written in SPELL into its flow visualisation and vice versa, We introduce the desired visualisation, which is developed in cooperation with SES and present the approach for implementing the bidirectional translation.

Desired Visualisation

In Sec. 1, we introduced requirements for the visualisation. To fulfil requirement 1 (hierarchical visual model), we developed a layered model containing different abstractions. In practice, it represents the call-hierarchy of a diagram out of another diagram, i.e., we provide the possibility to go from one more abstract layer to an underlying one which contains more fine-grained details. The first layer shows only relevant control structure parts of the main procedure - with the industrial partner SES, we elaborated special rules for defining the first layer out of the source code: the first branch of `if`, `for`, `while` or `try` statements, function calls, `Goto` and `Step` commands shall be situated on the first layer. The second layer will contain more detailed information, e.g., further branches, body of functions called on the previous layer. In general, the richness of detail is increasing with a growing layer depth. In the visual representation, shapes for statements of the same type which directly follow each other, are merged (see Ex. 1).

Requirement 2 (focus on telemetry data) is realised by specific shapes for very important SPELL statements (e.g., send and receive telemetry commands, steps, prompts).

Example 1 (SPELLFlow - concrete syntax). Fig. 2 illustrates the visualisation of the SPELL procedure (Fig. 1). Note, that we use the diagram number on the right top of each box in the following description. The first layer (diagram 1) contains the main procedure according to the rules mentioned before. The box following `Step 1 : User Input` belongs to the `Step` command and links to another diagram (number 2) on second layer, which contains a statement (assignment) that is allocated to this step but should not occur on first layer. If there are further statements, which should not occur on the first layer, boxes with + icons are shown signalling links to further layers. The `Goto` is visualised by a pentagon shape which links to the target statement - the step statement. In diagram 1, the box `Call fib(nr)` indicates a function call. It links to

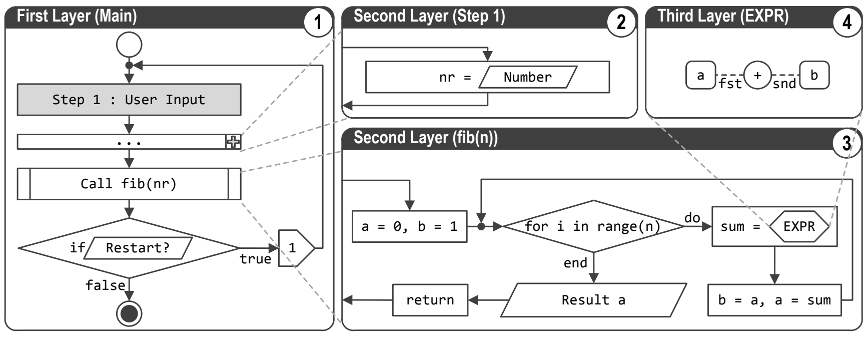


Fig. 2. Desired visualisation for the example procedure in *SPELL*

diagram 3, which is situated on second layer. The content of the function is represented by diagram 3. The shapes of statements of the same type are merged, e.g., in box `b = a, a = sum`. The `Prompt` statement gets a special shape (rhomboid), and expressions (hexagon), that are at least binary. Expressions are depicted explicitly in separate diagrams on the next layer. Consequently, hexagon `a + b` links to diagram 4, which shows the expression in full detail on third layer.

General Bidirectional Engineering Approach

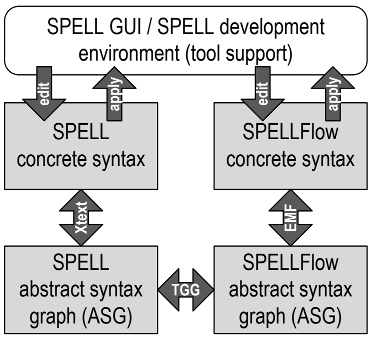

SPELL GUI / SPELL development environment (tool support)
edit
apply
edit
apply
SPELL concrete syntax
SPELLFlow concrete syntax
Xtext
EMF
SPELL abstract syntax graph (ASG)
TGG
SPELLFlow abstract syntax graph (ASG)

