GEMOC 2014 2nd International Workshop on The Globalization of Modeling Languages

Benoit Combemale, Julien Deantoni, Robert France

To cite this version:

Benoit Combemale, Julien Deantoni, Robert France. GEMOC 2014 2nd International Workshop on The Globalization of Modeling Languages. Benoit Combemale; Julien Deantoni; Robert France. GEMOC 2014 co-located with MODELS 2014, Sep 2014, Valencia, France. 1236, , pp.82, 2014, CEUR-WS, 1613-0073. hal-01074602

HAL Id: hal-01074602
https://inria.hal.science/hal-01074602v1

Submitted on 14 Oct 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

GEMOC 2014

2nd International Workshop on *The Globalization of Modeling Languages*

September 28, 2014, Valencia, Spain
co-located with MODELS 2014

Benoit Combemale
Julien Deantoni
Robert France

These proceedings are partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011)

Preface

This volume contains the papers presented at GEMOC 2014, the 2nd International Workshop on The Globalization of Modeling Languages held on September 27-28, 2014 in Valencia.

September 10, 2014
Sophia Antipolis Cedex

Benoit Combemale
Julien Deantoni
Robert B. France

Table of Contents

Workshop on the Globalization of Modeling Languages 1
Benoit Combemale, Julien Deantoni and Robert France

Unification or integration? The Challenge of Semantics in Heterogeneous Modeling Languages 2
Gabor Karsai

Supporting Diverse Notations in MPS' Projectional Editor 7
Markus Voelter and Sascha Lisson

Putting the Pieces Together - Technical, Organisational and Social Aspects of Language Integration for Complex Systems 17
Håkan Burden

Towards Integrating Modeling and Programming Languages: The Case of UML and Java 23
Patrick Neubauer, Tanja Mayerhofer and Gerti Kappel

A Declarative Approach to Heterogeneous Multi-Mode Modelling Languages 33
Tony Clark

Extensible Global Model Management with Meta-model Subsets and Model Synchronization 43
Dominique Blouin, Yvan Eustache and Jean-Philippe Diguet

Towards an Ontology-based Approach for Heterogeneous Model Matching 53
Mahmoud El Hamlaoui, Cassia Trojahn, Sophie EBERSOLD and Bernard COULETTE

Supporting Debugging in a Heterogeneous, Globally Distributed Environment 63
Jonathan Corley and Jeff Gray

Understanding Metalanguage Integration by Renarrating a Technical Space Megamodel 69
Vadim Zaytsev

Program Committee

Marsha Chechik	University of Toronto
Tony Clark	Middlesex University
Benoit Combemale	IRISA, Université de Rennes 1
Julien Deantoni	I3S, University of Nice Sophia Antipolis
Robert France	University of Colorado
Jeff Gray	University of Alabama
Jean-Marc Jézéquel	University of Rennes 1
Ralf Laemmel	Universität Koblenz-Landau
Marjan Mernik	LPM, University of Maribor
Gunter Mussbacher	McGill University
Richard Paige	University of York
Bernhard Rumpe	RWTH Aachen University
Juha-Pekka Tolvanen	MetaCase
Mark Van Den Brand	Eindhoven University of Technology
Eric Van Wyk	University of Minnesota
Markus Voelter	Independent

Author Index

B

Blouin, Dominique 37
Burden, Håkan 16

C

Clark, Tony 27
Corley, Jonathan 57
COULETTE, Bernard 47

D

Diguet, Jean-Philippe 37

E

EBERSOLD, Sophie 47
El Hamlaoui, Mahmoud 47
Eustache, Yvan 37

G

Gray, Jeff 57

K

Kappel, Gerti 17
Karsai, Gabor 1

L

Lisson, Sascha 6

M

Mayerhofer, Tanja 17

N

Neubauer, Patrick 17

T

Trojahn, Cassia 47

V

Voelter, Markus 6

Z

Zaytsev, Vadim 63

Workshop on the Globalization of Modeling Languages

Benoit Combemale[1], Julien Deantoni[2], and Robert France[3]

[1] IRISA, University of Rennes1, France, benoit.combemale@irisa.fr
[2] I3S, University of Nice Sophia Antipolis, France, julien.deantoni@polytech.unice.fr
[3] Colorado State University, USA, france@cs.colostate.edu

This volume contains the papers presented at GEMOC 2014, the 2nd International Workshop on The Globalization of Modeling Languages held on September 27-28, 2014 in Valencia.

Context and Motivation

Software intensive systems are becoming more and more complex and communicative. Consequently, the development of such systems requires the integration of many different concerns and skills. These concerns are usually covered by different languages, with specific concepts, technologies and abstraction levels. This multiplication of languages eases the development related to one specific concern but raises language and technology integration problems at the different stages of the software life cycle. In order to reason about the global system, it becomes necessary to explicitly describe the different kinds of relationships that exist between the different languages used in the development of a complex system. To support effective language integration, there is a pressing need to reify and classify these relationships, as well as the language interactions that the relationships enable. In this context, the proceedings of the workshop GEMOC 2014 include contributions that outline language integration approaches, case studies, or that identify and discuss well defined problems about the management of relationships between heterogeneous modeling languages.

This edition 2014 of the GEMOC workshop followed the successful first edition at MODELS 2013 in Miami, FL, USA. This new edition completes the state of the art and practice started last year. It also strengthen the community that broadens the current DSML research focus beyond the development of independent DSMLs to one that provides support for globalized DSMLs.

GEMOC 2014 is supported by the GEMOC initiative that promotes research seeking to develop the necessary breakthroughs in software languages to support global software engineering, i.e., breakthroughs that lead to effective technologies supporting different forms of language integration, including language collaboration, interoperability and composability.

Content

This workshop proceedings include an extended abstract of the keynote presentation given by Prof. Gabor Karsai, and 8 technical papers.

Unification or integration? The Challenge of Semantics in Heterogeneous Modeling Languages

Gabor Karsai

Institute for Software-Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37235, USA
gabor.karsai@vanderbilt.edu

Abstract. Model-driven software development and systems engineering rely on modeling languages that provide efficient, domain-specific abstractions for design, analysis, and implementation. Models are essential for communicating ideas across the engineering team, but also key to the analysis of the system. No single model or modeling language can cover all aspects of a system, and even for particular aspects multiple modeling languages are used in the same system. Thus engineers face the dilemma of either defining a unifying semantics for all models, or finding a solution to the model integration problem. The talk will elaborate these problems, and show two, potential solutions: one using a model integration language (for the engineering design domain) and another one using explicit and executable semantics (for the domain of distributed reactive controllers).

The problem

Engineered systems are increasingly built using model-driven techniques, where models are used in all phases of the system's lifecycle, from concept development to product to operation. Models are built for everything: from the smallest part to the entire system, and models are used for all sorts engineering activities: from design to analysis and verification, to implementation and manufacturing. Engineering models are often based on domain-specific abstractions of reality; for example a finite-element model represents a 3D shape and an engineering assembly drawing represents how those shapes need to be joined together by some manufacturing steps to form an assembly. While there is a multitude of domain-specific models used in the design of a complex system, somehow these models have to 'fit together' because (1) they are describing the same, single system, and (2) they need to be combined to allow cross-domain, system-level analysis of the design. Models created in (domain-specific) isolation are necessary and very useful, but insufficient when the larger system is considered - in the larger systems subsystems and their components interact, and these interactions have to be expressed as well.

Obviously, the same applies to software systems: multiple, often domain-specific models are used to describe a complex system. Somewhat differently from conventional engineering, where a multiple, (physical) domain modeling tools are used, in software we tend to use multiple domain-specific modeling languages. Arguably, every modeling tool has a 'language' (explicitly defined or not) and a modeling language without a supporting tool is only partially useful; hence in this paper we will focus on the issue of the modeling languages and their semantics. Semantics is a central question in language engineering: how do we specify what 'sentences' of an artificial, engineered language mean? Fortunately, in the theory of computer languages there has been many decades of research that produced techniques for specifying semantics of languages. However, these specifications rarely span multiple, potentially different languages.

The problem at hand is stated as follows: How can we integrate heterogeneous, domain-specific modeling languages so that the instance models can be linked together and system-level analysis can be performed on these models? It is easy to see that this problem has multiple facets. Naturally, one problem is that of semantics: how do we 'integrate the semantics' of multiple modeling languages? Say, if we have a model $M_{L_1}(A)$ of a subsystem A in a modeling language L_1, and another model $M_{L_2}(A)$ of the same A in a modeling language L_2, what does the composition $M_{L_1}(A) \times M_{L_2}(A)$ mean (if it is meaningful at all)? Similarly, what if we compose the models of two different components, say A and B, and ask the same question about $M_{L_1}(A) \times M_{L_2}(B)$? Clearly, the semantics of composition has to be defined. Another problem is more operational: how do we manage complex 'model repositories' where the models of the system (or systems) are being kept? If changes are made in one model, what is the impact of these changes on the dependent and related models? Building and managing such model repositories brings up many deep technical and pragmatic problems.

The problem stated above has a close relationship to systems engineering. One of the main tasks in systems engineering is to discover, understand, and manage cross-domain and cross-system interactions that make the engineering of complex so difficult. Solutions, like SysML are certainly a good step in the right direction, but they are rather limited as far as semantics is concerned, and more research is needed to place them on a solid theoretical foundation.

Unification or Integration?

There seem to be (at least) two approaches to solving the problem. One can be called as 'unification', where we design a universal modeling language that magically unifies all existing modeling languages. Domain-specific models will then be translated into this unified modeling language, and analysis and verification will happen on the unified models. The semantics of all domain-specific modeling languages would have to be re-expressed in the unified language (i.e. in a common semantic domain). However, this approach seems very unrealistic: the domain-specific languages are typically rich, so coming up with a language that unifies them all is extremely difficult, their semantics sometimes does not align

well, and creating a grand unified language does not seem feasible. Additionally, the set of languages to be integrated is changing from project to project, so a unified language will have to be extremely large. Arguably, the only 'language' that is common across domain-specific modeling is that of mathematics but this on such a high level that it is not pragmatic due to the loss of domain-specificity.

The other approach can be called as 'integration' where the focus is on integrating models: a model integration language (MIL) with 'sparse' semantics is used. The semantics of the model integration language is for capturing the cross-domain interactions in terms of the structure of the system. This model integration language is lightweight and (potentially) evolvable, so that new domain-specific modeling languages can be added to the suite as necessitated by the development project. In this approach domain-specific models 'stay' in their own modeling tools, and the integration models are reflections of these domain models. The integration models thus capture the interfaces of the domain models relevant for analyzing the interactions.The interfaces of the component (or subsystem) models in the MIL are 'rich' in the sense that they are multi-domain and their connectivity will allow the analysis of interactions throughout the system.

An example for a Model Integration Language

In one of our research projects[1], we have built a model-integration language to support the design of complex cyber-physical systems (CPS). CPS are defined as engineered systems that integrate physical and cyber components where relevant functions are realized through the interactions between the physical and cyber parts. Examples include highly automated vehicles, smart energy distribution systems, automated manufacturing systems, intelligent medical devices, etc. The design of such systems involves the design of the physical, the cyber (computational and communicational), and the cyber-physical components of the system and their integration. There are a number of complex engineering tools that solve parts of the problem, e.g. CAD tools for mechanical design, Finite Element Analysis (FEA) tools for determining stresses on structural elements, simulation tools for the analyzing the dynamics of the system, modeling and synthesis tools for the design and implementation of the (cyber) hardware and software, thermal analysis tools for verifying thermal behavior of the system, and so on; but they are used in isolation, by domain engineers. Purely understood cross-domain interactions lead to expensive design iterations.

We have designed and implemented a MIL called 'Cyber-Physical Modeling Language' (CyPhyML) [1] that allows the representation of cyber-physical components and the design CPS through composition. The language is primarily structural (i.e. composition-oriented), but components (and subsystems) have rich, typed interfaces, with four categories: parametric and property interfaces (for parametrization and configuration), signal interfaces (for cyber interactions),

[1] DARPA Adaptive Vehicle Make Program, META Design Tools and Languages project at Vanderbilt University

power interfaces (for physical interactions representing the dynamics), and structural interfaces (for geometric alignment of the physical components). In CyPhyML one can represent the design of an entire CPS, but the native models of the components and subsystem are stored in the domain-specific tools - CyPhyML merely describes how they are composed. Cross-domain analysis is supported by model interpreters that assemble complex model analysis campaigns from the CyPhyML models, possibly involving multiple analysis tools.

An example for Interaction Modeling

In another research project[2] we worked on the problem of semantic integration of models representing reactive controllers. The motivating example came from a system of systems: a spacecraft and a launch vehicle, where both systems have a reactive controller that interacts with its counterpart in the other system. Two major issues were posed: (1) each reactive controller was modeled in a different variant of the Statechart notation (Stateflow and UML State machines, specifically), and (2) the controllers were exchanging messages that influenced their behavior. The goal was to verify the concrete, integrated system (where the controller models and the message exchanges were given) through model checking.

The first problem was solved by developing a framework, called Polyglot[2], to specify the semantics of Statechart variants in an executable form. The framework is built as a set of Java classes that can be specialized according to the semantic variant yielding an 'interpreter' that receives events and produces reactions to events, but whose behavior is determined by the specific model, acting as the 'program' being interpreted. The 'model' is stored as a data structure in the interpreter. We have verified the correctness of the model interpreter(s) using numerous tests exercising the various features of the modeling language. The model interpreter produced the same output sequences from the same input sequences as the code generated by the Stateflow and Rational Rose code generators, respectively. Note that the interpreter (as well as the generated code) is purely sequential: it is executed upon the arrival of input events and produces output events upon each invocation.

The second problem was addressed by providing a framework, implemented again in Java, for representing (1) how a reactive controller is wrapped into a looping process that uses some (blocking or non-blocking) 'receive' and 'send' operations to interact with its environment, and (2) how two (or more) reactive processes interact with each other via some message exchange protocol. Note that the processes are concurrent, i.e. arbitrary interleaving of the process executions is possible, hence the system has a built-in non-determinism. The main idea here was to model the interactions (thus the integration) via modeling the 'glue': the scheduling of processes and the interaction protocols. In other words, we have created a (potentially non-deterministic) scheduler that modeled the behavior of

[2] Model Transformations and Verification project, supported by NASA ARC.

the composed system. For the analysis of the composed system we have relied on the Java Path Finder tool (from NASA) that allows the byte-code based verification of Java programs, permitting non-deterministic behavior.

Lessons Learned

The main lesson we have learned was that one should focus on problem-driven integration of models, and not on some grand unification. Models are built for a purpose and when a larger system needs to be analyzed, synthesized, implemented, verified, tested, operated, or maintained one has to be very pragmatic and concentrate on what the models are for, and consider integration accordingly. Hence, one should pay attention to how effectively such model integration can be supported by tools.

Acknowledgments

The concepts, ideas, and work described in this paper are the work of many people, including Janos Sztipanovits, Daniel Balasubramanian, Ted Bapty, Abhishek Dubey, Ethan Jackson, Xenofon Koutsoukos, Zsolt Lattmann, Tihamer Levendovszky, Nag Mahadevan, Adam Nagel, Sandeep Neema, Joseph Porter, Gabor Simko and many of the colleagues at the Institute for Software-Integrated Systems. The support of Dr Michael Lowry of NASA Ames and Dr Corina Pasarenau of CMU with whom we collaborated on the NASA project is also recognized. The specific work has been supported by the DARPA AVM program (HR0011-12-C-0008), by the NSF (CNS-1035655) and NASA (NNX09AV58A). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not reflect the views of DARPA, NSF, or NASA.

References

1. Sztipanovits, J., Bapty, T., Neema, S., Howard, L., Jackson, E.: Openmeta: A model and component-based design tool chain for cyber-physical systems. In: From Programs to Systems – The Systems Perspective in Computing (FPS 2014), Grenoble, France, Springer, Springer (2014)
2. Balasubramanian, D., Păsăreanu, C.S., Karsai, G., Lowry, M.R.: Polyglot: systematic analysis for multiple statechart formalisms. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2013) 523–529

Supporting Diverse Notations in MPS' Projectional Editor

Markus Voelter[1] and Sascha Lisson[2]

[1] independent/itemis, `voelter@acm.org`
[2] itemis AG, `lisson@itemis.de`

Abstract. To be able to build effective DSLs, these DSLs must not just use language concepts that are aligned with their respective domain, but also use notations that correspond closely to established domain notations – and those are often not purely textual or graphical. The underlying language workbench must support these notations, and combining different notations in a single editor must be supported as well in order to support the coherent definitions of systems that use several DSLs. In this paper we provide an overview over the notations supported by JetBrains MPS. MPS is a language workbench that uses a projectional editor, which, by its very nature, can deal with many different notational styles, including text, prose, math tables and graphics. The various supported notations are illustrated with examples from real-world systems.

1 Introduction

The GEMOC 2014 workshop description states: *To cope with complexity, modern software-intensive systems are often split in different concerns, which serve diverse stakeholder groups and thus must address a variety of stakeholder concerns. These different concerns are often associated with specialized description languages and technologies, which are based on concern-specific problems and solution concepts.* In particular, these different concerns also require different notations. Ideally, these notations are closely aligned with existing domain-specific notations used by the stakeholders. However, such existing notations are not necessarily just text: they use forms, diagrams, mathematical symbols, a mix of prose and structure or combinations of those. Representing such diverse notations faithfully requires a high degree of flexibility in the kinds of editors that can be built with the language workbench used to create the languages.

Projectional editing (see Section 2) allows creating editors that can use a wide variety of notations, including the ones mentioned above. In particular, it can also mix these notations seamlessly, leading to a more faithful representation of existing domain languages in tools. JetBrains MPS is one of the leading projectional editors, and this paper describes its capabilities in terms of notational flexibility.

Contribution This paper provides an overview over the notational styles currently supported by MPS. For each style we discuss why it is useful, where it is being used as well as some details about how to define the respective editors.

Availability of the Code JetBrains MPS is open source software available from `http://jetbrains.com/mps`. Also, those editor facilities that are separate plugins to MPS are open source software and their repositories are indicated in each case. The examples shown in this paper are mostly based on mbeddr [1] and are open source as well. The screenshots in Figures 7, 9 and 10 are taken from a commercial tool currently being developed by Siemens PLM software; however, the underlying editor facilities are all open source as well.

Structure In the next section we provide a brief overview over MPS' projectional editor and show briefly how to implement regular text editors. Section 3 introduces the fundamental notations supported by MPS and Section 4 discusses other useful features of the MPS editor. We conclude the paper with a brief discussion and summary in Section 5.

2 Projectional Editing in MPS

What is Projectional Editing? In parser-based editors users type sequences of characters into a text buffer. The buffer is parsed to check whether the sequence of characters conforms to a grammar. The parser ultimately builds an abstract syntax tree (AST), which contains the relevant structure of the program, but omits syntactic details. Subsequent processing (linking, type checks, transformation) is based on the AST. Modern IDEs (re-)parse the concrete syntax while the user edits the code, continuously maintaining an up-to-date AST in the background that reflects the code in the editor's text buffer. However, even in this case, this AST is created by a parser-driven transformation from the source text.

A projectional editor does not rely on parsers. As a user edits a program, the AST is modified *directly*. Projection rules are used to create a representation of the AST with which the user interacts, and which reflects the resulting changes. No parser-based transformation from concrete to abstract syntax is involved. This approach is well-known from graphical editors: when editing a UML diagram, users do not draw pixels onto a canvas, and a "pixel parser" then creates the AST. Rather, the editor creates an instance of `uml.Class` when a user drops a class. A projection engine renders the class as a rectangle. As the user edits the program, program nodes are created as instances of language concepts. Programs are stored using a generic tree persistence format (such as XML).

The projectional approach can be generalized to work with any notation, including textual. A code-completion menu lets users create instances based on a text string entered in the editor called the *alias*. The aliases allowed in any given location depend on the language definition. Importantly, *every next text string is recognized as it is entered*, so there is never any parsing of a sequence of text strings. In contrast to parser-based editors, where disambiguation is performed by the parser after a (potentially) complete program has been entered, in projectional editors disambiguation is performed by the user as he selects a concept from the code-completion menu. Once a node is created, it is *never* ambiguous, *irrespective of its syntax*: every node points to its defining concept. Every program node has a unique ID, and references between program elements are represented as references to the ID. These references are established during

```
[- if ( % condition % ) % thenPart % ?(- % elseIfs % /empty cell: <default> -)
    ?^[- ^ else % elsePart % -] -]
```

Fig. 1. Editor definition for the `IfStatement` (details in the running text).

program editing by directly selecting reference targets from the code-completion menu; the references are persistent. This is in contrast to parser-based editors, where a reference is expressed as a string in the source text, and a separate name resolution phase resolves the target AST element after the text has been parsed.

Projectional editing has two advantages. First, it supports flexible composition of languages because the ambiguities associated with parsers cannot happen in projectional editors. We do not discuss this aspect in this paper and refer the reader to [2]. The other advantage of projectional editors is that, since no parsing is used, the program notation does not have to be parseable and a wide range of notations can be used. This paper focusses on this aspect. Traditionally, projectional editors have also had disadvantages relative to editor usability and infrastructure integration; those are discussed in [3].

Defining a Simple Editor In order for the reader to better understand the explanations in Sections 3 and 4, this section briefly introduces the MPS structure and editor definitions. MPS' meta model is similar to EMF Ecore [4]. Language concepts (aka meta classes) declare children (single or lists), references and primitive properties. Concepts can extend other concepts or implement concept interfaces; subtype polymorphism is supported. Programs are represented as instances of concepts, called nodes. Each concept also defines one or more editors. These are the projection rules that determine the notation of instance nodes in the program. Editor definitions consist of cells arranged in various layouts. A cell can be seen as an atomic element of an editor definition. As an example, let us consider the `if` statement in C. Its structure is defined as follows:

```
concept IfStatement extends Statement
  alias: if
  children:
    condition: Expression[1]         elsePart:  StatementList[0..1]
    thenPart:  StatementList[1]      elseIfs:   ElseIfPart[0..n]
```

Fig. 1 shows the editor definition for the `IfStatement` concept. At the top level, it consits of a collection cell `[- .. -]` which aligns a sequence of additional cells in some particular way – a linear sequence in this case. The sequence starts with the constant (keyword) `if` and a pair or parentheses. Between those, the editor projects the `condition` expression; the `%` sign is used to refer to children of the current concept. The `thenPart` follows, and since it is a `StatementList`, it comes with its own curly braces. The `(- ... -)` collection captures the list of `else if` parts, if any. The `ElseIfPart` comes with its own editor which is embedded here. Finally, there is an optional set of cells (represented by the `?` and a condition expression that is not shown) that contains the `else` keyword as well as the `elsePart` child. A flag (not shown) determines that the `else` part is shown on a new line, leading to the expected representation of `if` statements.

```
double midnight2(int32 a, int32 b, int32 c) {
```

$$\text{return } \frac{-b + \sqrt{b^2 - \sum_{i = 1}^{4} a * c}}{2 * a};$$

```
} midnight2 (function)
```

Fig. 2. Mathematical symbols used in C expressions embedded into C functions.

LOOP	
lower:	[> { name } = % lower % <]
upper:	% upper %
body:	% body %
symbol:	SumSymbolSerif

Fig. 3. The definition of the sum symbol editor using the `LOOP` primitive.

3 Notations

This section discusses the notations supported by MPS. For each we provide a rationale, an example and a hint on how to build editors that use the style.

Textual Notations The first notation supported by MPS has been textual notations. Notations used by programming languages such as Java, C or HTML can be represented easily. The example in the previous section shows how to create editors for textual notations. The backbone is the `indent layout` collection cell which can deal flexibly with sequences of nodes, newlines and indentation.

Mathematical Symbols A plugin [5] supports mathematical notations. The plugin comes with a set of new layout primitives (cell types) that enable typical mathematical notations such as fraction bars, big symbols (sum or product), roots and all kinds of side decorations (as used in `abs` or `floor`). The plugin contributes only the editor cells so they can be integrated into arbitrary languages. So far they have been integrated into C (Fig. 2) and an insurance DSL.

Fig. 3 shows the definition of the sum editor. It uses the new primitive `LOOP` which can be used for everything that has a big symbol as well as things above, below and right of the symbol. The particular symbol is defined separately and referenced from the editor definition. The `LOOP` cell has three slots (`lower`, `upper` and `body`) into which child nodes can be added. The `Sum` expression defines children `upper`, `body` and `lower`, which are mapped to these slots. These slots can contain arbitrary editor cells, not just child collections: the `lower` slot contains a collection that projects the `name` property, an equals sign and the `lower` child.

Tables Tables can be used to represent collections of structured data or to represent two-dimensional concerns. For example, Fig. 4 shows a state machine

		Events	
		next(Trackpoint* tp)	reset()
States	beforeFlight	[tp->alt > 0 m] -> airborne	
	airborne	[tp->alt == 0 m && tp->speed == 0 mps] -> crashed [tp->alt == 0 m && tp->speed > 0 mps] -> landing [tp->speed > 200 mps && tp->alt == 0 m] -> airborne [tp->speed > 100 mps && tp->speed <= 200 mps && tp->alt == 0 m] -> airborne	[ ] -> beforeFlight
	landing	[tp->speed == 0 mps] -> landed [tp->speed > 0 mps] -> landing	[ ] -> beforeFlight
	landed		[ ] -> beforeFlight
	crashed		

Fig. 4. A state machine represented as a table; also shows nested headers.

rendered as a table. Another example used in mbeddr [1] is decision tables (essentially two nested `if` statements).

Tables come in several flavors. For example, a row-oriented table has a fixed set of columns and a variable list of rows. Users can add rows, but the columns are prescribed by the language definition. In contrast, the state machine shown in Fig. 4 is a cell-oriented table: users can add new columns (events), new rows (states) and new entries in the content cells (transitions). The language for defining tabular editors [6] takes these different categories into account. For example, the definition for the state machine uses queries over the state machine model to determine the set of columns and rows. The contents for the transition cells are also established via queries: each transition is a child of its source state and references the triggering event. Since both columns and rows can be added (or deleted) by the user, callbacks for adding and deleting both are implemented. The code below shows part of the table implementation for state machines.

```
table
  column headers:
    group "Events" {
      query {
        getHeaders (node)->join(string | EditorCell | node<> | Iterable) {
          node.inEvents(); }
        insert new header (node, index)->void { // callback for inserting }
        on delete: (node, index)->void { // callback for deleting }
    } }
  row headers: // similar
  cells:
    column count: node.inEvents().size;
    row count: node.states().size;
    cell: (node, columnIndex, rowIndex)->join(node<> | string | EditorCell | Iterable) {
      node<InEvent> evt = node.inEvents().toList.get(columnIndex);
      node<AbstractState> state = node.states().toList.get(rowIndex);
      node.descendants<Transition>.
        where({~it => it.parent==state && it.trigger.event==evt; });  } as vertical list
```

Prose with Embedded Code One characteristic of projectional editors is that the language structure strictly determines the structure of the code that can be written in the editor. While this is useful for code, it does not work for prose. Hence, an MPS plugin [7] supports "free text editing" in MPS: all the usual selection and editing actions known from text editors are supported. The resulting text is stored as a sequence of `IWord` nodes. By creating new concepts that implement the `IWord` concept interface, other specific nodes can be inserted into the sequence of words. Said differently, arbitrary structured program nodes can be embedded into the (otherwise unstructured) prose. The user can press `Ctrl-Space` anywhere in the prose block and insert instances of those concepts that are valid at this location. Fig. 5 shows an example of a requirement with a prose block that embeds a reference to another requirement. Other examples include references to arguments in function comments or embedded formulas.

Margin Cells Margin cells are rendered beyond the right editor margin; each margin cell is associated with an anchor cell inside the editor, and the margin cell is rendered at the y-position of that anchor cell. Fig. 6 shows an example. To use margin cells in the editor of some concept, the editor for that concept embeds a `margincell` cell which points to the collection that contains the margin

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

Fig. 5. The prose block includes a sequence of "normal" words plus a reference to another requirement (`§req(..)`). The reference is a real, navigable and refactoring-safe pointer, not just nice syntax.

contents (the comments in Fig. 6). The contents specified for the margin cell must implement the `IMarginCellContent` interface which contributes the facilities that connect the margin cell content to the anchor cell. Margin cells are available in the mbeddr.platform at `http://mbeddr.com`.

2.1 Hello, World

This tutorial showcases many of the features of mbeddr in an integrated example. The sources ZIP com.mbeddr.tutorial.zip is available from the download page at mbeddr.com. It is also part of the complete distro package.

Here is a comment.
23/06/14 15:58 (2 min ago) by markusvoelter

And a reply to it.
23/06/14 15:58 (2 min ago) by markusvoelter

Fig. 6. Margin cells used to support Word-like comments in MPS; other contents can be projected into the margin as well.

Graphics MPS supports editable graphical notations as shown in Fig. 7. They can be embedded into any other editor. MPS' support for graphical notations is new (available since MPS 3.1, June 2014) and not yet as mature as the rest of MPS, and the API for defining the editor is not yet as convenient as it should be. The code below shows part of the definition of the editor for Fig. 7: the contents of a block plus its ports are mapped as the contents of the diagram canvas.

```
diagram {
  content: this.contents,
           this.ancestor<Block>.allInPorts().toList,
           this.ancestor<Block>.allOutPorts().toList
  palette: custom AccentPaletteActionGroup
}
```

Custom Cells MPS supports embedding custom cells. This means that the user can plug in their own subclass of `CellProvider` and implement specific `layout` and `paint` methods. This way, any notation can be drawn in a low-level way. The cell provider can be parameterized, and ultimately, it can become a new, reusable primitive. Fig. 8 shows an example of a language that reports progress with work packages. It uses three custom cells: horizontal lines (parameterizable with thickness and color), check boxes (that are associated with boolean properties of the underlying language concept) and progress bars (whose percentage and color can be customized, typically by querying other parts of the model).

4 Other Features of the MPS Editor

As a language workbench, MPS supports the features known from traditional IDEs for custom languages. These include code completion, quick fixes, syntax

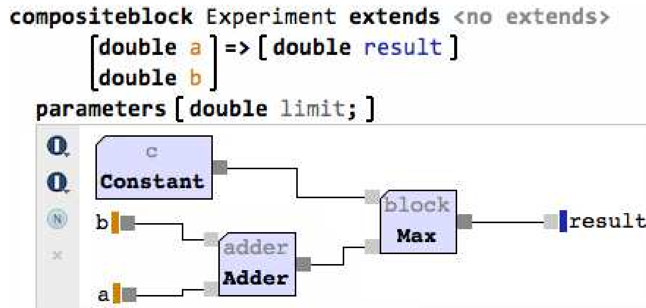


Fig. 7. A graphical editor embedded in a regular text editor.

coloring, code folding, goto definition, find references and refactorings. In this section we describe editor features that are specific to MPS' projectional editor.

Mixed Notations The various notations discussed in the previous section can all be mixed arbitrarily (with the aforementioned exception of embedding things *into* graphical editors). Since all editors use the same projectional architecture this works seamlessly. In particular, non-textual notations can be used inside textual notations. Examples include:

- mathematical symbols embedded in textual programs
- tables that contain text or math symbols
- tables embedded in textual programs
- mathematical symbols embedded in prose
- lines, progress bar other other shapes embedded arbitrarily

Multiple Editors A single concept can define several editors, and a given program can be edited using any of them. Each of the multiple editors has a tag, and by setting tags in an editor window (either by the user or programmatically), the editors corresponding to these tags can be selected. For example, state machines can be edited in a textual version (roughly similar to Fowler's state machine DSL [8]) or in the tabular notation shown in Fig. 4.

Partial Syntax Editors can also be partial in the sense that they do not project all contents stored in the AST. Of course the non-projected aspects of the program cannot be edited with this particular editor. But the contents remain stored in the AST (and are cut, copied, pasted or moved) and can be edited later. Using this facility, programs can be edited in ways specific to the current process

```
sorted: ☒    must be ok: ☐    hide ok ones: ☐
last updated: May 7, 2014 (4 months ago) by markusvoelter

FlightJudgementRules
| FasterThan100.impl (1)              24
| FasterThan200.impl (1)              32
| InitialNoPoints.inital (1)           8
```

Fig. 8. Custom widgets (checkbox, line, progress bar) used in an MPS editor.

```
atomicblock Adder realizes IAdder
    [ ________ ]=> [ __________ ]
    [ double a ]   [ double res ]
    [ double b ]
contract [ pre(0) positive_a: a > 0;  ]
         [ pre(1) positive_b: b > 0;  ]
         [ post(0) sum: res == a + b; ]
ccode { res = a + b; };
```

Fig. 9. A querylist is used to project the ports and contracts inherited from the interface realized by this block (in grey). New nodes ports or contracts can be entered above the grey lines.

```
adps/ ->des  Source:   _02DD
adps/ ->act  Info:     Controller Gain
/ ->desired  Type:     double/Nm/
/ ->actualT  Details:  constant<none>
/ ->ambient            range -10Nm..10Nm
[double/Nms
 double/Nm/ ->IntegralGain;     ]
```

Fig. 10. This tooltip shows the definition of the quantity referenced via the -> notation: it shows its type and various additional details. The tooltip uses the querylist to project derived nodes.

step or stakeholder. An example are the requirements traces shown in Fig. 14; programs can be shown with or without them.

Query List An MPS editor normally displays nodes at their phyiscal location. For example, the child `condition` of the `IfStatement` shown in Fig. 1 is projected as part of its parent editor. Sometimes, however, it is useful to render nodes in other places. An example is shown in Fig. 9: the grey parts are defined by the interface realized by the block, but they are still projected for the block itself.

To project nodes in locations where they are not defined, a `querylist` editor cell is used (available as part of the mbeddr.platform at `http://mbeddr.com`). Like other MPS collection cells it projects a list of nodes, but this list is assembled via an arbitrary model query. The result can be projected read-only (as in Fig. 9) or fully editable. The querylist also supports callback functions for adding new nodes (because it is not automtically clear where they would have to be inserted physically) or for deleting existing ones. This way, querylists support *views*.

Tooltips MPS can use the projectional editing facilities in tooltips (available in the mbeddr.platform). To define a tooltip, a special cell is inserted into the editor of the cell that should display the tooltip. Since the purpose of a tooltip often is to project information gathered from other parts of the model, tooltip editors often use querylists (Fig. 10).

Conditional Editors Conditional editors essentially support aspect orientation for editor cells. A conditional editor defines a decoration for existing editors as well as a pointcut that determines to which existing editor cells the decoration is applied. Figures 12 and 13 show examples. Importantly, these conditional editors can be defined *after the fact*, and potentially in a different language module. This way, arbitrary decorations can be overlaid over exiting syntax. The example in Fig. 12 renders an arrow above all references that have pointer type. Another example could be to change the background color of some nodes based on external data such as profiling times. By including a tooltip in the definition of the decoration, users can get more detailed information by hovering over the decorated part of the program. Another use case for conditional editors is the expression debugger shown in Fig. 11.

```
result = (10 + BASEPOINTS) * (alt + speed)
         25300: 20 = 10 + 10|BASEPOINTS ; 1265 = 1100|alt + 165|speed
```

Fig. 11. This expression debugger renders the values of all subexpression over or to the left of the expression itself. The original expression (without the debug info) is `(10 + BASEPOINTS) * (alt + speed))`.

Annotations Annotations are similar to conditional editors in that they can render additional syntax around (or next to) existing syntax without the original syntax definition being aware of this. However, in contrast to conditional editors, annotations are additional nodes (i.e., they are additional data in the program) and not just a property of the projection. The additional nodes are stored as children of the annotated node. Fig. 14 shows an example in which requirements traces are added to C code (details are discussed in [9]).

Read-Only Contents Especially in DSLs for non-programmers it is often useful to be able to project rigid, predefined, non-deletable skeletons of the to-be-written program in order to guide the user. For example, in Fig. 9, the keywords `atomicblock`, `realizes`, `contract` and `ccode`, as well as the brackets and lines, are automatically projected as soon as a user instantiates an atomic block. Similarly in Fig. 8, the grey line starting with "last updated" is automatically projected and consists of computed data. In MPS, parts of the syntax of a program can be marked as `readonly`, meaning that they cannot be deleted or changed. This does not just work for constants (keywords), but for arbitraty content (such as the inherited ports of blocks shown in Fig. 9).

5 Discussion and Summary

In this paper we have discussed the syntactic flexibility supported by MPS' projectional editor. We have described the various supported notational styles and emphasized that they can be combined flexibly. However, it is not enough to just compose different *notations* – other aspects of languages must be composed as well. Language composition with MPS is discussed in [2].

Over the last three years a team at itemis has been developing mbeddr [1], using MPS in a non-trivial development project. Many of the notations discussed

```
void normalizePosition(Position* p) {
  if (p->x > global->x) {
    p->x = global->x;
  } if
} normalizePosition (function)
```

Fig. 12. C references whose type is a pointer are annotated with the arrow on top. This works for all kinds of references, including to arguments, local variables and global variables.

```
pointerDeco editor pointerArrow for IRef
  cond: node.type.isInstanceOf(PointerType);
  node cell layout:
    [/
       $ custom cell $
       [> [next-editor] <]
    /]
```

Fig. 13. The editor applies to concepts that implemenet `IRef` and whose type is pointer. The editor renders the arrow (manually drawn in the `custom cell`) on top of the existing editor.

```
#constant TAKEOFF = 100; -> implements PointsForTakeoff
#constant HIGH_SPEED = 10; -> implements FasterThan100
#constant VERY_HIGH_SPEED = 20;
#constant LANDING = 100;
```

Fig. 14. The first two constants have traces attached. These are pointers to requirements shown in the code. The original definition of C is not aware of these annotations.

in this paper are used in mbeddr and its commercial addons. Several of the extensions have also been developed in the context of mbeddr. In addition, we are now also developing business applications (in the insurance and financial domains) with MPS. There, non-textual notations (and in particular, math and tables) are essential to be able to allow non-programmers to directly contribute to the programming effort. User feedback is very positive: they said that the abililty to have such notations is a signigicant advance over existing or alternative tools and approaches.

Based on this experience we conclude that the notations supported by MPS are reasonably complete relative to the notational styles encountered in practice. Classical textual notations are found in programming languages and DSLs; graphical notations are used by many modeling tools; mathematics are widespread in scientific or financial domains; tables are ubiquitous, as the the popularity of Excel demonstrates. And prose (with interspersed program elements) is an important ingredient to almost all these domains (for documentation, requirements or comments).

Acknowledgements Thanks to Bernd Kolb and Niko Stotz for feedback on this paper, and to Niko Stotz for building the margin cell notation.

References

1. Voelter, M., Ratiu, D., Kolb, B., Schaetz, B.: mbeddr: instantiating a language workbench in the embedded software domain. ASE Journal **20**(3) (2013) 1–52
2. Voelter, M.: Language and IDE Development, Modularization and Composition with MPS. In: GTTSE 2011. LNCS. Springer (2011)
3. Voelter, M., Siegmund, J., Berger, T., Kolb, B.: Towards user-friendly projectional editors. In: Proceedings of SLE'14. (2014) 20
4. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
5. Lisson, S.: MPS Math Plugin. `https://github.com/slisson/mps-math/`
6. Lisson, S.: MPS Tables Plugin. `https://github.com/slisson/mps-tables/`
7. Lisson, S.: MPS Richtext Plugin. `https://github.com/slisson/mps-richtext/`
8. Fowler, M.: Domain Specific Languages. 1st edn. Addison-Wesley Professional (2010)
9. Voelter, M., Ratiu, D., Tomassetti, F.: Requirements as first-class citizens. In: Proceedings of ACES-MB Workshop. (2013)

Putting the Pieces Together – Technical, Organisational and Social Aspects of Language Integration for Complex Systems

Håkan Burden

Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
burden@cse.gu.se

Abstract. Dealing with heterogenuous systems is often described as a technical challenge in scientific publications. We analysed data from 25 interviews from a study of Model-Driven Engineering at three companies and found that while the technical aspects are important, they do not encompass the full challenge – organizational and social factors also play an important role in managing heterogenuous systems. This is true not only for the development phase but also for enabling early validation of interdependent systems, where processes and attitudes have an impact on the outcome of the integration.

Keywords: Empirical and Exploratory Case Study, Model-Driven Engineering

1 Introduction

Complex systems, consisting of numerous and interdependent subsystems [15], require a plethora of languages for efficient implementation [7]. From the aspect of Model-Driven Engineering (MDE), the challenges are often described in technical terms [4] since heterogenuous languages imply different abstraction levels, representations and aspects of software [8], but also since the languages have their own domain-specific and platform-dependent constraints [11]. The one-sided focus on technical aspects is surprising since Kent already in 2002 pointed out that if MDE is to be successful it needs to encompass also the organisational and social aspects of software engineering [10], a claim that has since been reiterated [1, 9].

To explore to what extent language integration for comlex systems is a challenge in terms of technical, organisational and social aspects we analysed data collected at three different companies, looking for evidence regarding the motivations and challenges of heterogenuous development of embedded systems. Among the findings are that engineers tend to favour integration at the concrete code level instead of at the more abstract model level, that management needs to fit the right team with the right task and that which language you use identifies you as a software engineer.