Fig. 3. Bidirectional engineering from *SPELL* to *SPELLFlow* and vice versa

The approach for bidirectional engineering of SPELL (Fig. 3) from source code to its visualisation and vice versa will be integrated in two SES applications: the SPELL execution environment , which is used for operating satellites and the SPELL development environment, in which the SPELL programmer gets the possibility to implement SPELL procedures in the source code view and also in using the visualisation of SPELLFlow models for creating a skeleton as a way of code generation. Both SPELL environments are represented by the rounded rectangle on top of Fig. 3. Currently, SES uses SPELL source code - the concrete syntax of SPELL. For the bidirectional engineering process from SPELL source code to visual SPELL models (SPELLFlow), we use the Eclipse tool HenshinTGG, which is based on EMF [3]. The concrete syntax of SPELL will be imported in HenshinTGG using Xtext [4] which results in an abstract syntax graph (ASG) of the SPELL source code. We use HenshinTGG to define triple rules and generate forward translation rules (for the translation from SPELL ASG to SPELLFlow ASG) and generate backward translation rules (for the translation from the SPELLFlow ASG to the SPELL ASG). Using EMF, the abstract syntax graph of SPELLFlow is transferred to the concrete syntax, an XMI file, which can be imported into the SPELLFlowEditor to visualise the flow diagram. The SPELLFlowEditor for

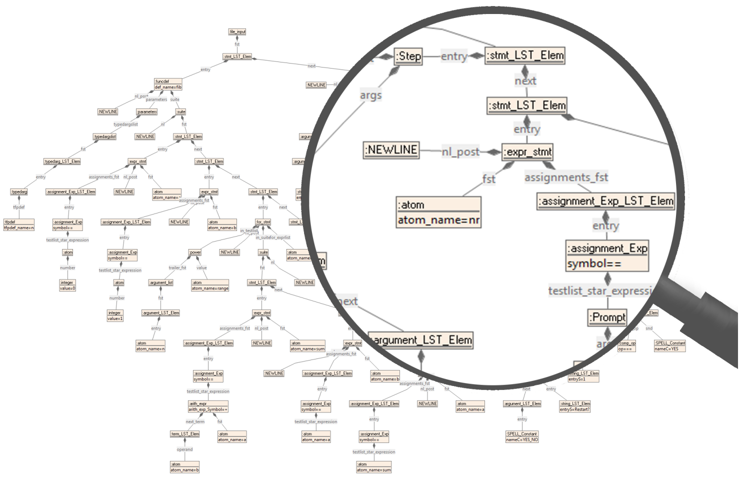


Fig. 4. Abstract syntax graph of the running example

the visualisation currently exists as a prototype and will be integrated into both SPELL environments.

Example 2. In Fig. 4 the abstract syntax graph (ASG) of the running example is illustrated. Due to the complexity, we highlight a detail of the ASG which represents parts of source code lines 13 and 14 (step and assignment statements, see Fig. 1). The ASG is a graph typed over the source part of the type graph. The types are indicated by "':'", e.g., `stmt_LST_elem` is the type of the second node from top.

4 Formal Framework and Application

In the following section, we briefly introduce the main concepts for model transformations based on TGGs [5] on the basis of the running example and the synchronisation framework [15] that we use in the project.

A triple graph is an integrated model, i.e., a model which is composed of a source model, a target model and correspondences between these models. It consists of three graphs: the source, correspondence, and target graphs, and two graph morphisms (mappings) specifying the correspondences between elements of the source and target model. In Fig. 7 an excerpt of the triple graph for our running example is given. A triple graph morphism defines mappings between triple graphs which preserve the correspondences.