The following section will describe the context of the three companies as well as how the data was collected and analysed. Section 3 structures the findings according to technical, organizational and social aspects of language integration. We then conclude and present our intentions to further investigate the interdependency between the factors in Section 4.

2 Model-Driven Engineering at Three Companies

During 2013 we conducted a case study of MDE at three companies – Volvo Cars Corporation, Ericsson AB and the Volvo Group. The respective organisations had different experiences with MDE but were all transitioning towards a more agile way of software development.

2.1 Heterogenuous Languages for Complex Systems

Electronic Propulsion Systems (EPS) is a relatively new unit at Volvo Cars with the responsibility of developing software for electric and hybrid cars. MDE was introduced in a step-wise manner as software development went from prototype vehicles to mass-production. The overall system design is described using AUTOSAR[1] while the software developed in-house at EPS is implemented using Simulink[2]. Simulink is also used for validation and integration purposes. The interfaces of the Simulink models are generated from the system-wide model. Besides graphical modelling languages, C is used for low-level details while numerous scripts help in everything from translating between different AUTOSAR-standards encoded in XML to deploying software on hardware.

Radio-base stations at Ericsson AB has employed different MDE technologies since the late 1980's, primarily focusing on UML as a descriptive and prescriptive modelling language including code generation. Besides using UML for design, implementation and testing various other languages form the engineers' tool box; C is used for functionality relying on optimal hardware performance, Java has a niche in functionality requiring GUIs, Erlang is regularly used for testing while home-made domain-specific languages – DSLs [12] – are used for specification, implementation and testing purposes.

Volvo Group Trucks Technology develops software for the Volvo Group's truck brands. While most of the software development is outsourced a few features are developed in-house using C. The interfaces of the top-level software components are automatically generated from the system model which is described in a company-specific dialect of EAST-ADL[3]. The two Volvo companies are independent in all aspects besides the name space which they share for historical reasons.

[1] http://www.autosar.org/
[2] http://www.mathworks.se/products/simulink/
[3] http://www.east-adl.info/

2.2 Data Collection and Analysis

The main source of data comes from 25 semi-structured interviews – 12 at Volvo Cars, nine at Ericsson AB and four at Volvo Trucks. The reason for fewer respondents at Volvo Trucks is that they entered the project later than the two other companies and have fewer engineers involved in MDE tasks. The interviews were audio-recorded and then transcribed. The interviews were complemented by a combination of observations, informal interaction [5] as well as seminars and regular meetings with representatives from the three companies.

The data has previously been analysed regarding the impact of tools on MDE adoption [16] as well as for comparing and contrasting MDE at the three companies [3]. For the purpose of this contribution we re-analysed the data deductively [14] searching for evidence concerning the technical, organisational and social aspects of heterogenuous systems and language integration across the model-driven engineering activities at the three companies.

3 Findings

As seen in the previous section, a variety of languages is used across the software. The variation comes both in terms of adapting languages depending on the nature of the included subsystems, but also due to where in the lifecycle the language is to be applied.

3.1 Technical Aspects of Language Integration

From the interviews a recurring theme is that the tool used for encoding a solution is just a means for producing low-level code. The following quote is from Ericsson, "*I don't see Rose RT or another modeling tool as a language. It's what they produce that is the language, and mostly it's always been C++ for us. So I don't think – I can consider, for example, Rose RT as a tool like Eclipse or something. Lets you develop code.*" A similar experience was encountered at Volvo Cars when one interviewee was asked about the impact of changing implementation language from C to Simulink, "*You still have C code.*"

The emphasis on the generated language is also dominant in how multiple languages are to be integrated. Where academic research is focused on composing modeling languages on a meta-level [4, 7], industrial practitioners prefer to integrate at the code level. At Ericsson where multiple languages are used in various combinations, integration is mainly done at the code level through the build environment. This coincides with our findings from a previous study on the integration of a graphical modelling language and a textual DSL conducted within another organisation at Ericsson [2].

Due to the number of suppliers and sub-contractors both Volvo companies rely on being able to integrate their subsystems in the form of binaries, so that merging on a meta-level is impossible. However, having access to the source would still be beneficial but for debugging purposes, not for merging the partial solutions.

One of the few examples where two languages are integrated in the same development environment comes from Ericsson where a DSL for testing was developed internally, "*The reason for it being that it could often take quite long time to compile some of our models [...] from a few minutes up to a few hours, depending on how big the model was.*" The answer was to define their own testing language on top of the modeling language and the interviewee was told that the DSL was developed during all the hours the team sat waiting for the model compiler to terminate.

3.2 Organisational Aspects of Language Integration

An engineer at Ericsson expressed the need for management to assign tasks according to the skills of the developers. "*In the big systems, we have different languages. Yes, and that is complex. And that's a problem. It's hard to expect that the software designer or verifier is equally good in all languages. And we've had to handle that in our team.*"

While one engineer at Ericsson saw the need for different languages according to domain and platform constraints – "*so you have all these levels and you have very different requirements for different parts of this product. It's so big that I don't even think that one solution fits all. You should be able to use several approaches*" – there is still an organisational wish to limit the number of languages to limit accidental complexity "*you shouldn't do it without a need. So you shouldn't – if you try to solve the same problem, I think you should try to use similar language or similar ways of working.*"

Stable interfaces in the decomposition of complex systems is desirable [4] but not always possible to obtain due to changing requirements or underspecification of new and ground-breaking features [6]. In these cases an agile organisation that lets developers work at both ends of the interface can be a way forward even if this demands that the developers master more than one language. But as one engineer at Ericsson said, "*using a new language is probably the smaller problem compared to learning the product and the domain and everything*".

3.3 Social Aspects of Language Integration

"*If you go to a different language, like, I don't know, whatever language, it will say that the for loop looks like this, but the functionality of it is the same. It doesn't matter how you put it in the words. Instead of 'for', you put like an 'f' or whatever. It's the same functionality. So when you know the base, you don't need to learn, like really study the new ones. You only adapt to them. From my point of view.*" The quote is from an engineer at Volvo Cars who explains his perception of using different languages for similar tasks.

However, not all engineers are interested in learning new languages. One interviewee from Ericsson described his experiences from developing a customer interface with a team located in a different town in the following way, "*an option was to do it in C++ because that's the most cost effective way. And the owner*

said something like 'I don't think that's a good idea because the organization we're from, people are working there because they want to do Java'."

At Volvo Trucks a similar sentiment was aired as the topic of introducing a new modelling language was raised, "*we have a lot of people that like to write C code and they like script languages. And they always do scripts for something. And they are pretty comfortable. They like writing a code with a blank page just writing C code.*"

4 Conclusions and Future Work

From a technical aspect of heterogenuous language integration there is a difference between the emphasis of academic contributions and industrial praxis in that while the former focus on merging languages on a source or meta-level the latter successfully integrating the target representations. This pragmatic approach is supported by proven techniques developed for integrating third generation programming languages. Organisationally the challenge seems to be for management to assign the right team – with the right skills – to the right task, a parallel challenge to the challenge of applying the right tools to the right problem [16]. Finally, from a social aspect it is not just enough that the engineers have the right skills – they also need to be open for using new languages. This is due to the fact that learning a new language is not a major obstacle but which language(s) you use is part of your identity as as software engineer and not all engineers are willing to redefine their competencies. In relation to Kent's critique of MDA [10], it seems that while the technical aspects of language integration have an important role to play in the development of complex systems, the possibilities for improved development and product quality can only be realised if the organisational and social aspects are seen as equally important.

In the case of setting up a simulation environment at Volvo Cars the challenge is not just to integrate different modelling languages but also agreeing on the same version of Simulink since different versions imply different properties in the generated code. Here, the challenge is organisational due to the fact that the developers want the newest features which enable new solutions while management responsible for integration need to know that the new version is stable before updating. Updating legacy models to comply with the newer versions can be both a time consuming and an error-prone task. With external organisations submitting their intellectual property in the form of human-readable models or code the question also becomes an issue of trust – not a technical factor of how to best compose two or more languages. How to organise an ecosystem [13] of simulation environments for continuous integration is still an open question we hope to address in future work by exploring how technical, organizational and social factors coincode in language integration.

References

1. Ameller, D.: Considering Non-Functional Requirements in Model-Driven Engineering. Master's thesis, Universitat Politcnica de Catalunya (June 2009)

2. Burden, H., Heldal, R., Lundqvist, M.: Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing. In: 6th International Workshop on Multi-Paradigm Modeling MPM'12. ACM, Innsbruck, Austria (October 2012)
3. Burden, H., Heldal, R., Whittle, J.: Comparing and Contrasting Model-Driven Engineering at Three Large Companies. In: ESEM 2014, 8th International Symposium on Empirical Software Engineering and Measurement. Torino, Italy (September 2014)
4. Combemale, B., Deantoni, J., France, R., Boulanger, F., Mosser, S., Pantel, M., Rumpe, B., Salay, R., Schindler, M.: First Workshop On the Globalization of Modeling Languages (GEMOC 2013). In: CEUR-WS (ed.) GEMOC - 1st International Workshop On the Globalization of Modeling Languages. pp. 3–13. Benoit Combemale, Julien DeAntoni, Robert France (September 2013)
5. Davis, K.: Methods for studying informal communication. Journal of Communication 28(1), 112–116 (1978)
6. Eliasson, U., Burden, H.: Extending Agile Practices in Automotive MDE. In: XM 2013, Extreme Modeling Workshop. Miami, USA (October 2013)
7. Hardebolle, C., Boulanger, F.: Exploring Multi-Paradigm Modeling Techniques. Simulation 85(11-12), 688–708 (November 2009)
8. Hardebolle, C., Syriani, E., Sprinkle, J., Mészáros, T.: Summary of the 6th International Workshop on Multi-Paradigm Modeling. In: Proceedings of the 6th International Workshop on Multi-Paradigm Modeling. pp. 5–6. MPM '12, ACM, New York, NY, USA (2012)
9. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 471–480. ICSE '11, ACM, New York, NY, USA (2011)
10. Kent, S.: Model Driven Engineering. In: Proceedings of the Third International Conference on Integrated Formal Methods. pp. 286–298. IFM '02, Springer-Verlag, London, UK (2002)
11. Mellor, S.J., Kendall, S., Uhl, A., Weise, D.: MDA Distilled. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2004)
12. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM computing surveys (CSUR) 37(4), 316–344 (2005)
13. Messerschmitt, D.G., Szyperski, C.: Software Ecosystem: Understanding an Indispensable Technology and Industry. MIT Press Books 1 (2005)
14. Runeson, P., Höst, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples. Wiley (2012)
15. Simon, H.: The Sciences of the Artificial. Karl Taylor Compton lectures, MIT Press (1996)
16. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In: Moreira, A., Schaetz, B. (eds.) MODELS 2013, 16th International Conference on Model Driven Engineering Languages and Systems. Miami, USA (October 2013)

Towards Integrating Modeling and Programming Languages: The Case of UML and Java*

Patrick Neubauer, Tanja Mayerhofer, and Gerti Kappel

Business Informatics Group, Vienna University of Technology, Austria
{neubauer, mayerhofer, gerti}@big.tuwien.ac.at

Abstract. Today, modeling and programming constitute separate activities carried out using modeling respectively programming languages, which are neither well integrated with each other nor have a one-to-one correspondence. As a consequence, platform and implementation details, such as the usage of existing software components and libraries, are usually introduced on code level only. This impedes accurate model-level analyses that take platform-specific decisions into account as well as the direct deployment of executable models on the target platform. In this work we present an approach for integrating existing software libraries with fUML models—an executable variant of UML models for which a standardized virtual machine exists—not only at design time but also at runtime. As a result of that, the modeler is empowered with the capabilities provided by existing software libraries on model level. Our approach is evaluated based on unit tests and initial case studies available in the ReMoDD repository that assess the correctness, performance, and completeness of our implementation.

1 Introduction

Back in 1966, the first object-oriented programming language was born. Its name is SIMULA and it combined modeling and programming in a unified approach, that is one of the strengths provided by object-orientation [4]. Carrying this idea further, the language BETA, which was developed based on SIMULA and DELTA, was designed for supporting both designing and programming systems. It even provided besides a textual syntax also a graphical notation for representing the same abstract syntax tree, such that the user could switch between both forms of representation. This was possible, since they had a one-to-one correspondence. Later, the graphical syntax of BETA was replaced by UML and, therewith, the one-to-one correspondence was broken referred to as impedance mismatch [4].

Today, the design of programming languages and modeling languages is largely separated from each other as it is carried out by different communities. Likewise, in today's mainstream object-oriented software development, modeling and programming are conceived as separate activities. This results in an impedance mismatch between modeling languages and programming languages. For instance, there is no one-to-one

* This work is co-funded by the European Commission under the ICT Policy Support Programme, grant no. 317859 and by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the FFG BRIDGE program grant no. 832160.

correspondence between the modeling concepts provided by UML and the programming concepts provided by Java.

In model driven engineering, this issue can be overcome by generative software development approaches utilizing intelligent code generators that are able to bridge the gap between modeling and programming languages for generating complete and deployable program code from models. However, these code generators add platform and implementation details, such as invocations of existing software components and libraries, to the resulting code, which are not reflected in the models. Hence, in model-level analyses targeted at validation, verification, or optimization, they cannot be considered. Therefore, platform and implementation details that are to be considered in model-level analyses have to incorporated into the model. One approach to achieve this is envisioned by Seidewitz [10] who proposes to avoid the "programming gap" at all, by incorporating all implementation details into the model by utilizing UML's action language. While this allows to incorporate algorithmic details into UML models, platform and implementation details concerning the usage of existing software components and libraries cannot be incorporated into the model in this approach. For instance, it is not possible to integrate an existing datastore available in terms of a software library into a UML model. This impedes accurate model-level analyses taking into account the behavior of reused software components and libraries, as well as the direct deployment of models on the target platform as envisioned by Seidewitz. The need for making existing software available on model level was also highlighted by Selic [11]. He describes that one way to accomplish this is by allowing direct calls to such existing software from within the model.

This paper is concerned with enabling the integration of existing software components with UML models. This enables on the one hand to construct more accurate UML models enabling model-level analyses closer to the target platform, and on the other hand a direct deployment of models on the target platform. In this work, that emerged from first ideas by Mayerhofer *et al.* [5, 6] and subsequent work in Neubauer's master's thesis [7], we propose the integration of software libraries with executable UML models developed with fUML—a subset of UML whose execution semantics was standardized by OMG in terms of a virtual machine (VM). More specifically, at design time, the software library is reverse-engineered into a UML class model representing the library's API structure, which is in the following used by the modeler to reference library classes and operations. At runtime, these references are used by an *Integration Layer* to locate, instantiate, and modify compiled stateful library components. As a result of this, the modeler can take advantage of the full power provided by already existing libraries as well as re-use existing software components. Thereby, we aimed at fulfilling the following requirements: (i) making it as natural to the modeler as possible to use existing software libraries (i.e., the modeler does not need to touch any source code), (ii) making the usage of libraries transparent to the modeler (i.e., the modeler can interact with library components just like with any other natively defined fUML component), and (iii) not extending the fUML metamodel or modifying the fUML VM as this would break the conformance to the standard.

The remainder of this paper is organized as follows. First, we introduce fUML in Section 2. Thereafter, we describe our approach for integrating software libraries with

fUML models in Section 3. In Section 4, we report on two initial case studies which we carried out for evaluating our approach. Related work is discussed in Section 5. Finally, Section 6 concludes the paper and provides an outlook on future work.

2 Foundational UML

fUML, introduced by OMG [8] in 2011, precisely defines the execution semantics of a subset of UML in terms of a virtual machine capable of executing fUML compliant UML models. Thereby, the UML subset considered by fUML includes UML class diagrams to define the structural aspects of a software system and UML activity diagrams to define its behavioral aspects. With the introduction of fUML, UML can be used as a programming language, that is nevertheless more abstract than existing third-generation programming languages [12]. Therewith, it seeks to overcome the model-code impedance mismatch by replacing source code entirely with fUML models.

However, fUML does neither provide nor foresees a functionality to provide access to existing software libraries. Instead, it intends to build its own library called *Foundational Model Library*. This library contains user-level elements, which can be referenced in fUML models. However, it is only composed of packages that define primitive types, such as Boolean, Integer, Real, and String, and a set of primitive functions for those data types (e.g., a function to concatenate two String values). While the first version of the fUML specification [8] already defines the Foundational Model Library, the latest version of the specification V1.1 still does not further extend the library's capabilities. When comparing the Foundational Model Library with libraries found in traditional third-generation programming languages, such as Java, the capabilities provided by, e.g., the Java Class Library (JCL) or any third-party library, are far beyond those of the Foundational Model Library. By looking at functionality provided by JCL, one can not only find features to build graphics and sound, access databases, and perform math operations, but also sophisticated abstractions on the underlying operating system and hardware to provide access to resources like the network or file system. While, from a technical standpoint, it is possible to extend the Foundational Model Library with the aforementioned functionalities, doing so is infeasible due to the required effort of adding them by the still rather small community currently using fUML. Furthermore, it is not only desirable to use JCL functionality in fUML models, but also to re-use already existing software components in fUML models, i.e., to re-use any existing software library.

3 Library Support for fUML

We presented the initial idea to integrate software libraries with fUML in [6]. In our approach, required classes of existing software libraries are integrated into the fUML model during design time by using reverse engineering techniques. During runtime, a dedicated Integration Layer is employed, which handles calls to any specified software library. Thus, the Integration Layer extends the fUML VM with the ability to handle access to software libraries. For this, it makes use of the command and event API developed for fUML within the moliz project [5]. This command and event API allows the

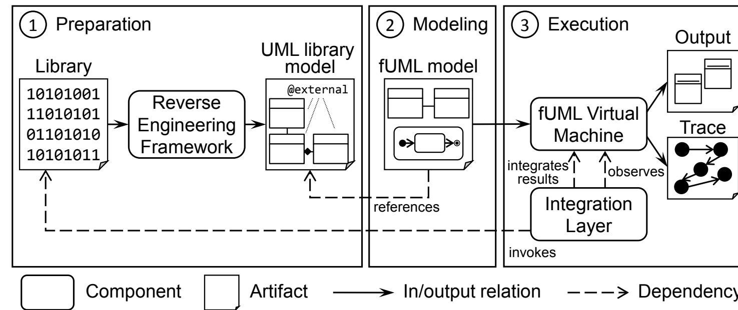


Fig. 1: Library support for fUML at a glance

Integration Layer to detect calls to software libraries by fUML models during runtime, and to re-integrate the result of the performed library call back into the fUML runtime environment. For performing the actual call to the software library, the Integration Layer makes use of reflection techniques.

The implementation of our Integration Layer[1] originating from Neubauer's master's thesis [7] currently supports the integration of Java libraries in fUML models. However, it is also possible to support other programming languages which provide reflection capabilities. The primary goal of the prototype is to provide the modeler with the possibility to instantiate Java library classes using CreateObjectActions of fUML, to modify Java library class instances using AddStructuralFeatureValueActions, and to call operations upon them via CallOperationActions. Additionally, the result of a library call, such as the return value of an operation call, is translated from Java to fUML and integrated into the fUML runtime environment.

Dynamic Class Loading and Reflection. The two major capabilities required by the Integration Layer to fulfill its goals are dynamic class loading and the reflection technique. The former ability builds upon the Java classloader that is part of the Java runtime environment and enables to dynamically load Java classes into the Java virtual machine. The reflection technique enables meta programming and hence allows the Integration Layer to instantiate and modify Java classes during runtime.

Figure 1 visualizes the entire approach in three steps from making libraries available to be referenced in the fUML model at design time to the point where the fUML model is executed. In the following, these steps are described in more detail.

Step 1: Preparation. In the *Preparation* step, the Java library to be used is prepared for being integrated with fUML models. Therefore, the structural information about the Java library's API is reverse-engineered into a UML class model (depicated as "UML library model" in Figure 1) either from the source code or a compiled JAR file using, e.g., the Eclipse plug-ins MoDisco[2] or Jar2UML[3]. To reference Java library classes and operations inside an fUML model, references from the fUML model to the obtained

[1] `https://github.com/patrickneubauer/fuml-library-support`
[2] `http://www.eclipse.org/MoDisco`
[3] `http://soft.vub.ac.be/soft/research/mdd/jar2uml`

UML library model may be created. Hence, the modeler can use the Java library in a natural way by referencing the reverse-engineered UML library model and does not need to deal with source code. To make the Java library usable at runtime, the UML library model is extended with UML comments indicating library classes as well as the location of the library's compiled JAR file.

Step 2: Modeling. Having obtained a UML library model from the Java library that is annotated with the location of the compiled JAR file, the library can be used in the development of an fUML model in the *Modeling* step. For this, any Eclipse-based UML editor like, e.g., the Eclipse UML Model Editor or the Papyrus UML Model Editor[4] can be used. Whenever the fUML model has to access the Java library, references from the fUML model to the UML library model are created. In more detail, to create an instance of a Java library class, a CreateObjectAction may be used having the classifier reference set to the UML class representing the respective Java class in the UML library model. Using an output pin, the created object can, e.g., be provided to the target input pin of a CallOperationAction. Such a CallOperationAction can be defined to call an operation on the Java library object. For this, the operation reference of the CallOperationAction has to be set to the respective UML operation representing the Java class' operation in the UML library model. To specify the value of an existing Java object's member field, an AddStructuralFeatureValueAction can be used referring to the respective UML attribute in the library model via its structuralFeature reference.

Step 3: Execution. To start the execution of an fUML model in the *Execution* step, the Integration Layer passes the fUML model to the fUML VM. Furthermore, using the command and event API provided by the moliz project [5], it registers itself as a listener to the fUML VM. Hence, the Integration Layer is notified about any event occurring during the execution of the fUML model and acts upon any required library access. For example, if the execution of a CreateObjectAction is completed, an *activity node exit event* referring to the CreateObjectAction is received. In case the action refers to a UML class contained by the UML library model, the Integration Layer detects that it has to instantiate the respective Java class. Similarly, modifications of existing Java objects via AddStructuralFeatureValueActions and Java operation calls via CallOperationActions are detected by the Integration Layer. The result of the library access, such as the instantiated Java object or the return value of an operation call, is integrated back into the fUML runtime environment by translating it from Java to fUML and providing it to the fUML VM. Thus, resulting values can be further processed during the fUML model execution. This means that an Java object instantiated by the fUML model via a CreateObjectAction may be modified by the model using *AddStructuralFeatureValueActions* and *CallOperationActions*. Thus, the Integration Layer makes state full Java objects accessible at runtime. As an example, we describe the handling of CreateObjectActions to instantiate Java classes in more detail. A detailed description of handling AddStructuralFeatureValueActions and CallOperationActions can be found in [7].

Handling a CreateObjectAction. When the Integration Layer receives an *activity node exit event* concerning a CreateObjectAction referring to a library class, it interrupts the model execution and performs the following steps.

[4] `http://www.eclipse.org/modeling/mdt`

1. **fUML Placeholder Object Retrieval.** As a result of executing the CreateObjectAction, the fUML VM created an fUML object of the referenced UML class, which is contained by the UML library model. This fUML object represents a stub or placeholder object of the Java object to be instantiated.
2. **Java Object Creation.** The Java class to be instantiated is determined by examining the classifier reference of the CreateObjectAction referring to the UML class that represents the Java class in the UML library model. While the namespace and name of this UML class uniquely identify the Java class, the location of the JAR file containing the Java class is captured in a UML comment owned by that UML class. Being supplied with both the unique identification and location of the Java class, the Integration Layer uses the reflection technique to create an instance of that exact Java class.
3. **fUML Object Creation.** To integrate the instantiated Java object into the fUML runtime environment, it is first transformed into an fUML object. For this, a new fUML object is created and its feature values are set according to the Java object's member field values. For example, in case the Java member field is of type int, a corresponding fUML IntegerValue is assigned to the fUML object. After translation, the fUML placeholder object retrieved in the first step is replaced by the translated fUML object.
4. **CreateObjectAction Result Assignment.** Finally, the fUML object created in the previous step is assigned to the CreateObjectAction's result output pin in form of an object token containing a reference to this fUML object.
5. **Object Bookkeeping.** In order to use the instantiated Java object in later stages of the fUML model execution (e.g., by using a CallOperationAction to call a library operation upon this object), both the Java object and the fUML object are added to a dedicated map data structure.

4 Critical Discussion

Succeeding the development of the prototype, we carried out initial case studies that are available in the ReMoDD repository[5]. Two of them are presented in this section. Additional case studies are discussed in [7]. The main research questions we aimed to answer through these case studies are as follows:

- **Correctness.** Is the developed prototype correct? In particular, does the prototype work as expected? If not, what causes the malfunction?
- **Performance.** How is the performance of executing an fUML model accessing a software library compared to executing a similar plain Java application? How much overhead is imposed by the prototype on top of the fUML VM performance?
- **Completeness.** Can every library be used? What are the limitations?

Mail Case Study. In this case study, we built an fUML model that uses a software library to compose and send an e-mail to an existing e-mail address. The library

[5] http://www.cs.colostate.edu/remodd/v1/sites/default/files/fuml-extlib-examples.zip

used in this case is Apache's Common Email library[6]. Used fUML action types include ValueSpecificationAction, CreateObjectAction, and CallOperationAction. Within the fUML model, a *SimpleEmail* object is created and multiple calls upon it are made to specify various e-mail parameters, such as its subject and receiver. The parameter values itself are specified through multiple ValueSpecificationActions. At the end, the last operation call finally transmits the created e-mail to the specified recipient.

Petstore Case Study. In the Petstore case study, we developed a model that creates, stores, and retrieves objects through the Google App Engine datastore. To realize the access to the datastore, the Objectify API[7] was used. The fUML model created for the Petstore case study behaves as follows. Initially, domain entities, such as *Customer*, *Address*, and *Order*, are registered through the Objectify service. Then, for testing purposes, the datastore is setup to run on the local machine rather than in the cloud. Next, both a *Customer* and *Address* object are created and multiple of their features (e.g., first name, last name, and city) are specified. While all of those features are simple Strings, assigning the created *Address* instance to the *Customer* requires handling a CallOperationAction receiving a complex input parameter. Afterwards, the created *Customer* instance is saved in the datastore. In order to verify the success of the previous operation, the same *Customer* is looked up in the same datastore.

Correctness Results. To ensure the correctness of the prototype, we built unit tests for the individual prototype components. The correctness was also confirmed by the case studies. For the case studies, we built a Java application that implements the same functionality as the respective fUML model, and compared the results obtained by executing both. On one hand, in the Mail case study, the composed e-mail was correctly sent and received by the specified recipient. On the other hand, in the Petstore case study, both the instance of the *Customer* and *Address* were created, the *Address* was added as a complex field to the *Customer*, and the *Customer* was populated to the local datastore. Finally, the same *Customer* was retrieved from the datastore showing the Integration Layer's correct treatment of the Petstore case study. Hence, the developed prototype successfully obtains the desired result in both case studies.

Performance Results. Since the performance of the Mail case study heavily depends on the current state of the network, the performance has been evaluated with the Petstore case study, in which the datastore is placed on the local machine and, hence, is not liable to network interference. Table 1 shows the performance results of executing both the fUML model and a corresponding Java implementation. Every benchmark has been performed five times and the visualized results represent the median of all executions. When looking at both execution time and memory one can see that executing the fUML model takes up a considerable longer time and larger amount of memory. However, the Integration Layer implementation only imposes a very small additional overhead on top of the fUML VM. In case of 50 consecutive executions, the measured overhead amounts to approximately 3% in terms of execution time and less than 1 MB in terms of memory (not depicted in the table) constituting less than 1% in overall memory overhead. The remainder of the overhead is imposed by the model execution, i.e., the fUML VM itself resulting from the fact that the fUML VM is not yet as optimized

[6] `http://commons.apache.org/proper/commons-email`
[7] `https://code.google.com/p/objectify-appengine`

as the Java virtual machine. For example, while the fUML standard foresees concurrent threading of the execution, the Java implementation of the fUML VM does not (yet) provide such capabilities.

Completeness. The implemented prototype currently supports only a subset of fUML's actions. In detail, the following fUML actions have been taken into consideration during the development of the prototype: CreateObjectAction, CallOperationAction, and AddStructuralFeatureValueAction. While the prototype supports all kinds of primitive data structures offered by the fUML standard, complex data types are only supported in a subset of possible use cases. Moreover, for example, the Java reflection library as well as Java libraries making use of dependency injection cannot be directly used as there are no corresponding concepts in the fUML standard. Additionally, static field and static operations are not supported. A list of explored limitations can be found in [7].

# of Executions	Time (Java)	Time (fUML)	fUML Overhead	Integration Layer Overhead	Memory (Java)	Memory (fUML)
1	583 ms	830 ms	42.37%	2.08%	5 MB	75 MB
10	936 ms	1260 ms	34.62%	4.03%	2 MB	77 MB
50	1199 ms	2541 ms	111.93%	2.75%	2 MB	88 MB

Table 1: Performance Results

5 Related Work

As discussed in Section 2, fUML defines its own library called Foundational Model Library. With this library, fUML provides primitive data types and associated primitive behaviors. Furthermore, this library can be extended by implementing for each data type or function to be added, a dedicated interface and registering this implementation at the fUML VM. Considering the vast amount of existing software libraries, which have to be added to the Foundational Model Library, in order to enable the development of fUML models executable on the target platform, and the still rather small community using fUML, this approach is currently infeasible from a practical point of view. Furthermore, re-using existing software components would require considerable additional implementation effort in this approach.

Nevertheless, some extensions of the Foundational Model Library exist. The Alf standard [9] provides a textual notation for fUML models and extends fUML with an additional library that provides further data types (e.g., Natural and Collection types) and corresponding functions (e.g., Collection functions). Cuccuru *et al.* [1] propose an extension of Alf with the Value Specification Language (VSL). VSL is a language standardized by the OMG for defining the values of non-functional properties of real-time and embedded systems in UML models. As part of this extension, additional data types and functions for these data types were added to Alf's library.

The Cameo Simulation Toolkit provided by No Magic, Inc. constitutes a commercial implementation of fUML. In this tool, it is possible to access software libraries in fUML models by embedding Java source code in the body of fUML OpaqueActions. This approach fundamentally differs from our approach as it requires the modeler to write actual source code. Furthermore, the result of executing source code embedded in the fUML model can only be accessed by source code embedded in another part of the fUML model. However, it is not possible to further process the result using (other) native fUML actions, such as the AddStructuralFeatureValueAction. A similar approach as the one implemented by the Cameo Simulation Toolkit is also provided by the commercial UML tools IBM Rational Rhapsody, IBM Rational Software Architect, and Enterprise Architect. While they do not fully conform to fUML, they support embedding source code in UML actions and thus enable access to software libraries.

A different approach is taken by the UML execution and debugging tool Pópulo [3]. In this tool, the supported action language can be extended using UML profiles. Therefore, the execution semantics of the introduced stereotypes have to be implemented by Java classes inheriting from dedicated classes of Pópulo's API. This approach is very similar to the approach of extending the fUML Foundational Model Library.

Another related approach we want to mention is Umple [2], which is a model-oriented programming language provided with a user interface that allows both graphical and textual modeling as well as programming in parallel. For modeling, Umple supports many UML modeling concepts used in UML class diagrams and UML state machines. Imperative code to be added to the model, such as code for class operation bodies, is written in one of the target programming languages supported by Umple, such as Java and PHP. In a code generation step, code for the respective target language may be generated. This approach differs from our approach in that it creates code from models which is then compiled and executed, while in our approach executable models are directly interpreted by the fUML VM.

6 Conclusion

In this work, we have presented an approach to integrate software libraries with UML models. In this approach, an existing software library can be integrated with fUML compliant UML models at design time, as well as at runtime during model execution. Therefore, the library to be used is reverse-engineered into a UML library model that represents the library's API structure. This UML library model can be referenced by an fUML compliant UML model through CreateObjectActions for creating stateful instances of library classes, AddStructuralFeatureValueActions for modifying these instances, and CallOperationActions for calling operations on these instances. During the fUML model execution, a dedicated Integration Layer operating on top of the fUML VM handles accesses to compiled stateful library class instances using the reflection technique and dynamic class loading.

The conducted initial case studies evaluated the feasibility of the approach particularly focusing on correctness, performance, and completeness. The correctness of the prototype has been successfully evaluated by unit tests and confirmed by the case studies. Regarding performance, we found that the total overhead caused by the Integration

Layer sums up to approximately 3% in execution time and less than 1% in memory. The remaining overhead is caused by the fUML VM, which suggests an optimization of the fUML VM. In terms of completeness, it was found that the implementation supports a subset of available fUML actions and libraries. As future work we plan to increase the set of supported fUML actions in the prototype, to conduct new case studies eventually discovering further limitations, and to revise the implementation for overcoming these limitations.

With our approach, fUML is enhanced with the rich power of existing software libraries. In particular, it enhances fUML with further capabilities, such as accessing operating system functionality as well as reusing existing software components. Thus, it allows to build more accurate models that are closer to the target platform and, hence, enables more accurate model-level analyses for validation, verification, or optimization purposes. Furthermore, our approach enables to develop fUML models that are executable on the target platform, such that the compilation of models to source code through code generation becomes obsolete.

References

1. Cuccuru, A., Gérard, S., Terrier, F.: Defining MARTE's VSL as an Extension of Alf. In: Proc. of 14th Int. Conference on Model Driven Engineering Languages and Systems (MODELS'11), LNCS, vol. 6981, pp. 699–713. Springer (2011)
2. Forward, A., Badreddin, O., Lethbridge, T.C., Solano, J.: Model-Driven Rapid Prototyping with Umple. Software: Practice and Experience 42(7), 781–797 (2012)
3. Fuentes, L., Manrique, J., Sánchez, P.: Pópulo: A Tool for Debugging UML Models. In: Companion of 30th Int. Conference on Software Engineering (ICSE'08). pp. 955–956. ACM (2008)
4. Madsen, O.L., Møller-Pedersen, B.: A Unified Approach to Modeling and Programming. In: Proc. of 13th Int. Conference on Model Driven Engineering Languages and Systems (MODELS'10), LNCS, vol. 6394, pp. 1–15. Springer (2010)
5. Mayerhofer, T., Langer, P., Kappel, G.: A Runtime Model for fUML. In: Proc. of 7th Int. Workshop on Models@run.time (MRT'12). ACM (2012)
6. Mayerhofer, T., Langer, P., Wimmer, M.: Towards xMOF: Executable DSMLs based on fUML. In: Proc. of 12th Workshop on Domain-Specific Modeling (DSM'12). ACM (2012)
7. Neubauer, P.: Integration of External Libraries into the Foundational Subset of UML. Master's thesis, Vienna University of Technology (2014), http://publik.tuwien.ac.at/files/PubDat_227306.pdf
8. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0 (2011)
9. Object Management Group: Action Language for Foundational UML (ALF), Version 1.0.1 (2013)
10. Seidewitz, E.: UML: Once More with Meaning (2013), http://www.slideshare.net/seidewitz/uml-once-more-with-meaning
11. Selic, B.: The Pragmatics of Model-Driven Development. IEEE software 20(5), 19–25 (2003)
12. Selic, B.: The Less Well Known UML - A Short User Guide. In: Proc. of 12th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'12). LNCS, vol. 7320, pp. 1–20. Springer (2012)

A Declarative Approach to Heterogeneous Multi-Mode Modelling Languages

Tony Clark

Department of Computer Science, Middlesex University, London, UK
t.n.clark@mdx.ac.uk

Abstract. This paper proposes a declarative approach to multi-mode heterogeneous DSLs based on term rewriting. The paper presents a data model and algorithm for processing syntax structures. It has been validated by an implementation that supports a range of languages. The paper includes an example language that supports both game construction and execution.

1 Introduction

Domain Specific Languages (DSLs) [25, 17] are motivated by the need to define languages that match specific use-cases, as opposed to General Purpose Languages (GPLs). Whilst GPLs are usually supported by standard text editors, DSLs, by their nature, often contain a range of more exotic syntax elements that are arguably better supported by syntax-aware editors. This has led to the development of a range of technologies to support DSL development and that generate tools for each DSL. Where the DSL is limited to text, languages such as EMFText [8], MontiCore [15, 16], TCS [10], and XText [6], MPS and Spoofax [11] allow a DSL to be quickly and conveniently defined and the associated tooling generated. These technologies are mainly based on grammarware [12] that integrate language parsers with editors in order to achieve a workbench. Many of the technologies integrate static and dynamic analysis of the resulting DSL. These technologies have become quite mature and the term Language Workbench [7] has been coined to describe this type of engineering tool.

Whilst languages used for programming or scripting tend to be exclusively text-based, modelling languages have included a much wider palette of elements. UML for example, has a number of sub-languages that are based on graphs, but also includes text in the form of OCL and action languages. Relatively few technologies support the definition and tooling of DSLs containing graphical syntax elements. Exceptions include Eugenia [13, 14], GMF [9], MetaEdit+ [23].

There has been increasing attention to *heterogeneous* (mixing graphical and textual notations) [1, 21, 5, 20]. Intentional Software and MPS are both developing tools that support *projectional editors* [22]. A recent model-based approach to mixing text and graphical languages is described in [2] that uses projectional editing techniques over a model. Whilst most of the reported work agrees on the general principles and proposed approaches, there has been little work on providing a concrete heterogeneous approach. In addition, most language use-cases

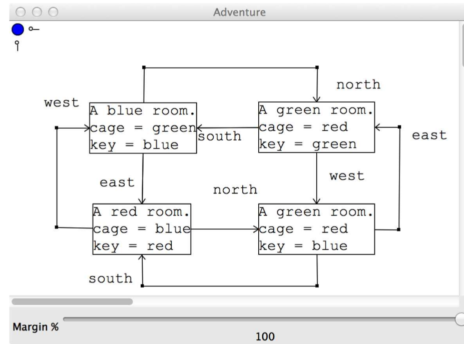


Fig. 1: Defining the Game

involve multiple modes, minimally *definition* and subsequent *use*. Other modes include *debugging* and using a language from the perspective of different stakeholders. Most DSL technologies do not support multi-mode interaction within the same tool-set.

This paper presents a novel declarative approach to the definition and associated tooling for heterogeneous multi-mode DSLs. The contribution is to propose that simple term rewriting can be used as the basis of this approach. This paper describes an algorithm that is suitable for this purpose and demonstrates how the declarative approach can be used to define a multi-mode heterogeneous DSL for building and playing a game. The approach has been validated by implementing the algorithm and the associated tool can be downloaded with examples.

2 Example

Consider a game that involves a collection of rooms that are connected by corridors. A room is either empty or contains a locked cage. The cage is painted red, green or blue. Inside the cage is a painted key. A key can be used to unlock a cage of the same colour and get the key inside. The player starts off in a room with a red key. The aim of the game is to visit all the rooms and unlock all the cages. Figure 1 shows the definition of a dungeon using the language editor for game construction. Rooms are created as nodes and corridors as labelled edges. The text in a room-node shows the colour of the room, the colour of a cage and the colour of the key in the cage. The blue dot at the top-left corner of the tool is used to access a room-creation menu. Edges between room-nodes are created by dragging the mouse from a source node to the target (a menu is used to select a direction). When a room-node is created, its colour and contents are uninitialised: the mouse is used to select from pre-defined colours for the room, cage and key.

The language operates in two modes: creation and play, it is possible to switch between the modes by pressing `p` and `c` on the keyboard. Figure 2 shows play mode. The player starts in the blue room with a red key. The player makes a move by pressing the first letter of the direction on the keyboard. Since the

player does not have a green key they must move from the starting room; they press `n` to go north and arrive at a green room with a red cage. The player can open the cage since their key matches the cage colour. This is done by pressing `u` on the keyboard. Finally, the player goes back south.

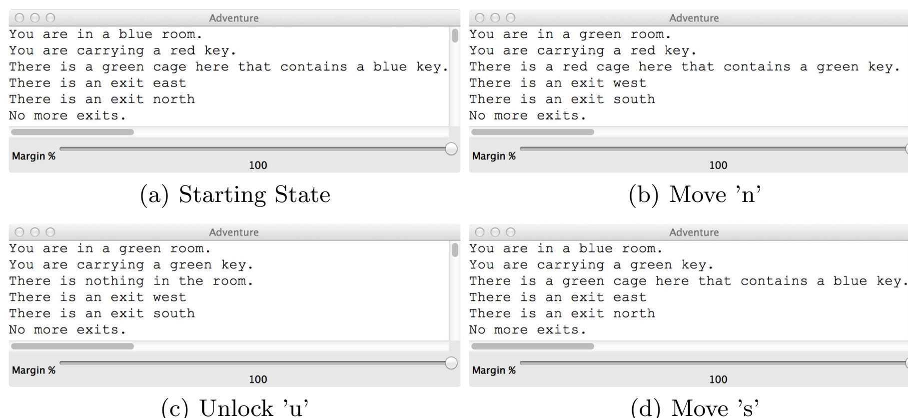
The game shows a number of features of the projectional editor. Interaction with the language can be moded; in this example there are two modes, but in general there can be any number. The abstract syntax can be projected on to graphs and text. In addition, language features can be created by menus made available as blue-dots. Figure 1 shows a blue dot that is used to create room-nodes, but in general a language may offer many different types of item. Figure 2 shows that the state of the game is projected to become formatted text. Figure 3 shows how the editor that is generated from the language definition supports creation of language elements: (a) creation of a new room element; (b) selection of a room colour; (c) selection of a type of edge between rooms.