Triple graphs are typed over a type triple graph *TG* via a triple graph morphism. Triple graph morphisms between triple graphs have to preserve the typing. *TG* can be seen as the meta-model. Triple graphs can have attributes and node type inheritance. For this, we use the formal notation presented in [5,6].

$$\begin{tikzcd}
L \arrow[r, hook, "tr"] \arrow[d, "m"'] \arrow[dr, phantom, "(PO)"] & R \arrow[d, "n"] \\
G \arrow[r, hook, "t"'] & H
\end{tikzcd}$$

Fig. 5. Tripe rule

A triple rule *tr* as shown on the first row of Fig. 5 is an inclusion of triple graphs $L \hookrightarrow R$, i.e., all elements in L are uniquely mapped

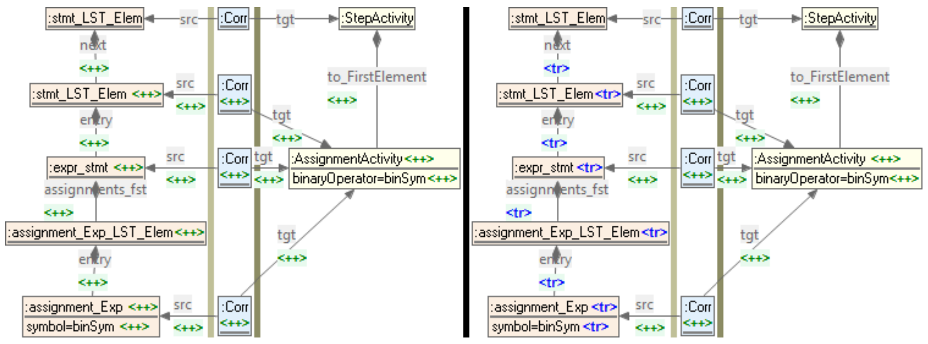


Fig. 6. Triple rule *T_Step_assignment_Expr-2-AssignmentActivity* (left) and derived FT rule (right)

to elements in R. Consequently, triple rules are non-deleting. They specify, how a consistent triple graph can be extended on all three parts simultaneously resulting again in a consistent triple graph. The rule application is illustrated in Fig. 5. The triple rule tr is applied to triple graph G via a graph morphism m called match. The result is triple graph H, where L is replaced by R in G [6]. Triple rules can be extended by negative application conditions (NACs) defining forbidden context in order to restrict the rule application [5].

Example 3 (Triple Rule). In Fig. 6, a triple rule is illustrated. Elements marked with <++> are created by this triple rule. Unmarked elements are called context elements. This triple rule creates correspondences between an assignment expression following a `Step` command in the SPELL ASG with an assignment activity in the SPELLFlow ASG. The latter element is situated on a new layer in the target graph. The new layer is indicated by containment edges, i.e., the assignment activity is contained by the step activity.

A TGG is a tuple $TGG = (TG, S, TR)$ containing a type triple graph TG, a start graph S, which is usually the empty triple graph, and a set of triple rules TR. A TGG generates all consistent triple graphs. For $TG = (TG_S \leftarrow TG_C \rightarrow TG_T)$, we use $\mathcal{L}_{TG}$, $\mathcal{L}_S$, $\mathcal{L}_T$ to denote the language, i.e., the classes of all graphs typed over TG, TG_S, or TG_T, respectively.

For the translation from the source into the target model, we use a set of operational forward translation rules (FT rules) that are generated automatically out of the set of triple rules [14]. Each FT rule tr_{FT} differs only on the source part from the corresponding triple rule tr: Each <++> is replaced by a Boolean valued marker `<tr>`. In order to translate a source model to an integrated model, all elements in the source model are initially marked with false. When applying an FT rule, the `<tr>`-marker is set to true, so that the specific rule cannot be applied again on the same elements. So, the source model will be translated stepwise into an integrated model, without modifying the source model. Similar to the generation of FT rules, operational backward translation rules (BT rules) can be created in order to translate backward a target model into the integrated model.