3 Declarative Language Definition

The approach uses a simple term representation for both concrete and abstract syntax, and uses term rewriting as the technology to support all language modes. Section 3.1 describes the data representation and an rewriting algorithm, section 3.2 describes how concrete syntax is represented as trees in normal forms, and section 3.3 describes how rules are used to define the terms used to represent abstract syntax.

3.1 Syntax Trees and Transformations

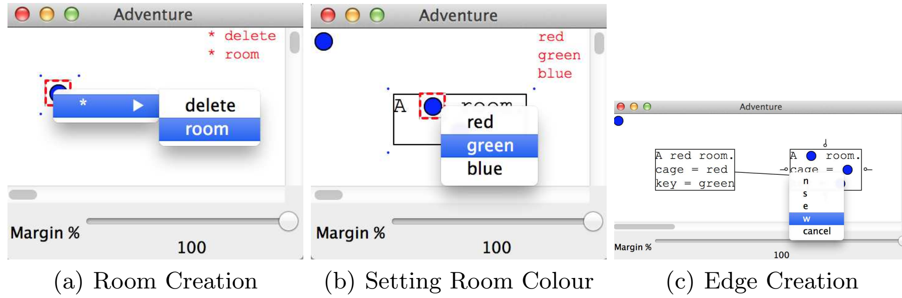
The DSL editor manages a syntax tree. A tree is in one of a number of forms: *atomic* in which case it is a string, number, char or boolean; *term* in which case it has the form `f[i1,...im](t1,...,tn)` where `f` is the term-*functor* which is a



(a) Starting State (b) Move 'n'

(c) Unlock 'u' (d) Move 's'

Fig. 2: Playing the Game



(a) Room Creation (b) Setting Room Colour (c) Edge Creation

Fig. 3: Editor Interactions

name, `i1,...,im` is the identity of the term, and `ti` are sub-trees. The identity of a term is used to associate a term with layout information. Where the identity of a term is not important it is omitted and assumed to be unique.

The editor works by creating an initial tree referred to as *abstract syntax*. The abstract syntax tree is *transformed* into a concrete syntax tree by applying pattern directed transformation rules. Concrete syntax is a tree whose term-functors are pre-defined.

The editor displays a concrete syntax tree, and then waits for a user-event. The set of user-events is pre-defined; each event is mapped on to a term whose functor designates the type of event: `key`, `selected`, `drag`, *etc.* The abstract syntax tree `t` and the new event-term `e` are combined into a new term whose functor is $\leftarrow$, written `t ← e`. The transformation process is repeated, allowing rules that process `e` to transform `t` appropriately.

Transformation rules have the form `p → e` where `p` is a pattern and `e` is an expression. A pattern is a tree that contains variables. A variable is just a name that starts with an upper-case letter. An environment `E` associates variables with either trees or lists of trees, and associates functors with functions. An environment that contains no lists is applied to a pattern to create a tree `E(p)=t`. For example, `{X ↦ 10}(f[0](X))=f[0](10)`.

Patterns may be repeated. Such a pattern must be nested within a parent pattern and is followed by ... The informal meaning of such a pattern is '*match as many elements as possible then continue*'. For example, the pattern `f[0](p...,10)` will match a term whose functor is `f`, whose identity is `0` and whose children end with `10`, providing that all other children each match pattern `p`.

An environment that applies to a pattern containing repeated sub-patterns will contain mappings between variables and lists of trees. For example,
`{X ↦[1,2,3]}(f(X...,10))=f(1,2,3,10)`. Repeated patterns need not be atomic as in the following example: `{X ↦[1,2,3]}(f(g(X)...,10))=f(g(1),g(2),g(3),10)`.

Expressions are patterns that can refer to local variables and local function definitions. A local function definition is a collection of rules that are selectively applied to part of a tree. For example, if `L=10` and `f(A)=g(A)` are local definitions then the term `x(f(L))` in the context of these definitions will be transformed to `x(g(10))`.

```
 1 proc transition(tree,locals,transformations,reductions,table) {
 2   while(true) {
 3     tree := transform(transformations,locals,tree);
 4     let tree' := transform(reductions,tree) such that tree' ∈ 𝒩
 5     table := display(tree',table);
 6     tree := get_event(tree',tree,table);
 7   }
 8 }
 9 fun transform(rules,locals,tree) {
10   while ∃ p → e ∈ rules and ∃ E ∈ ℰ such that I ⊆ E and E(p) = tree do {
11     tree := E(e)
12   }
13   let f[i_1,...,i_n](t_1,...,t_n) = tree
14   tree' = f[i_1,...,i_n](t'_1,...,t'_n) where t'_i = transform(rules,locals,t'_i)
15   if tree' = tree
16   then return tree
17   else return transform(rules,locals,tree')
18 }
```

Fig. 4: Projectional Editor Algorithm

A rule `r` is a pattern and an expression `p` $\rightarrow$ `e`. Given a tree, `t`, and some local definitions `L`, `r` is applicable when there is an environment `E` that contains `L` for which `E(p)=t`. The result of applying the rule to `t` is then `E(e)`.

The behaviour of the editor is defined in figure 4. The procedure `transition` performs a loop that transforms the abstract to the concrete syntax tree, displays the concrete syntax tree and then waits for a user event. The arguments of `transition` are: the abstract syntax tree, the local definitions, two sets of rules `transformations` and `reductions`, and a table that maps term identities to layout information.

Line 3 uses the transformation rules to change the current abstract syntax tree. This allows events to be change the state of the tree. Line 4 uses the reduction rules to transform the tree into a normal form, *i.e.*, a member of the set of trees $\mathcal{N}$ that can be drawn by the editor. The resulting concrete syntax `tree'` is displayed by the editor in line 5 using the table to remember the layout information on each loop within the procedure `transition`.

Line 6 waits for a user event. Such an event will occur with respect to the concrete syntax, so the table is used to make the correspondence between elements in `tree'` and in `tree`, resulting in a new abstract syntax tree of the form `t` $\leftarrow$ `e`.

The procedure `transform` is used to apply rules to a tree in the context of some local definitions. Lines 11-13 continually select a rule that is applicable and updates the tree. Once there are no more applicable rules, line 15 transforms the children of the tree. If any children have changed then the process is repeated (line 18) otherwise no more rules are applicable to any part of the tree and it is returned (line 17).

3.2 Normal Forms

The set $\mathcal{N}$ of normal forms contains trees whose functors and structure correspond to concrete syntax elements that can be drawn on a screen and that

can respond to user events. Different editors may define different sets of normal forms, for example a text-only editor may only support trees that correspond to string layout, whereas a graph editor may only support trees representing collections of nodes and edges.

This paper uses a game to explain the key features of the projectional editor approach to heterogeneous DSLs. The normal forms used by the game are:

`atom` Any atomic value is a normal form.

seq`(t1,...,tn)` The sub-tree normal forms are displayed in sequence.

nl Produce a new-line.

graph`(etypes,nodes-and-edges)` The graph is displayed on the screen and supports selection, new edge, movement, resize, and mouse click events. The identities of the nodes and edges are used to ensure that the layout is consistent; therefore, a node with a fresh identity will cause a new node to appear on the screen. The `etypes` define the permissible edge types, and `nodes-and-edges` is a mixed sequence of nodes and edges.

edge-types`(t1,...,tn)` each `ti` is of the form `type(source,target)` where `type` is the type designator for edges that can be drawn from nodes of type `source` to nodes of type `target`.

node`(ntype,i,display)` where `ntype` designates the type of node, `i` is the identity of the abstract syntax element represented by the node (for passing back events on this node), and `display` is a normal form that is displayed when this node is drawn.

edge`(source,sdec,target,tdec,label)` where `source` and `target` are the source and target node identities, `sdec` and `tdec` are the edge-end decorations for the source and target, and `label` is a label on the edge.

vbox`(pelements)` A vertical box of elements that are all of the form `l(e)` where `l` is one of the layout designators: **centre**; **left**; **right**.

3.3 Abstract Syntax

The editor must start with an initial abstract syntax structure. This is defined by an **abstract** clause in the language definition that corresponds roughly to a type definition for abstract syntax trees. The clause consists of a number of rules of the form `name` → `element` where `element` is one of: a term of the form `f(e1,...,en)` where each `ei` is an element; **str** in which case the element denotes an editable string in the concrete syntax; `e*` where `e` is an element in which case the abstract syntax denotes a sequence of `e`'s of arbitrary length, and is manifest in the concrete syntax in the form of a *hole* that can be selected and incrementally extended via a menu; a disjunction `e1 | e2` which will manifest itself in the concrete syntax as a hole associated with a menu that allows the user to choose between filling the hole with `e1` or `e2`. For example:

```
1 abstract {
2   numbers → number*
3   number  → zero | add(number)
4 }
```

represents an abstract syntax tree that is generated from the first rule (`numbers`). Since the first rule is `number*` we do not know how many instances of `number` to generate, so a hole is displayed allowing the user to generate a `number` followed by another hole, or to delete the hole (completing the sequence). A `number` also generates a hole allowing the user to choose between replacing the hole with a `zero` or a tree containing another number. Although the display of the holes is fixed, the actual representation of terms of the form `add(add(zero))` will depend on the reduction rules that map it into a normal form.

4 Game Implementation

This section describes the game implementation in terms of the abstract syntax definition, the locals, the transformation rules and the reduction rules. The abstract syntax for the game is:

```
1 abstract {
2   game   → game(construct,map(rooms(room*),exits),player)
3   room   → room(colour,empty | cage)
4   colour → red | green | blue
5   cage   → cage(colour,colour) }
```

The game is initially in `construct` mode which means that it will be displayed as a graph and allow new rooms and exits to be added. The map contains an extensible sequence of rooms and empty `exits` and `player` terms. A room has a colour and is either empty or contains a cage. A cage term contains two colours, one for the cage-lock and the other for the key in the cage.

The locals defines the edge types used between room-nodes and a function called `exits-from` that is used to map a room id and a list of all exits from all rooms, to just the exits from the designated room:

```
1 locals {
2   E = edge-types(n(room,room),s(room,room),e(room,room),w(room,room))
3   exits-from(I,Exits) =
4     case Exits {
5       exits → nil
6       exits(exit(D,I,_),Exit...) → cons(D,exits-from(I,exits(Exit...)))
7       exits(_,Exit...) → exits-from(I,exits(Exit...)) } }
```

The transformation rules are used to handle user events and use pattern matching to dispatch on the state of the game:

```
 1 transform {
 2   game(_,map(rooms(Room[I](Colour,Contents),R...),Exits),_) ← ↓p →
 3     game(play,map(rooms(Room[I](Colour,Contents),R...),Exits),player(I,red))
 4   game(_,map(Rooms,Exits),Player) ← ↓c → game(construct,map(Rooms,Exits),Player)
 5   game(play,map(Rooms,exits(X1...,exit(n,S,T),X2...)),player(S,Carrying)) ← ↓n →
 6     game(play,map(Rooms,exits(X1...,exit(n,S,T),X2...)),player(T,Carrying))
 7   game(play,map(Rooms,exits(X1...,exit(s,S,T),X2...)),player(S,Carrying)) ← ↓s →
 8     game(play,map(Rooms,exits(X1...,exit(s,S,T),X2...)),player(T,Carrying))
 9   game(play,map(Rooms,exits(X1...,exit(e,S,T),X2...)),player(S,Carrying)) ← ↓e →
10     game(play,map(Rooms,exits(X1...,exit(e,S,T),X2...)),player(T,Carrying))
11   game(play,map(Rooms,exits(X1...,exit(w,S,T),X2...)),player(S,Carrying)) ← ↓w →
12     game(play,map(Rooms,exits(X1...,exit(w,S,T),X2...)),player(T,Carrying))
13   game(play,map(rooms(R1...,Room[I](C,cage(C-Col,K-Col)),R2...),X),player(I,C-Col)) ← ↓u →
14     game(play,map(rooms(R1...,Room[I](C,empty),R2...),X),player(I,K-Col))
15   game(construct,map(R,exits(E...)),Player) ← new-edge(Type,S,T) →
16     game(construct,map(R,exits(exit(Type,S,T),E...)),Player)
17 }
```

If the user presses the p key at any time (line 2) then the game changes state to play. If the user presses n when the game is in play (line 5), and if there is an exit north from the player's current location S then the abstract syntax tree transforms into a new state where the player's current location is T. The player can unlock a cage using key u (line 13). Finally, if the game is in construct mode and the user drags an edge between two graph nodes, then the message new-**edge**(t,s,t) is send to the tree causing a new exit to be added to the map (lines 15,16). The reduction rules transform the current state of the game into a normal-form ready for display by the editor:

```
1  reduce {
2    game(play,map(rooms(R1...,Room[I](Col,Contents),R2...),exits(Exit...)),player(I,Carry)) →
3      player2str(Carry,Col,Contents,exits-from(I,exits(Exit...)))
4    game(construct,map(rooms(R...),exits(X...)),_) → graph(E,room2n(R)...,exit2e(X)...)
5    player2str(Carry,Col,Contents,X) →
6      seq('You are in a ',Col,' room.',nl,carry(Carry),contents(Contents),exits(X))
7    carry(Key-Colour) → seq('You are carrying a ',Key-Colour,' key.',nl)
8    contents(empty) → seq('There is nothing in the room.',nl)
9    contents(cage(C-Col,K-Col)) →
10     seq('There is a ',C-Col,' cage here that contains a ',K-Col,' key.',nl)
11   exits(nil) → 'No more exits.'
12   exits(cons(Exit,Exits)) → seq('There is an exit ',Exit,nl,exits(Exits))
13   exit2e(Exit[I](Type,S,T)) → edge[I](S,none,T,arrow,label['direction',I](target,Type))
14   n → 'north'
15   s → 'south'
16   e → 'east'
17   w → 'west'
18   room2n(Hole[I](H)) → node['new-room',I](new-room,I,H)
19   room2n(Room[I](Col,Stuff)) →
20     node['room',I](room,I,vbox['b',I](centre(seq('A ',Col,' room.')),centre(Stuff)))
21   red   → 'red'
22   green → 'green'
23   blue  → 'blue'
24   empty → 'empty'
25   cage(Cage-Colour,Key-Colour) → seq('cage = ',Cage-Colour,nl,'key = ',Key-Colour)
26 }
```

Lines 1 - 4 show the two rules that detect whether the game is being constructed or played. If played, then the game state is translated into text. If constructed then the game state is translated to a graph.

5 Implementation

The editor described in this paper has been implemented in the programming language Racket and used to define a range of heterogeneous languages. An implementation pack accompanies this paper[1]. The pack includes a Mac disk image stand-alone-editor.dmg of the editor implementation, several saved languages (*.xml) and the source code of the language definitions language-definitions.rkt. Once you have installed the editor, navigate to the bin directory and start the tool before dragging any of the xml files onto the editor pane to load up the language definition. The pack includes the game language definition, an example adventure, a use-cases implementation of hotel booking and a library class diagram. See the language definitions for more details.

[1] http://www.eis.mdx.ac.uk/staffpages/tonyclark/Software/projectional_editor_demo.zip

6 Conclusion and Future Directions

This paper has proposed a declarative approach to multi-mode heterogeneous DSLs. The approach freely mixes graphical and textual syntax and an algorithm has been presented to process the syntax structures. The algorithm has been validated by an implementation, but leaves room for future development. The approach is structural whilst other approaches integrate text parsing with projectional editing (*e.g.*, [4]); it may be possible to integrate both approaches. The meta-language described in this paper provides no support for error handling and will simply go wrong if the rules fail to produce a normal-form or if a local rule definition produces a tree of an unexpected type. One way to address this is to have a separate category of rules that are used for checking and error reporting. Related to this, the language does not support static checking. For example, it should be possible to detect the use of unbound identifiers and undefined functors. Some aspects of static checking should be easy to achieve, however it would also be desirable to define a type system so that the use of syntax structures can be checked before use. There is interest in the modularity and composition of languages and DSLs in particular [24, 3, 18, 19]. A key challenge to achieving engineered integration is posed by concrete syntax. By inverting the focus of attention to abstract-syntax, a projectional editor does not suffer from such problems. However, there are still significant issues to be addressed and this could be a fruitful area for future work.

References

1. Francisco Pérez Andrés, Juan de Lara, and Esther Guerra. Domain specific languages with graphical and textual views. In *Applications of Graph Transformations with Industrial Relevance*, pages 82–97. Springer, 2008.
2. Colin Atkinson and Ralph Gerbig. Harmonizing textual and graphical visualizations of domain specific models. In *Proceedings of the Second Workshop on Graphical Modeling Language Development*, pages 32–41. ACM, 2013.
3. Walter Cazzola and Ivan Speziale. Sectional domain specific languages. In *Proceedings of the 4th workshop on Domain-specific aspect languages*. ACM, 2009.
4. Lukas Diekmann and Laurence Tratt. Parsing composed grammars with language boxes. *Workshop on Scalable Language Specifications*, 2013.
5. Luc Engelen and Mark van den Brand. Integrating textual and graphical modelling languages. *Electronic Notes in Theoretical Computer Science*, 253(7), 2010.
6. Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.
7. Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.
8. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *Model Driven Architecture-Foundations and Applications*, pages 114–129. Springer, 2009.

9. Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. Language evolution in practice: The history of gmf. In *Software Language Engineering*, pages 3–22. Springer, 2010.
10. Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254. ACM, 2006.
11. Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010.
12. Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005.
13. Dimitrios S Kolovos, Louis M Rose, Saad Bin Abid, Richard F Paige, Fiona AC Polack, and Goetz Botterweck. Taming emf and gmf using model transformation. In *Model Driven Engineering Languages and Systems*. Springer, 2010.
14. Dimitrios S Kolovos, Louis M Rose, Richard F Paige, and Fiona AC Polack. Raising the level of abstraction in the development of gmf-based graphical model editors. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, pages 13–19. IEEE Computer Society, 2009.
15. Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular development of textual domain specific languages. In *Objects, Components, Models and Patterns*, pages 297–315. Springer, 2008.
16. Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*, 12(5):353–372, 2010.
17. Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.
18. Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. Language boxes. In *Software Language Engineering*, pages 274–293. Springer, 2010.
19. Bernhard Rumpe. Towards model and language composition. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*. ACM, 2013.
20. Markus Scheidgen. Textual modelling embedded into graphical modelling. In *Model Driven Architecture–Foundations and Applications*, pages 153–168. Springer, 2008.
21. Christian Schneider. On integrating graphical and textual modeling. *Real-Time and Embedded Systems Group, Christian-Albrechts-Universität zu Kiel*, 2011.
22. Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 451–464. ACM, 2006.
23. Juha-Pekka Tolvanen and Steven Kelly. Metaedit+: defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.
24. Federico Tomassetti, Antonio Vetró, Marco Torchiano, Markus Voelter, and Bernd Kolb. A model-based approach to language integration. In *Modeling in Software Engineering (MiSE), 2013 5th International Workshop on*. IEEE, 2013.
25. Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

Extensible Global Model Management with Meta-model Subsets and Model Synchronization

Dominique Blouin, Yvan Eustache and Jean-Philippe Diguet

Lab-STICC, Université de Bretagne-Sud, Centre de recherche, BP 92116
56321 Lorient CEDEX, France
{dominique.blouin, yvan.eustache, jean-philippe.diguet}@univ-ubs.fr

Abstract. We present an infrastructure for the management of models of heterogeneous meta-models in model-based development environments. The infrastructure consists of a Global Model Management (GMM) modeling language, which allows the capture of the meta-models used in a modeling environment. Relations between meta-models and subsets of these meta-models can be declared and interpreted during model evolution for automated global model management. The infrastructure is implemented in an Eclipse EMF based EDA (Electronic Design Automation) tool. Its use is demonstrated by the generation and synchronization of AADL and VHDL code targetting an FPGA to control a self-balancing toy car.

Keywords: Global Model Management, Model Coordination, Model Transformations / Synchronization, Triple Graph Grammars, EDA Tools

1 Introduction

Model-based engineering makes use of many models of different kinds to capture all aspects of a system. As presented in [1], a significant proportion of design errors are due to inconsistencies between the heterogeneous models used to develop the evolving system. Mechanisms are required to ensure consistency of models is automatically maintained, and that proper traceability links can be established between models and maintained during model evolution. Such mechanisms should be at the heart of every model-based development tool, and it should be extensible so that modeling tools can be easily configured to target new domains making use of other modeling languages and types of relations between models.