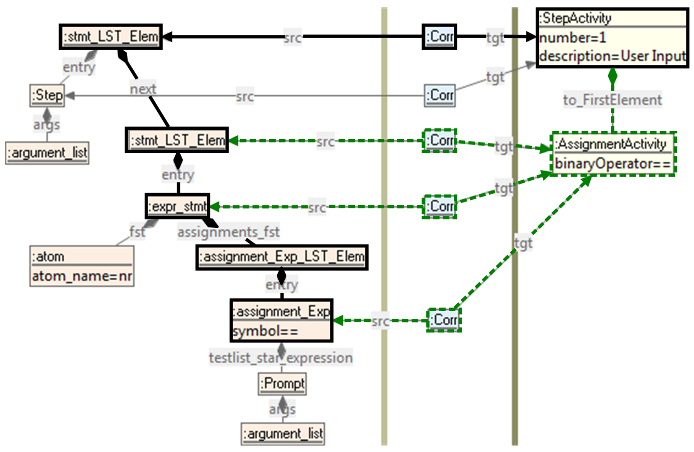


Fig. 7. Excerpt from triple graph (integrated model)

Example 4 (FT/BT Rule and FT-Rule Application). We consider the triple rule in Fig. 6. In the corresponding FT rule, each `<++>` marker is replaced by a `<tr>` in the source part. In the corresponding BT rule, the marker is replaced in the target part.

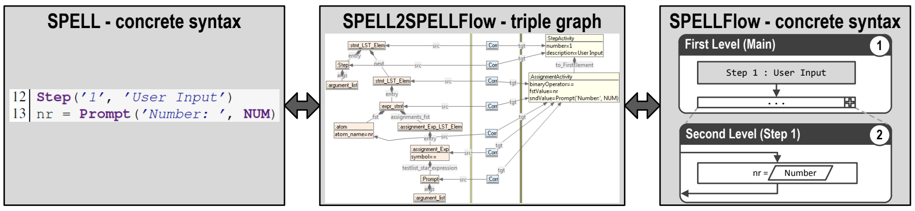
In general, the translation from a source model into a target model needs the source graph as a basis. In our example, the translation is performed from the SPELL ASG, which is illustrated in Fig. 4. It will be translated stepwise into an integrated model with the set of FT rules. In Fig. 7 we illustrate the application of the FT rule *FT_Step-assignment_Expr-2-AssignmentActivity* generated out of the triple rule in Fig. 6. The Step statement is already translated, so that the rule *FT_Step-assignment_Expr-2-AssignmentActivity* can be applied. The elements marked with a fat border are required context elements which are mapped by the FT rule. After applying the FT rule, the elements marked with a dashed line are created by this FT rule.

In future work, we will apply the model synchronisation framework based on TGGs [15]. The main idea is to propagate changes from one domain to the other by reusing the operational forward and backward translation rules.

$$\begin{tikzcd}[row sep=small] \multicolumn{2}{c}{\forall\, G'^S \in \mathcal{L}_S :} & \\ G^S \arrow[r, leftrightarrow, "r"] \arrow[d, "a"'] \arrow[dr, phantom, "\searrow{:}\mathsf{fPpg}"] & G^T \arrow[d, dotted, "b"] \\ G'^S \arrow[r, leftrightarrow, dotted, "r'"'] & G'^T \end{tikzcd} \qquad \begin{tikzcd}[row sep=small] \multicolumn{2}{c}{\forall\, G'^T \in \mathcal{L}_T :} & \\ G^S \arrow[r, leftrightarrow, "r"] \arrow[d, "a"'] \arrow[dr, phantom, "\swarrow{:}\mathsf{bPpg}"] & G^T \arrow[d, dotted, "b"] \\ G'^S \arrow[r, leftrightarrow, dotted, "r'"'] & G'^T \end{tikzcd}$$

Fig. 8. Synchronisation operations fPpg, bPpg

In Fig. 8, we illustrate the forward propagation operation (fPpg) which applies the model update *a* performed in the source model to the integrated model. On the left side of the figure, we illustrate the fPpg operation and on the right side, we illustrate the symmetric backward propagation operation bPpg. The forward operation consists of three steps: The *forward alignment* step constructs a new correspondence graph by deleting all correspondence elements which became invalid by the source model update *a*. The *deletion* operation creates a consistent integrated model in removing parts which became inconsistent by update *a*. The *forward addition* operation executes the operational forward rules, until all untranslated elements are translated. Due to the definition



```
Step('1', 'User Input')
nr = Prompt('Number: ', NUM)
```

Fig. 9. Excerpt: Summary of bidirectional engineering process

of this operation, the resulting integrated model is consistent. The bPpg operation is symmetric. In [15], we have shown for this model synchronisation framework that also correctness and completeness properties hold.

However, translating all updates performed synchronously in the source and target model to the integrated model can cause conflicts. In [7], an appropriate conflict resolution is discussed.

Due to the well-defined formal frameworks we use for defining the translation and later synchronisation, requirement 3 (concise and correct visual models and functional behaviour) is fulfilled. The model is concise, because we defined a hierarchical (multi-layer) view on the SPELL source code, especially the main layer of SPELLFlow provides an abstract view on the SPELL source code. The correctness and completeness w.r.t. correspondence patterns between SPELL and SPELLFlow is ensured by Theorem 8.2 in [15]. To show functional behaviour, we use the automatic critical pair analysis provided by HenshinTGG [13,14].

In Fig. 9, we show an overview of the whole bidirectional engineering process for an excerpt of our running example. The SPELL source code is parsed using Xtext yielding the SPELL ASG (left). This ASG is translated into an integrated model (middle) represented by a triple graph in using the set of FT rules. The target part is the SPELLFlow ASG, which is exported as an XMI file. This XMI file is imported into the SPELLFlowEditor which displays the desired SPELLFlow digram (right) in concrete syntax. To generate source code out of the visualisation, we will perform the same process in the backward direction and apply the set of BT rules for the translation. At a later stage, we will apply the presented synchronisation framework.

5 Related Work

TGGs were introduced in [21] and since then refined and extended by several works [19,12,22]. Many works focus on defining and preserving correctness properties and functional behaviour of TGGs [5,14]. Based on the delta-lenses framework [24], TGGs were extended by bidirectional model synchronisation frameworks [10,15]. These results will be reused in the presented approach.

In [17,1], a new type of TGGs was introduced: view triple graph grammars (VTGGs), in order to model domain-specific views of a source model. The authors present different views, e.g., domain-specific views or views presenting different abstraction layers, and describe an appropriate model transformation technique satisfying every type of view. VTGGs are very promising for the approach presented in this paper, though the definition of VTGGs as given in [1] is too restrictive for our approach and needs to be relaxed.

The Atlas Transformation Language (ATL) [18] is a widely-used framework for specifying model transformations in a declarative manner. However, the approach only supports the specification of unidirectional transformations and requires to specify each direction of bidirectional model transformations separately. Therefore, in contrast to TGGs, the approach does not allow to generate operational translation rules for forward and backward model transformations from one consistent specification.

Several works deal with model visualisation and visualisation languages. In [9], a general approach for defining a visualisation language and its simulation is given based on typed algebraic graph transformation. In [11], an overview of different software visualisation approaches and important properties for an appropriate visualisation are discussed. SPELLFlow matches most of these requirements. Koschke [20] presents a tool suite for software visualisation in reverse engineering. There, different visualisations are provided as additional information. In contrast, it is planned that SPELLFlow will replace the source code view completely. Both papers present surveys on software visualisation where the majority of interviewees (more than 80% in each survey) agree that software visualisation is at least important.

6 Conclusion

In this paper we introduced a new visual modelling language (*SPELLFlow*) for the visualisation of procedures written in the satellite control language *SPELL*. The requirements for the syntax and semantics of the visual language SPELLFlow were developed in cooperation with the industrial partner SES. We presented an approach for the automatic generation of SPELLFlow models from SPELL programs, and the generation of SPELL source code from SPELLFlow models. This bidirectional engineering approach is based on the formal framework of TGGs and supported by the tool HenshinTGG and a visual editor based on Eclipse GMF which we developed for SPELLFlow.