In this paper, we present a Global Model Management (GMM) infrastructure to solve these problems. It is inspired from state of the art research and from experience gained in developing the Kaolin EDA (Electronic Design Automation) tool [2], which makes use of several rich modeling languages such as AADL (Architecture Design and Analysis Language, [3]) and VHDL (VHSIC (Very High Speed Integrated Circuits) Hardware Description Language, [4]). The tool aims at simplifying the development of electronic systems implemented on FPGAs (Field Programmable Gate

Arrays) by generating automatically the platform-specific models and VHDL implementation code from abstract functional AADL models. Our GMM language includes the concept of meta-model subset as introduced in [5], to declare sets of constraints restricting the use of complex and rich languages such as AADL and VHDL. Validation of subset constraints ensure given activities can be performed on models. In addition, automated model synchronization is provided through an extension of the core GMM language making use of an enhanced version of MoTE [6], which is based on Triple Graph Grammars (TGG) [7].

The rest of this paper is divided as follows: the next section introduces the GMM language and its interpretation semantics. Then, section 3 demonstrates the use of the GMM infrastructure through an example consisting of a self-balancing radio controlled car whose implementation code is automatically generated using interpreted GMM relations. Related work is then presented in section 4, followed by the conclusion and perspectives in section 5.

2 The GMM Infrastructure

2.1 The GMM Modeling Language

Many approaches to GMM include the concept of mega-model, for which several definitions can be found. A unified definition is provided in [8], which has the advantage of being free of implementation details, and can be extended to make use of specific transformation tools and other artifacts. Our core GMM language (Fig. 1) is inspired from this work. It includes the central concepts of *model* and *relation*. A model is defined as an element that contains models and relations between models. It is hierarchical, meaning that it can contain other models as children. This allows for better structuring of models. For instance, and as explained in [8], large languages such as UML would benefit from such structuring by having some of their model elements declared as models (e.g. class diagrams, sequence diagrams, etc.).

Models need to be related to each other, so a mega-model must be able to contain *relations* between its contained models. A relation also owns an *intention*, which describes the intended use of the relation. Inspired from [9], we further distinguish between *factual* and *obligation* relations. A factual relation must be deleted as soon as its intention is not satisfied anymore. For example, if the intent of the relation is to provide traceability between two models, then if one model is deleted, the relation must be deleted because its intention is not satisfied anymore. On the opposite, an obligation relation defines that something should always hold but does not necessarily do so between the related models. Therefore, obligation relations should not be deleted when their intention does not hold, but provide means for (re-)establishing the validity of the relation. This is represented by the *establish validity* operation, which takes as input a *modeling environment* and an *execution context*. The modeling environment provides all models *concerned* by the relation, so that they can be processed to (re-)establish validity. The *execution context* describes the context in which the validity of the relation should be established. It captures the *type* of *operation* that was

performed on a *source model* of the modeling environment that was changed thus requiring validity to be re-established. Predefined types of operation follow the basic CRUD (Create, Read, Update and Delete) types used for database persistence. The establish validity operation returns a collection of models of the environment that have been updated as a consequence of (re-)establishing the validity of the relation.

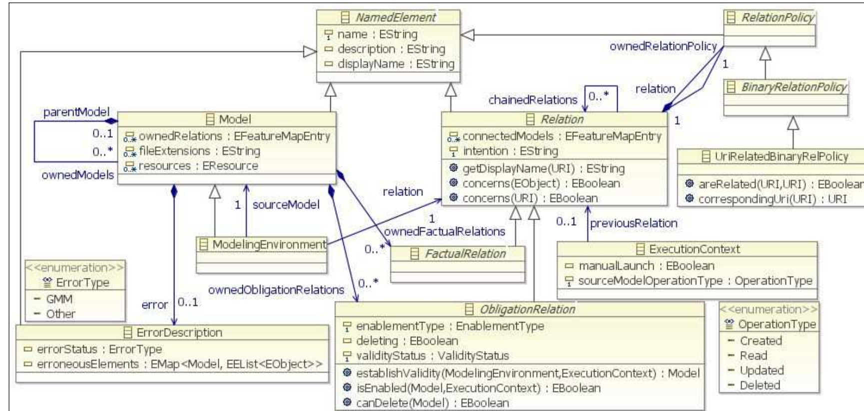


Fig. 1. The core GMM language

A model may be in an *error* state. For example, a model concerned by a relation may not be valid, and this information stored in the model can be processed by an obligation relation when re-establishing validity. In addition, the relation can itself update the error state on the models to indicate for example what prevents the validity of the relation to be established. This error information can then be displayed to users of the modeling tool, for instance through markers of the Eclipse environment.

The models that are concerned by a GMM relation must be determined in some way. For this purpose, the *relation policy* class is introduced. Subclasses can provide specific ways of relating models, for example, by determining that two models of different meta-models are related only when the file names of their resources have the same base file name, with meta-models being identified by file extensions.

A relation between models does not necessarily exist in isolation. Some relations may require other relations to hold. For example, a transformation chain can be seen as a set of chained obligation relations that must be executed one after the other to establish the validity of the chain of concerned models. For this reason, a GMM relation can declare *chained relations*.

A *meta-model* (Fig. 2) represents a specific type of model to which a model can be related through a *conformance* relation. It is also a place where useful information can be stored to be used by tools processing models of the meta-model. For instance, model comparison, which often needs to be customized per meta-model, can be specified by attaching model *comparison settings* to a meta-model.

Rich modeling languages such as UML or AADL often support a large number of *activities* (performance analysis, code generation, etc.). As pointed out in [5] for

AADL, guidance on how a language should be used for a given set of activities to be performed is essential to ensure tools interoperability. The authors of [5] proposed a DSML to capture *subsets* of modeling languages, and a revised version of this DSML is integrated in our GMM language, which introduces the concept of *meta-model subset* (Fig. 2). It consists of a set of *constraints* expressed in terms of the *cardinality* of a set of *model elements* of a given model. Various ways can be provided to construct sets of model elements, but this is still an ongoing work. The objective is to express subsets in a way that their constraints can be interpreted by tools without evaluation on a specific model, for customization according to a given subset. For example, the AADL graphical editor used in Kaolin can interpret a subset to automatically hide any element of the palette whose classifier is forbidden by the subset.

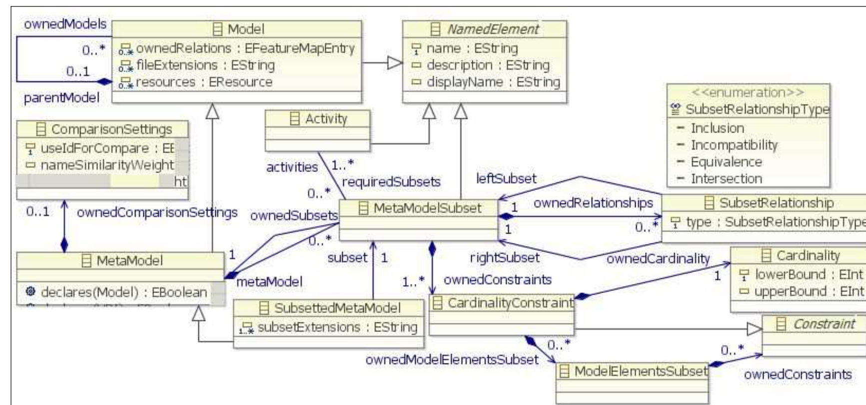


Fig. 2. The GMM concepts for meta-models and meta-model subsets

Following [5], meta-model subsets can be related to each other according to four types of relationships (*inclusion*, *incompatibility*, *equivalence* and *intersection*). A subset can be composed of other subsets through inclusion relations. It can also be associated with a given set of activities, thus allowing the analysis of tools interoperability according to their associated activities and subsets compatibility and equivalence.

A meta-model subset can be associated with a *subsetted meta-model* declared for a given meta-model. Models conformed to a subsetted meta-model are first validated against the meta-model, and then against the constraints provided by the associated meta-model subset.

2.2 Extension for Model Synchronization.

Model transformations are first class entities in model-based development. In GMM, they are represented as specific *relations* between models. Other types of operations between models could also be represented such as merge and refactoring, but it remains to be explored. Most model transformations are unidirectional and work in

a batch mode, i.e. from a set of input models they can only *create* an output model instead of updating an existing model. This is not sufficient since once generated, a model may need to be modified as it provides a different view of the system that may need to be updated by users. Hence, modifications must be reflected back in the source model to maintain consistency. Often this must be performed without re-instantiating the source model, since it may contain information that is not represented in the target model. Incremental transformations, which update only parts of a model, are therefore required. This is called model synchronization.

Only a few model transformation tools can currently satisfy these requirements. Among these tools, MoTE [6] can transform models in either directions using batch or synchronization mode. In addition, an enhanced version of MoTE has recently been developed [10], providing several improvements required for synchronize models of rich languages such as AADL. Being fully EMF-based, MoTE can be easily integrated into our GMM infrastructure in the form of a language extension (Fig. 3).

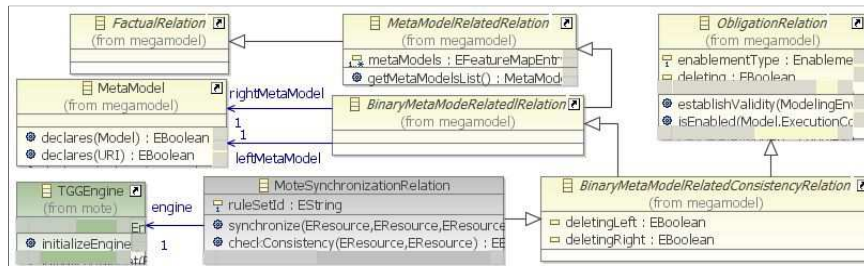


Fig. 3. The GMM extension for model synchronization with MoTE

The extension consists of a *MoTE synchronization relation*, which factually relates two models through their meta-models (*binary meta-model related relation*). At the same time, it is also an *obligation relation establishing* that the two related models must be maintained *valid* by ensuring their consistency. This is achieved by a MoTE *TGG engine* used by the MoTE synchronization relation.

2.3 GMM Model Interpretation

Our GMM language and its interpreter are deployed in the Eclipse Integrated Development Environment (IDE) as depicted in Fig. 4. A mega-model declaring the meta-models, their subsets and their relations is stored in the workbench configuration directory. A GMM controller listens for model change or read events (e.g. editor opening) sent by the Eclipse platform. For a given model source of a received event, the controller instantiates a modeling environment containing the models in the workspace and an execution context whose operation type reflects the type of the event. It then calls the GMM engine that interprets all relations declared in the mega-model that concern the models of the modeling environment. Obligation relation is currently the only type of relations interpreted in our GMM language. The GMM engine calls the *establish validity* operation passing the created modeling environment and execu-

tion context objects. For a MoTE synchronization relation, the operation consists of calling the appropriate operation on the associated MoTE TGG engine according to the specified execution context and for each target model of the modeling environment. For an execution context with a read operation, this means that the model loaded in memory may be updated in a next operation. If the model corresponding to the source model does not exist, the relation calls the engine to perform a batch transformation to create the model. Otherwise, a check consistency operation on the TGG engine is performed. Both of these operations cause a TGG correspondence model to be created between the models and stored in the TGG engine's memory. Later on, when the model is updated, a resource change event is sent to the GMM controller, which is translated into an execution context of type *Update* sent to the GMM engine. The MoTE relation then calls the TGG engine to synchronize the target models of the modeling environment. When a model is deleted, the corresponding model may be deleted or not, as specified by the relation's deletion properties.

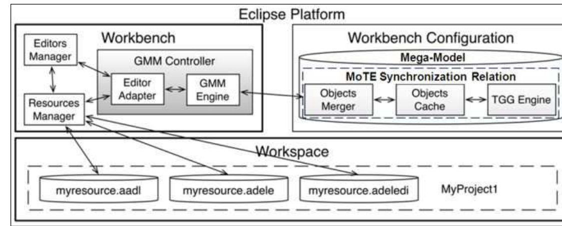


Fig. 4. The GMM infrastructure integrated in the Eclipse platform

The MoTE synchronization relation also takes care of creating appropriate errors carried by the models in case inconsistencies are detected during the creation of the TGG correspondence model. Model objects that are not mapped by the correspondence model are inspected according to a model elements coverage policy associated with the relation, which determines if unmapped elements should have been mapped or not. In the former case, this indicates that the models are inconsistent, and appropriate errors are set for the model elements. The MoTE relation will not process models until the inconsistencies are resolved through manual update of the models. The relation makes use of a cache of the model objects, which are linked by the correspondence models stored in the TGG engine memory. Changes made by any tool to a model are first merged into the cache, which preserves the object instances, thus ensuring the links of the correspondence model used to synchronize the models remain valid. The merge layer is implemented with EMF Compare [11], using comparison settings defined per meta-model declared in the mega-model.

3 Example

This section presents an example illustrating the use of the GMM infrastructure, where many details are omitted due to lack of space. It consists of an electronic sys-

tem implemented on an FPGA to control a self-balancing toy car (hereafter RC Car) from a smart phone (lower left part of Fig. 5). AADL is used to specify PIM and PSM models for the system. From the AADL PSM, a VHDL model is generated, which can be taken as input by FPGA vendor tools for synthesizing the circuit in the FGPA.

AADL is a component-based language for modeling both the software and hardware parts of embedded systems. It supports the specification of systems as an assembly of software and hardware components divided into categories. Software categories are thread, thread group, data, process and subprogram. Hardware categories are processor, virtual processor, memory, device, bus and virtual bus. Hardware and software component classifiers can be declared in libraries or hierarchically organized in systems for reuse. AADL components interact through features (interaction points) and connections, which together model data or control flows between components.

The first step to design a system in Kaolin is to create an AADL functional model independent of any execution platform (diagram of Fig. 5). It is then transformed into an AADL PSM, which describes the FPGA chosen by the user and the synthesized functions taking into account execution platform-specific details. The GMM language is used to specify the AADL and VHDL meta-models, including three subsetted meta-models for the AADL PIM and PSM, and for a subset of VHDL that can be handled by FPGA synthesis tools (synthesizable VHDL).

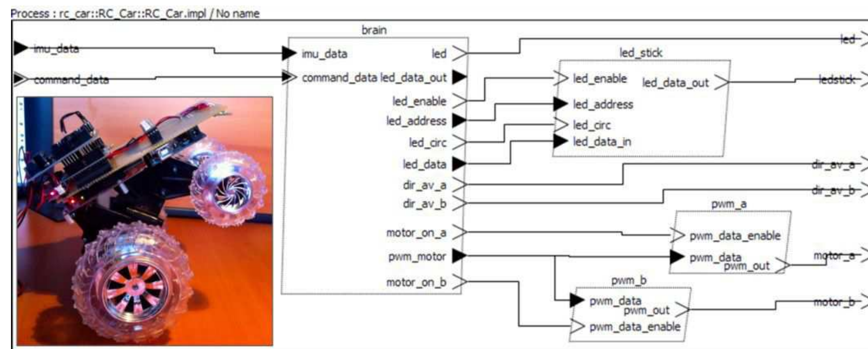


Fig. 5. The self balancing toy car and an AADL functional diagram for its control system

The objective of the AADL functional subset is to ensure AADL is used correctly for PIMs to be transformed into AADL PSMs, The functional subset includes a first subset that restricts the AADL language to its software part, which consists of forbidding the use of hardware constructs (processor, virtual processor, memory, bus, device and bus access). Additional constraints are then added to the functional subset to ensure only AADL *threads* and *data* subcomponents are used and contained in a single AADL *process* (Fig. 5).

From a functional AADL model, an AADL PSM is generated, conformed to an execution platform subsetted meta-model ensuring execution platform models are properly defined to be transformed into synthesizable VHDL code. Similar to the PIM subset, the PSM subset includes a subset restricting the constructs to hardware AADL

elements. Then, another subset describing how FPGAs must be modeled with AADL is provided, following an AADL extension developed to model FPGAs [12]. It includes the AADL hardware subset. Finally, the last required subset is *Synthesizable VHDL*, which cannot be described here due to the lack of space. Other VHDL subsets ensuring simulatability, testability and reusability, and as implemented by tools such as the Leda RTL checker [13] could also be modeled with our GMM language.

Once the required subsetted meta-models have been created, relations between models conformed to these subsetted meta-models can be declared to transform / synchronize the models. These relations are implemented as MoTE synchronization relations, allowing to maintain the consistency of models as they are updated, but also to check their consistency. The most complex relation is the *functional to FPGA execution* platform relation, which generates from an AADL PIM (Fig. 5) an AADL PSM (Fig. 6) describing the content of the synthesizable component of the specific FPGA platform selected by the user.

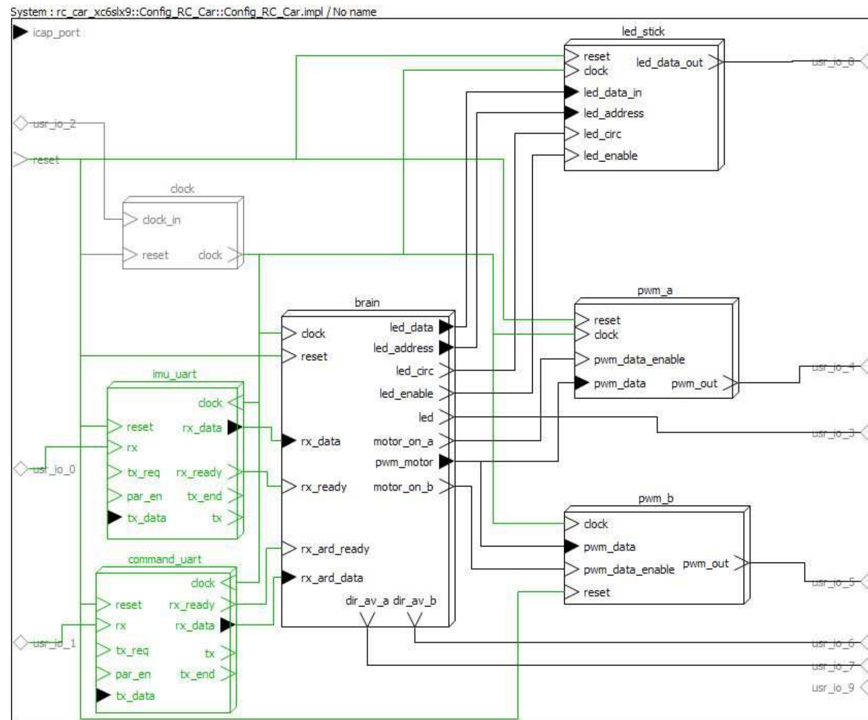


Fig. 6. An AADL execution platform model generated from the RC Car functional model

Each thread of the functional model is transformed into a processor subcomponent, which exhibits execution platform details such as clock and reset signals. The generated AADL FPGA component extends a template for the selected FPGA target. Grey

elements on the diagram are inherited from the template, which in this case includes a clock. Green elements on the diagram have been added after the transformation to take into account requirements for the specific FPGA. In this case, 2 UART (Universal Asynchronous Receiver/Transmitter) controllers (in green) have been added to fix communication incompatibility between the brain controller and FPGA ports predefined in the template. Adding these controllers is currently performed by a Java procedure called at the end of the transformation, but the intent is to implement this as a GMM relation. In this way, new refinement relations can be integrated in the mega-model to target other execution platform specific needs.

4 Related Work

Several approaches have been proposed for GMM, most of them making use of mega-models. A summary is presented in [8], with our GMM language derived from the proposed unified definition. In [9], dynamical traceability management is proposed through the categorization of relations into factual and obligation types, which was also introduced in our language. In [14], another infrastructure for GMM is proposed and implemented in Eclipse, which combines mega-models with model weaving. However, it only supports basic functionality such as model navigation through traceability links. Automated production of the links and model synchronization are not supported. Our work enhances these approaches with meta-model subsets and model synchronization based on automated traceability link production. Our approach is also extensible so that new relations and tools can be integrated as needed.

A difficulty in GMM is to identify the relationships that are needed between models. The GEMOC initiative [15] proposes an initial categorization of relations in three different forms: interoperability, collaboration, and composition. Interoperability supports the exchange of information across models with minimum coupling between the models. It seems similar to model weaving proposed in [14]. Collaboration relationships support *coupled development* of models, where the development of a model directly influences the form of other models. This is similar to our synchronization relation, which influences the form of the associated models by maintaining their consistency during model evolution. Finally, composition relationships combine information from several models to create new forms of models. This is similar to EMF views [16], where several meta-models can be combined to provide new views on models, similar to database views.

5 Conclusion and Perspectives

Our experiment in using GMM for our EDA tool shows that several benefits can be obtained through explicit representation of the used meta-models and subsets, along with relations between models. Interpretation of relations ensures models are properly managed during their evolution to prevent errors introduced early in the development process. We think every model-based IDE should include a GMM infrastructure. One advantage of our GMM is that it was developed using rich and realistic modeling

languages, which revealed important needs such as meta-model subsets and improved automated model synchronization.