According to the requirements set up by SES, we will apply the synchronisation framework presented in [15] using HenshinTGG. Finally, we will evaluate our implementations regarding efficiency and usability in order to integrate the implementations in the daily work of satellite controllers and developers at SES.

Acknowledgements. Supported by the Fonds National de la Recherche, Luxembourg (3968135).

Fonds National de la Recherche Luxembourg

References

1. Anjorin, A., Rose, S., Deckwerth, F., Schürr, A.: Asymmetric delta lenses with view triple graph grammars (to appear). ECEASST pp. 1–15 (2013)
2. Eclipse Consortium: Eclipse Graphical Modeling Framework (GMF) (2013), `http://www.eclipse.org/modeling/gmp/`

3. Eclipse Consortium: Eclipse Modeling Framework (EMF), Version 2.8.3 (2013), `http://www.eclipse.org/emf`
4. The Eclipse Foundation: Xtext, Version 2.3.1 (2013), `http://www.eclipse.org/Xtext/`
5. Ehrig, H., Ermel, C., Hermann, F., Prange, U.: On-the-Fly Construction, Correctness and Completeness of Model Transformations based on Triple Graph Grammars. In: Proc. MODELS'09. LNCS, vol. 5795, pp. 241–255. Springer (2009)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science, Springer (2006)
7. Ehrig, H., Ermel, C., Taentzer, G.: A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications. In: Proc. FASE'11. LNCS, vol. 6603, pp. 202–216. Springer (2011)
8. Ermel, C., Hermann, F., Gall, J., Binanzer, D.: Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars. ECEASST 54, 1–14 (2012)
9. Ermel, C.: Simulation and animation of visual languages based on typed algebraic graph transformation. Ph.D. thesis, Technische Universität Berlin (2006)
10. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. SoSyM 8, 21–43 (2009)
11. Gracanin, D., Matkovic, K., Eltoweissy, M.: Software visualization. ISSE 1(2), 221–230 (2005)
12. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. SoSyM 9, 21–46 (2010)
13. EMF Henshin, Version 0.9.6 (2013), `http://www.eclipse.org/henshin/`
14. Hermann, F., Ehrig, H., Golas, U., Orejas, F.: Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In: Proc. MDI'10. pp. 22–31. MDI '10, ACM (2010)
15. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., Engel, T.: Model synchronization based on triple graph grammars: correctness, completeness and invertibility. SoSyM pp. 1–29 (2013)
16. Hermann, F., Gottmann, S., Nachtigall, N., Braatz, B., Morelli, G., Pierre, A., Engel, T.: On an Automated Translation of Satellite Procedures Using Triple Graph Grammars. In: Proc. ICMT'13, LNCS, vol. 7909, pp. 50–51. Springer (2013)
17. Jakob, J., Königs, A., Schürr, A.: Non-materialized Model View Specification with Triple Graph Grammars. In: Graph Transformations, LNCS, vol. 4178, pp. 321–335. Springer (2006)
18. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72, 31–39 (2008)
19. Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Tech. Rep. TR-ri-07-284, Department of Computer Science, University of Paderborn, Germany (2007)
20. Koschke, R.: Software Visualization for Reverse Engineering. In: Revised Lectures on Software Visualization, International Seminar. pp. 138–150. Springer (2002)
21. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Tinhofer, G. (ed.) Proc. WG'94. LNCS, vol. 903, pp. 151–163. Springer (1994)
22. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Proc. ICGT'08. pp. 411–425. No. 5214 in LNCS, Springer (2008)
23. SES Engineering: SPELL - Satellite Procedure Execution Language & Library, Version 2.3.13 (2013), `http://code.google.com/p/spell-sat/`
24. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting Parallel Updates with Bidirectional Model Transformations. In: Proc. ICMT'09. pp. 213–228. Springer (2009)