However, many aspects of our infrastructure require improvements. Despite our enhancements, the TGG language would benefit from a complete review to improve aspects such as reuse of TGG elements across several TGGs. The study of other types of relations as proposed in [15] is also of interest, and in particular the integration of EMF views in GMM implementing meta-model composition relations. Finally, model to meta-model conformance could be enforced during meta-model evolution, represented as a conformance relation of obligation type, making use of frameworks such as Edapt [17].

6 References

1. P. H. Feiler, *Model-based Validation of Safety-critical Embedded Systems*, Aerospace Conference, pp. 1-10, 2010.
2. Y. Eustache, D. Blouin, M. Lanoë, J-P. Diguet, P. Coussy, Kaolin, a Model-based EDA Tool to Program, Reuse or Retarget Embedded Systems on FPGAs, Data Automation and Tests in Europe (DATE) conference, University Booth, 2014.
3. SAE International, Architecture Analysis and Design Language (AADL), http://standards.sae.org/as5506b/.
4. IEEE Standard VHDL Language Reference Manual, ANSI/IEEE Std 1076-1993, 199.
5. V. Gaudel, A. Plantec, F. Singhoff, J. Hugues, P. Dissaux, J. Legrand, *Enforcing Software Engineering Tools Interoperability: An Example with AADL Subsets*, IEEE International Symposium on Rapid System Prototyping (RSP), pp. 59-65, 2013.
6. The Model Transformation Engine (MoTE), http://www.mdelab.de/mote/.
7. A. Schürr, *Specification of graph translators with triple graph grammars*, in Graph-Theoretic Concepts in Computer Science, LNCS Volume 903, pp. 151-163, 1995.
8. R. Hebig, A. Seibel, H. Giese, *On the Unification of Megamodels*, Proc. of the 4th International Workshop on Multi-Paradigm Modeling (MPM), volume 42 of ECEASST, 2011.
9. A. Seibel, S. Neumann, H. Giese, *Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance*, Softw. Syst. Model, Volume 9, Issue 4, pp. 493-528, 2010.
10. D. Blouin, A. Plantec, P. Dissaux, F. Singhoff, J-P. Diguet, *Synchronization of Models of Rich Languages with Triple Graph Grammars: An Experience Report*, International Conference in Model Transformation (ICMT), pp. 106-121, 2014.
11. The EMF Compare Framework, http://www.eclipse.org/emf/compare/.
12. D. Blouin, D. Chillet, E. Senn, S. Bilavarn, R. Bonamy, C. Samoyeau, *AADL Extension to Model Classical FPGA and FPGA Embedded within a SoC*, International Journal of Reconfigurable Computing (IJRC), 2011.
13. The Leda RTL Checker, http://www.synopsys.com/tools/verification/.
14. F. Jouault, B. Vanhooff, H. Bruneliere, G. Doux, Y. Berbers, J. Bezivin, *Inter-DSL coordination support by combining megamodeling and model weaving*, Proceedings of the 2010 Symposium on Applied Computing (SAC), 2010.
15. B. Combemale, J. DeAntoni, B. Baudry, R. B. France, J-M. Jezequel, J. Gray, *Globalizing Modeling Languages*, Computer, vol.47, no.6, pp.68-71, June 2014.
16. The EMF Views Project, http://emfviews.jdvillacalle.com/.
17. The Edapt Project, http://www.eclipse.org/edapt/.

Towards an Ontology-based Approach for Heterogeneous Model Matching

Mahmoud EL HAMLAOUI[1, 2], Cassia TROJAHN[1], Sophie EBERSOLD[1] and Bernard COULETTE[1]

[1] IRIT Laboratory, MACAO Team & MELODI Team
{firtname.lastname}@irit.fr
[2] ENSIAS, SIME Laboratory, IMS Team, University of Mohamed V Souissi

Abstract. The overall goal of our approach is to relate models - of a given domain - that are designed by different actors in different Domain Specific Languages, and thus are heterogeneous. Instead of building a single global model, we propose to organize the different source models as a network of models, which provides a global view of the system through a correspondence model. This latter, conform to a correspondence meta-model is built via a manual matching process. In this paper we explore the possibility of representing models as ontologies and take advantage of an automated process to match them.in order to enhance the automation of the matching process.

Keywords: DSL, heterogeneity, correspondence, matching, refine, ontology

1 Introduction

Models manipulated by different actors within the design process of complex systems are by nature heterogeneous. One common way to describe such models is to use Domain Specific Languages (DSLs) [21]. Matching these models is one of the key tasks for ensuring the consistency of the overall system. However, due to the specific way these models are represented, correspondences between heterogeneous models are generally established manually by the domain expert [3]. Approaches for automating the matching process usually consist in transforming the input models to other representations, such as tree models [7] or directed labeled graphs [1], before the matching step itself. In this paper, we propose to take advantage of ontologies, which provide a formal representation of the knowledge about a given domain [19], and automated ontology matching approaches [12], in order to automate and optimize the model matching process. Our proposal extends a previous work on matching models through a correspondence model [13]. Correspondences link two or more elements belonging to different models. They are first established at meta-model level, and are then instantiated (refined) at model-level. We distinguish domain independent relationships (e.g., “dependency”, “generalization”, “aggregation”, and “similarity”) from domain specific ones. In our previous proposal, the first step of the matching process consists in automatically refining a set of correspondences at model-level, from corre-

spondences established by the domain expert at meta-model level. The second step consists in filtering out potential incorrect correspondences from the set of instantiated ones. In [3], we have proposed to refine the correspondences from meta-model level to model level in a semi-automatic way, the expert being in charge of filtering (with the help of a tool) the correspondences that have been generated at model level. In this paper, we propose to exploit ontology-based matching approaches for enhancing the automation of the matching process at model level. We illustrate our approach with a case study extracted from a Conference Management System (CMS). We focus in this paper on the similarity relationship between two model elements.

The rest of this paper is organized as follows. Section 2 introduces the case study that was chosen to illustrate our approach. Section 3 presents our correspondence meta-model and the matching process between meta-models and models. Section 4 discusses how correspondences at model level can be established through refinement of correspondences at meta-model level. Section 5 presents how we use ontologies in our approach. Section 6 evaluates the performance of our proposal. Section 7 compares our proposal to related work, and finally Section 8 concludes and draws perspectives.

2 Running example: Conference Management System (CMS)

CMS is a software system aiming to automate functions needed in the management of a scientific conference: call for papers, paper submission, paper assignment for evaluation, notification of the final decision, registration, etc. Even if it is not really a complex system, the CMS has been chosen because it is firstly well known to most of researchers; secondly, it is relevant because it involves different actors, working with different points of view. We consider that there are three business domains in the CMS, covering various aspects of modelling. Each business domain is described by a dedicated model, conform to a dedicated meta-model, and is manipulated by actors with specific roles: (a) *Software Architect*: responsible for the conference management design; he creates a model – called *softwareDesign* – expressed through a specific software design meta-model; (b) *Database Administrator*: responsible for storing data; he creates a model – called *persistence* – expressed through a persistence meta-model; (c) *Process Engineer* : responsible for the way to conduct a CMS; he creates a model – called *businessProcess* – expressed through a business process meta-model. Due to lack of space the CMS meta-models are not described in this paper. Fig.1, Fig.2 and Fig.3 show fragments of the partial models of CMS.

3 Meta-model and Model Matching Process

Meta-model for representing correspondences (MMC). Establishing correspondences between models is a way to ensure and to maintain consistency of the whole system. Our approach consists in analyzing input (meta)-models in order to identify correspondences that exist among them. These correspondences are then stored into a model of correspondences (M1C) which is conform to a meta-model of correspond-

ences (MMC). It expresses links among meta-elements of distinct meta-models as well as links among elements of distinct models. MMC is composed of a generic part common to all the domains, and of a specific part which depends on the specific domain modeled (Fig.4). The main meta-classes of this correspondence model are : (a) *CorrespondenceModel:* represents all the correspondences established among at least two (meta)-elements belonging to different (meta)-models; (b) *Correspondence*: composed of a relationship and its extremities, which represent elements from input models; (c) *Relationship*: abstract meta-class that defines relationships among (meta-) elements of different (meta-)models and allows for defining n-ary correspondences connecting more than two elements at once. Its definition is done through specialization of ”Relationship”, by introducing two sub-meta-classes: “DomainIndependentRelationship” and “DomainSpecificRelationship”; (d) *DomainIndependentRelationship*: abstract meta-class that represents the generic relationships that may exist in different domains; and (e) *DomainSpecificRelationship*: abstract meta-class representing relationships among models in a specific domain. New relationships are specified by specialization of this meta-class according to the studied domain. The other meta-classes will be described progressively in subsequent sections.

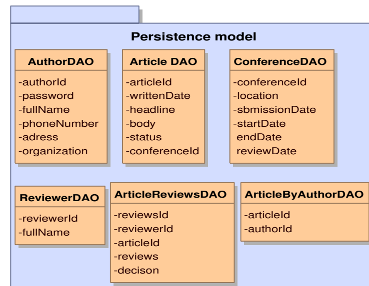


Fig. 1. Extract of the persistence model

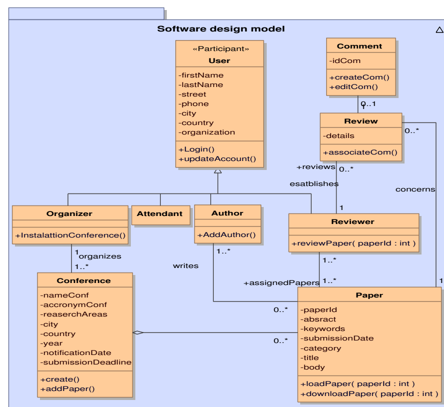


Fig. 2. Extract of the software design model

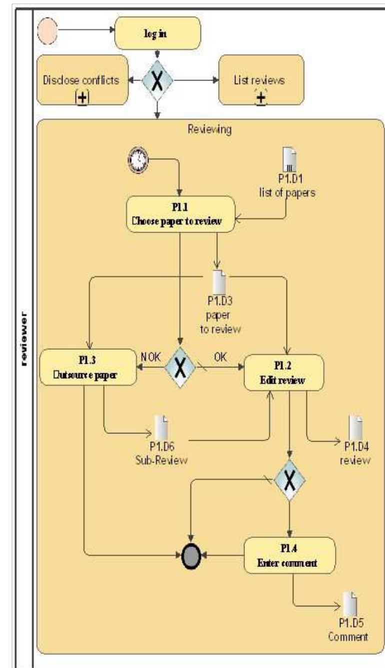


Fig. 3. Extract of the business process model

Global matching process. The correspondence model (M1C) is created via a “matching process”. This process, described in [13], involves two actors, namely, a domain

(design) expert who can be seen as an orchestrator of the system, and a tool to support the automated phases. The process takes as input the various models, their respective meta-models and the kernel of the meta-model of correspondences (the generic part). It begins by identifying correspondences between meta-elements so as to produce an intermediate correspondences model called M2C. Correspondences stored in M2C are called High Level Correspondences (HLCs). HLCs are then refined in order to produce Low Level Correspondences (LLCs) at model level.

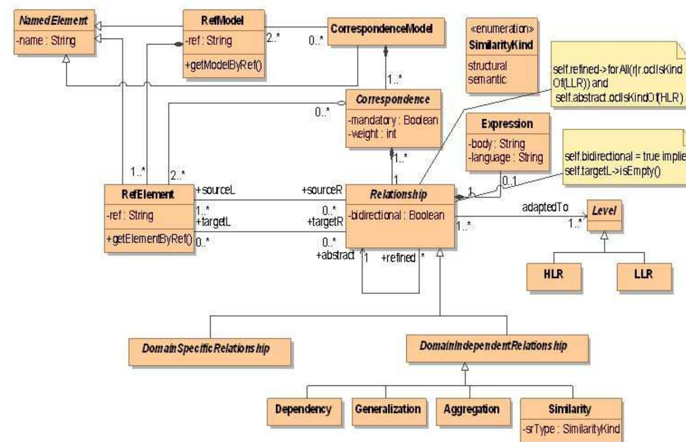


Fig. 4. Overview of the generic part of the meta-model of correspondences (MMC)

4 Defining correspondences on the CMS case study

HLCs stored in the M2C model are defined only once during the modelling cycle. Fig.5 shows examples of HLC correspondences using 5 types of relationships. For example, a correspondence encompasses a link between the meta-element “Task” on one side and the meta-element “Operation” on the other side through a “Contribution” relationship. A “Similarity” is defined between the meta-elements “Property” and “Column”, but also between the meta-elements “Table”, “Entity” on one side and “Table” and “DataObject” on the other side. The last type of relationship, called “Aggregation”, is used to relate the meta-elements “Property” and “Column”.

HLCs are exploited to generate LLCs stored in the M1C. Actually, LLCs are obtained by refinement of HLCs. In software and system engineering, refinement is a classical way to reuse [9] [10] [4]. It can be seen as a way of crossing different levels of abstraction with the purpose of adding details when passing from a given level to a more concrete one. According to [8], even though refinement is a key concept in MDA (Model Driven Architecture), it is loosely defined, and open to misinterpretation. The refine notion has also been defined in UML [6] as a stereotype for “Abstraction” - a directed relation from an element to another one stating that the dependent element (concrete) depends on the other one (abstract). A transition from HLC to

LLC is similar to a transformation of a PIM (platform independent model) into a PSM (platform specific model) in the context of MDA.
In this work we consider *refinement* as a transformation between HLCs and LLCs. This is done by projecting abstract correspondences on the concrete level. The principle consists in defining a HLC and then to reuse it each time it is needed at the model level. Cx=[Similarity(Entity, Table)], Cy=[Similarity(Paper, ArticleDAO)] are respectively examples of a HLC and the corresponding refined LLCs.
Thus, we define Refine (denoted by Rf) as follows: Ci Rf Cj, where Ci and Cj are two correspondences, iff Ci is defined from Cj which means that Ci is an upgrading of Cj by adding details in order to precise the correspondence. Fig.6 describes some LLCs created by refining HLCs presented in Fig.5.

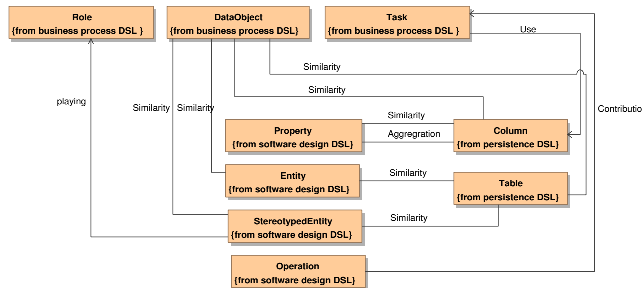


Fig. 5. Example of HLCs from CMS system

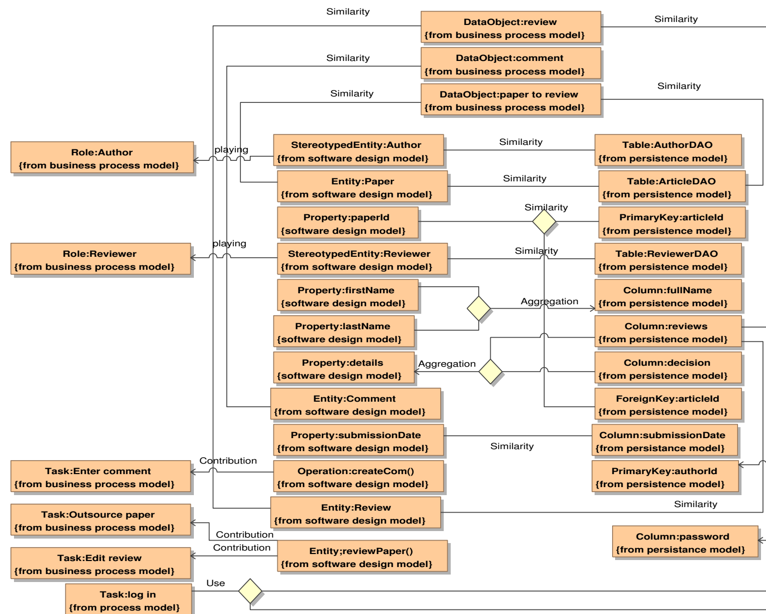


Fig. 6. Example of LLCs from CMS system

5 Ontology based approach

Ontologies are an explicit, formal specification of a shared conceptualization of a domain of interest [19]. They provide a model of the concepts of a domain and how these concepts are related to each other. In our proposal, we first automatically transform a set of Ecore (meta-)models into OWL ontologies and then we exploit ontology matching strategies in order to filter out potential incorrect LLCs.

The transformation process is carried out with the help of TwoUse (Transforming and Weaving Ontologies and UML in Software Engineering)[5]. In this process, each meta-class is represented as an OWL concept and the meta-properties are represented as OWL object properties (relating two objects) or OWL data properties (relating an object to a data value). We refer to these elements as terminological components (TBox) of ontologies. For the models, the instances of meta-models are represented as OWL individuals. Each individual is related to a type (meta-class) and is described by a set of OWL object and data properties, which are defined at meta-model level. We refer to the instances as assertional components (ABox) of ontologies.

The correspondence model (Fig.5) is also represented as an OWL ontology (TBox), where HLCs and LLCs are represented as OWL individuals (ABox). While HLCs are manually established by the domain expert before the transformation step, LLCs will be automatically generated within our matching process.

The first step of the matching process consists in automatically generating a set of OWL LLCs from the set of OWL HLCs (refining process). From each OWL HLC relating a source and a target elements (OWL individuals), one OWL LLC is automatically created. The second step consists in automatically filtering out potential incorrect correspondences from the initial set of OWL LLCs, with the help of ontology matching strategies. For each source and target elements (“RefElement” in Fig.6) of a LLC, we retrieve the value of the data property describing them (i.e., following the correspondence model, the “name” data property of a “RefElement” describes this element at model level, e.g., “conferenceId”). These values are then compared using different syntactic metrics. Correspondences that do not match using these metrics are removed from the initial set of LLCs.

In fact, MMC serves as a contextual basis for the matching process at model level, limiting the generation of a correspondence to elements whose type participates in a HLC. Even if the contextual information helps avoiding the creation of correspondences between elements of types that do not match (e.g., an “Operation” and a “Property”), it does not guarantee the generation of correct correspondences. Although many different ontology matching approaches have been proposed in the literature [11][12], and most of them exploit the Tbox level of ontologies. Here, we work essentially at the ABox level, adopting syntactic strategies for comparing data properties of OWL individuals. This first choice is due to the fact that we are working in a domain-specific set of ontologies where the terms used to describe the elements at model level through different models is supposed to be similar. Finally, external resources as WordNet do not cover the vocabulary of a particular domain, as the one we consider here.

6 Preliminary evaluation

We have conducted our experiments using the CMS case study. In this first series of experiments, we focused on the "similarity" relationship. The correspondences at meta-model level (HLC) were manually established by a domain expert. Manually created correspondences at model level (LLC) were used as reference alignments to evaluate our approach. As evaluation metrics, we applied well known metrics such as precision (the ratio of correctly found correspondences over the total number of returned correspondences), recall (the ratio of correctly found correspondences over the total number of expected correspondences, i.e., number of correspondences in the reference alignment) and f-measure (the harmonic mean of precision and recall). As depicted in Fig. 5, the alignment at meta-model level is composed of 7 HLCs involving the similarity relation. From these HLCs, through a projection process, for each pair of models (sofwareDesign-persistence, BusinessProcess-SoftwareDesign, and BusinessProcess-Persistence), we automatically generated the set of all possible LLCs (i.e., a LLC is generated between elements whose types are involved in a HLC). We refer to this set as "projection" in the following. In order to filter out incorrect correspondences from this projection set, we syntactically compare the values of the data properties of each ontology individual using equality of strings and Levenshtein edit distance [17]. This metric measures the effort to transform a string into another. We have also run LogMap [18], a publicly available ontology matching system that is able to deal with the ABox level of ontologies. LogMap is based on reasoning and inconsistency repair techniques.

Table 1 presents the results for the 4 sets of correspondences ("projection", "equality", "edit distance", "LogMap"), which have been generated independently of each other. For SofwareDesign-Persistence (SD-PS), the projection set contains 462 correspondences. As expected, the precision for this set is very low (many incorrect correspondences have been generated), while recall is high (many correct correspondences were retrieved). An expected behavior has also been observed for the equality set, where precision is quite high and recall is low. Using the Levenshtein distance, we obtain intermediary values of precision and recall (and the best F-Measure). For LogMap, although having good values of precision, it was not possible to deal with the very specific way individuals are represented in our models. For BusinessProcess-SoftwareDesign (BP-SD), although projection still generates the best recall, equality performed better than edit distance, especially in terms of precision.

	Projection			Equality			Edit distance			LogMap		
	P	F	R	P	F	R	P	F	R	P	F	R
SD-PS	0.01	0.02	0.63	0.50	0.26	0.18	0.26	0.33	0.45	0.28	0.22	0.18
BP-SD	0.06	0.12	1.00	0.40	0.50	0.66	0.18	0.28	0.66	0.00	0.00	0.00
BP-PS	0.01	0.03	1.00	0.00	0.00	0.00	0.11	0.18	0.50	0.00	0.00	0.00

Table 1. Precision (P), Recall (R) and F-measure (F) for the sets of correspondences generated

It is due to the fact that most correspondences in the reference alignment are characterized by an equality between their data property values. Log-Map was not able to found any correspondence for this pair (we do not distinguish empty alignments from

wrong ones – these alignments are indicated with zeros in Tables 1 and 2). Finally, for BusinessProcess-Persistence (BP-PS), an opposite behavior was observed, where edit distance outperforms equality. This is due to the fact that correspondences in the reference alignment do not involve equality and very few ones that can be detected by the metrics we apply in this paper (e.g., "paper to review" and "ArticleDAO").
When considering the intersection between the sets (Table 2), i.e., combining contextual information from projection and a syntactic metric on data property values, for SofwareDesign-Persistence (SD-PS), the best values where obtained when combining projection and edit distance. For BusinessProcess-SoftwareDesign (BP-SD) and BusinessProcess-Persistence (BP-PS), a similar behavior was observed. Although our initial approach is naive, we obtain better results when combining projection and syntactic metrics. Our process could also include a pre-processing step in order to treat compound terms (e.g., “Article DAO” instead of “ArticleDAO”). Furthermore, the use of other syntactic metrics [20] is envisaged.

	Projection+equality			Projection+edit			Projection+LogMap		
	P	F	R	P	F	R	P	F	R
SD-PS	0.50	0.26	0.18	0.65	0.52	0.45	0.50	0.26	0.18
BP-SD	1.00	0.80	0.66	0.66	0.66	0.66	0.00	0.00	0.00
BP-PS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 2. Precision, Recall and F-measure when combining projection and syntactic measures

7 Related Works

Model matching. Several research works related to models matching have been discussed in a previous work [13]. Among identified shortcomings of existing approaches, one can note that they manage only binary correspondences and therefore cannot establish n-ary ones that relate a model element to a set of elements belonging to other models. In addition, studied approaches lead to produce a correspondence model between each pair of input models whereas we build a unique model of correspondences among input models. Several approaches consist in transforming the input (meta-)models to other formats in order to apply their matching techniques. For example, MatchBox [7] transforms input meta-models into a tree model called AMC (Auto Mapping Core). The process continues by applying a set of matching strategies to produce the model of correspondences. Another approach, described in [10], consists in transforming the meta-models to directed labeled graphs. The correspondences are then obtained by applying the Similarity Flooding algorithm. We could not use those approaches in our work because they propose to apply matching techniques only at meta-model level and they do not consider the model level. Furthermore they are not appropriate with meta-models which have a big semantic gap between them because in this case, according to [2], this results in a very low precision and in a very poor recall (mostly below 0.10). The result is satisfactory only when it comes to matching techniques between a former meta-model and a new version such as UML 1.4 and UML 2.4 as they share the same properties.

Ontology matching. Many approaches to the matching problem have been proposed in literature since the last decade [14][11][12]. These approaches can be classified along the many features that can be found in ontologies (labels, structures, instances, semantics), or with regard to the kind of disciplines they belong to (e.g., statistics, combinatorial, semantics, linguistics, machine learning, or data analysis). While terminological and syntactic methods lexically compare strings (tokens or n-grams) used in naming entities (or in the labels and comments concerning entities), semantic methods utilize model-theoretic semantics to determine whether or not a correspondence exists between two entities. Approaches may consider the internal ontological structure, such as the range of their properties (attributes and relations), their cardinality, and the transitivity and/or symmetry of their properties, or alternatively the external ontological structure, such as the position of the two entities within the ontological hierarchy. The instances (or extensions) of classes could also be compared using extension-based approaches. In addition, ontology matching systems rely not on a single approach. Finally, using domain ontologies [15] and different knowledge sources [16] as background knowledge within the matching process has been exploited in the literature. One initiative for evaluating ontology matching approaches for model matching was proposed in the context of the Ontology Alignment Evaluation Initiative (OAEI) campaigns. The aim was to compare classical model matching approaches and ontology-based approaches. Each test case consisted of a pair of Ecore meta-models, a pair of OWL ontologies, and a reference alignment. However, no system participating in that campaign was able to deal with this data set, showing that there is room left for exploiting this kind of approach.

8 Final remarks and Future Work

Our general goal is to related heterogeneous models corresponding to different points of view on a complex system. In this paper, we have put the emphasis on the use of ontology matching strategies to automate the production of correspondences at model level. Based on a common set of correspondences at meta-model level, a set of correspondences at model level is generated, and then refined with the help of syntactic matching approaches. Although we focus on the similarity relationship, as most of existing approaches in model matching, our work is a gateway that will enable us to exploit other kinds of relationships. The novelty with respect to other approaches is the use of correspondences at meta-model level, which serve as a context to produce correspondences at model level.

As future work, we have identified several directions to improve the approach proposed in this paper. First, we intend to exploit other relationships than similarity (i.e., dependency, aggregation), what is marginally treated in both models and ontology matching areas. Secondly, we plan to use matching approaches, in particular those based on background knowledge (e.g., using domain ontologies). This approach can be interesting in order to find the non-trivial relations between models. Third, we plan to apply other syntactic metrics and combine them with semantic ones.

References

1. Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel matching for automatic model transformation generation. In *Model Driven Engineering Languages and Systems*. Springer, pages 326–340, 2008.
2. Gerti Kappel, Horst Kargl, Gerhard Kramler, Andrea Schauerhuber, Martina Seidl, Michael Strommer, and Manuel Wimmer. Matching meta-models with semantic systems-an experience report. In *BTW Workshops*, 38–52, 2007.
3. Mahmoud El Hamlaoui, Sophie Ebersold, Adil Anwar, Bernard Coulette, Mahmoud Nassar. Heterogeneous models matching for consistency management. Proc. IEEE RCIS, 1-12, Marrakech, Morocco, 2014.
4. Nenad Medvidovic and Richard Taylor. A classification and comparison framework for software architecture description languages. Software Engineering, IEEE Transactions on, 26(1):70–93, 2000.
5. Fernando Silva Parreiras, Steffen Staab, and Andreas Winter. TwoUse: Integrating UML models and OWL ontologies. Technical Report 16/2007. Tech. Inst. für Informatik, 2007.
6. OMG UML. Uml 2.0: Superstructure specification. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/, November 2007.
7. Konrad Voigt, Petko Ivanov, and Andreas Rummler. Matchbox: combined meta-model matching for semi-automatic mapping generation. SAC ’10, ACM, pages 2281–2288, New York, NY, USA, 2010.
8. Dennis Wagelaar. Context-driven model refinement. In Proceedings of the 2003 European Conference MDAFA’03, pages 189–203, Berlin, Heidelberg, 2005. Springer-Verlag.
9. Wayne Wolf. Hardware-software co-design of embedded systems [and prolog]. Proceedings of the IEEE, 82(7):967–989, 1994.
10. Michael Wooldridge. Agent-based software engineering. Software Engineering. IEEE Proceedings- [see also Software, IEEE Proceedings], 144(1):26–37, 1997.
11. Rahm Erhard, Bernstein Philip. A survey of approaches to automatic schema matching. The VLDB Journal. Edited by P. Scheuermann. Springer-Verlag, 334-350, 2001.
12. Euzenat Jerôme, Shvaiko Pavel. Ontology matching. Springer, 2nd edition, 2013.
13. Mahmoud El Hamlaoui, Sophie Ebersold, Adil Anwar, Bernard Coulette, Mahmoud Nassar. Towards a framework for heterogeneous models matching. Journal of Software Engineering, Science Alert, 132-151, New York, USA, 2014.
14. Yannis Kalfoglou and Marco Schorlemmer. Ontology mapping: the state of the art. The Knowledge Engineering Review Journal (KER). Volume 18, 1-31, 2003.
15. Zharko Aleksovski, Warner ten Kate and Frank van Harmelen. Using multiple ontologies as background knowledge in ontology matching. CISWeb Workshop, 35-49. 2008.
16. Marta Sabou, Mathieu d’Aquin, Enrico Motta. Exploring the Semantic Web as Background Knowledge for Ontology Matching. Journal on Data Semantics XI. Volume 5383, 156-190, 2008.
17. Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. English Translation in Soviet Physics Doklady, 10(8) p. 707-710, 1966.
18. Ernesto Jimenez Ruiz, Bernardo Grau, Yujiao Zhou and Ian Horrocks. Large-scale Interactive Ontology Matching: Algorithms and Implementation. ECAI, 444-449, 2012.
19. Thomas Gruber. Toward principles for the design of ontologies used for knowledge sharing, International Journal of Human-Computer Studies, 43(5-6), 907–928, 1995.
20. Michelle Cheatham and Pascal Hitzler. String Similarity Metrics for Ontology Alignment. In Proceedings of the International Semantic Web Conference. Springer, 294-309, 2013.
21. Martin Fowler. Domain-Specific Languages. Addison-Wesley, 2010.

Supporting Debugging in a Heterogeneous, Globally Distributed Environment

Jonathan Corley and Jeff Gray

The University of Alabama, U.S.A.
corle001@ua.edu, gray@cs.ua.edu

Abstract. Model-Driven Engineering has emerged as a software development paradigm that can assist in separating the issues of the problem space of a software system from the complexities of implementation in the solution space. As software systems have become more complex, a need for multiple abstractions to describe a single system has emerged. The development teams of these massive systems are also often geographically distributed. These emerging concerns for MDE systems have led to a need for a heterogeneous, and potentially globally distributed, modeling environment. As these modeling environments are being explored, new challenges are being uncovered. In this paper, we discuss the need for debugging support in heterogeneous, globally distributed modeling systems and identify a number of challenges related to debugging that must be overcome to support this evolving paradigm for software development.

1 Introduction

Model-Driven Engineering (MDE) has emerged as a software development paradigm that can assist in separating the issues of the problem space of a software system from the accidental complexities of implementation in the solution space. MDE approaches often use customized domain-specific modeling languages (DSMLs) that capture the intent of a particular group of users through abstractions and notations that fit a specific domain of interest. Through the application of DSMLs, various stakeholders in a project are enabled to view and edit the system using an abstraction most appropriate to their needs and expertise. However, the disparate abstractions introduced can create barriers between components in the same project by separating these concerns into distinct DSMLs without the ability to describe interactions between components [1]. An electrical engineer may produce a wiring diagram while a software engineer produces a component diagram. If the two engineers are both working on a shared project then the implementation of these designs will impact each other. This scenario indicates a need for shared reasoning, analysis, and communication between these two groups to enhance the cohesiveness of the final design and resulting implementation.

As the development of support for heterogeneous, globally distributed modeling environments progresses, a key concern is what support is expected in these new environments. Debugging is a common task that all software developers encounter across different software artifacts [2]. Though debugging has been a consistent aspect of the software development process, debugging tool support has changed little over the past

half century [2]. Several novel approaches to debugging have been introduced in the research literature, such as omniscient debugging [3] and query-based debugging [4,5,6], each claiming to improve efficiency and effectiveness of developers. However, commercial tool support available to programmers focuses primarily on stepwise execution of code, typically with break-points [5]. MDE tools are typically less mature than tools available for traditional general-purpose programming languages (GPLs). With respect to MDE research, Mannadier and Vangheluwe observed there has been little concern for the state of debugging support [7]. However, over a decade ago, Bran Selic commented that if developers are not satisfied with the "day-to-day" application of MDE, then MDE will be rejected by developers [8]. One of the most common tasks undertaken by software developers is debugging. Therefore, we believe improved debugging support will aid adoption of MDE, especially for complex systems described by multiple DSMLs.

2 Background and Related Work

Like all software systems, evolution also occurs in software models. In MDE, the evolution of models is commonly defined using model transformation languages (MTLs), which can be used to specify the distinct needs of a requirements or engineering change at the software modeling level. Model transformations are also a type of software abstraction that can be subject to human error. Thus, traditional approaches to bug localization may also be applied to assist in locating errors in model transformations.

2.1 Stepwise Execution with Breakpoints

The most commonly implemented debugging approach is stepwise execution, which enables the developer to observe hidden state information dynamically during execution. A stepwise execution environment generally possesses the following features: play, pause, stop, and step. Play allows for continuous execution; pause suspends execution at the current state; stop terminates execution. Most tools support three types of step (*i.e., stepOver*, *stepIn*, and *stepOut*) enabling developers to incrementally progress the execution. Numerous MDE tools (including TROPIC [4], GReAT [9], ATL [10], and more) provide support for stepwise execution.

2.2 Query-based Debugging

Query-based debugging (QBD) promotes the use of queries to aid a developer in locating the source of a bug. These queries aid a developer in gaining a better understanding of the underlying system. Most QBD techniques have promoted the use of a rich, expressive query language [4,6]. However, some approaches focus on a smaller range of query options. The Whyline [5] focuses on queries designed to lead from a point of error to the fault that generated the observable error. These queries are termed "why did" queries (*e.g.,* "why did attribute x have value y?") [5]. QBD has been empirically evaluated and demonstrated to improve developer efficiency and effectiveness during debugging tasks [5,6].

We are only aware of one MDE tool, TROPIC [4], that supports query-based debugging. TROPIC provides an interactive query console that enables developers to specify OCL queries. TROPIC supports debugging by converting transformations to a Transformation Net (TN), a specialized colored Petri-net. TROPIC supports many modeling languages, but converts all models to a common TN representation. TROPIC is intended for a single developer and does not support collaborative distributed development.

2.3 Omniscient Debugging

We are not aware of any MDE tool supporting omniscient debugging, which can be considered an extension of stepwise execution that provides the ability to traverse back through the execution of a debugging session dynamically at run-time. Current work in the area of omniscient debugging is focused on GPLs. However, the technique would also be beneficial in the MDE context. MTs are a software abstraction subject to error similar to GPLs [3]. An error may manifest at a point later in execution than the source of the defect, the fault. A concern common to both an MDE and GPL context is the time and effort required to reach a portion of the system's execution that exercises the defect. The developer may target the location of an error and thereby miss the location of the fault. In a traditional stepwise execution environment, the developer would need to restart the system and target a new location. This process of restarting to inspect new target locations may even be repeated multiple times. Restarting may require a nontrivial amount of time to reach the desired location, and may require significant manual input from the developer. Omniscient debugging enables full traversal of the execution history thereby eliminating this concern.

3 Challenges and Potential for Global Debugging Tool Support

Though the various debugging techniques can be applied to an MDE context, the application of these techniques to a globally distributed, heterogeneous modeling system brings new challenges that must be overcome. In this section, we discuss challenges unique to this paradigm.

3.1 Supporting an Extensible Debugging Environment

In a heterogeneous modeling environment, the underlying system natively supports a variety of DSMLs. This requirement is not a consideration in GPL design where a single language is supported. The more versatile MDE system must be able to represent information in the most appropriate formalism. However, the developers of these systems cannot always anticipate the unique concerns and features of future developers. The issue of supporting debugging for the variety of DSMLs available encounters similar concerns. To address these concerns, a heterogeneous modeling system must be extensible, and may require developers to provide the debugging support appropriate to their formalism with no support from the underlying tool. However, this creates a scenario where future developers will often duplicate effort for common concerns. An alternative is to provide an extensible base of support that future developers may extend.

This base of support should handle simple concerns including the ability to manually control execution of the system, visualize and explore state information, and collect state information as the system progresses. Providing these three basic features enables stepwise execution, omniscient debugging, and query-based debugging. An extension to these that would enrich debugging tool support is the ability to reload transformations and alter model elements at runtime. This extension would enable a developer to freely explore and modify the system in order to identify faults and test alterations.

A heterogeneous modeling environment may also enable the execution of multiple MTLs providing a range of potential features. Consider a system that includes a distinct MTL for both inplace and outplace transformations. These scenarios each create a unique concern for debugging support. The inplace transformation displays a single model (or set of models), but the outplace transformation must provide facilities for identifying input and output models. If providing traceability features, the inplace transformation would link elements from a previous version of the same model(s), whereas the outplace transformation would link elements from input model(s) to output model(s). These differences can lead to a varied implementation of the same concerns, but a heterogeneous environment may need to consider either or both. Such an environment must also enable defining the basic characteristics of stepwise execution (*e.g.,* what constitutes a step or scope). A step in a GPL is a statement. However, MTLs are typically defined by rules. A rule may be a simple graph transformation or a more complex component that can contain other rules (*e.g.,* ATL pre and post conditions [10]). The definition of step may therefore occur at various levels of granularity. Future designers should be able to define precise semantics for this concern that match most closely with the intended MTL environment. Similarly, defining what constitutes a scope is vital to a stepwise execution environment.

3.2 Supporting Many Formalisms in a Consistent Debugging Interface

Query-based debugging (QBD) is a promising debugging technique that provides developers with facilities to ask questions directly to the system. Query languages are typically strongly coupled with the target language. The target language may have concepts of classes and inheritance or functions and return types. However, in a heterogeneous modeling environment the target language is not fixed. The system allows (and encourages) developers to use a wide array of DSMLs to facilitate the goal of using the most appropriate abstraction. This design results in a system that may or may not possess a vast and varying set of concepts and structures. Applying QBD in a heterogeneous environment must therefore provide some design to handle the variability of DSMLs without the designers of the system making any assumptions regarding the specific terminology and structures available to future developers. This concern is further exacerbated by the inclusion of graphical query languages.

A modeling system is not always designed with the intention to limit the system to a single view. The system may include multiple metamodels and even multiple concrete syntax for the same metamodel. For example, a vehicle may contain many varied subsystems such as brakes, steering, power windows, blinkers, and many more. A vehicle is also a single unit where many subsystems have a direct impact on others. A prime example is how electrical wiring directly impacts the functioning of the power window

subsystem. If the power windows exceed the capacity for the electrical wiring system then the windows will fail to open and close properly. However, these two systems may be designed using different DSMLs and in a typical modeling environment would be in separate models. However, a heterogeneous modeling environment would enable developers to view these systems either together or separately as needed. This leads to further concerns when applying QBD. If the developer of the car system were to pose a query such as “why did the power window not rise?” the system may need to search through models defined using several DSMLs to provide the answer. However, current work in the area of QBD has always assumed a single language and there is no existing technique that is concerned with searching across multiple languages.

A primary concern for omniscient debuggers is the collection of trace information required to revert the system to previous states of execution. This collection of trace information forms a history of execution for the system. In a heterogeneous modeling environment, the collection of trace information is complicated by the varying structures. An omniscient technique must collect the smallest units of information for each modification in order to minimize the space consumption of the history structure. However, an omniscient debugger must also collect information relevant to the structure of the model in order to ensure proper application of any change. For example, assume a model element ‘a’ relies on model element ‘b’ and both are deleted. When reverting the delete operation, the system must ensure there is never a state where ‘a’ exists without ‘b’ to avoid violating constraints of the modeling environment. Similarly, if element ‘a’ is always altered to match any modification in ‘b’, the underlying execution environment may capture these modifications independently, but upon replaying these modifications may incur an additional modification to element ‘a’ (both when ‘b’ is reverted causing ‘a’ to be automatically altered and when ‘a’ is directly altered to revert the recorded modification). However, these structural concerns may vary depending on the specific formalisms used.

3.3 Supporting Debugging in a Globally Distributed Environment

A global software engineering system is concerned with geographically distributed development teams. A natural implication of this environment is that developers will utilize separate physical machines even while actively collaborating. However, the implications of separate machines accessing a common environment can be more subtle. The typical debugging environment includes a single machine and therefore assumes a single point of control for the debugging environment. The introduction of more points of access to control the environment creates a more complex scenario. Consider the following scenario: James is inspecting the state of execution at a specific point, and Elizabeth (unaware of James’s intent) progresses the transformation to a state she is interested in investigating. In this scenario, James and Elizabeth are each intent on exploring distinct points in history. A simple solution is to provide facilities for James to express his intent to explore the current state and even possibly lock the system to the current state. However, this solution restricts Elizabeth from following her own thread of investigation. A global omniscient debugging system could also offer facilities for both parties to independently explore the system while simultaneously enabling the two developers to share an environment. Thus, Elizabeth and James could work both

collaboratively on a single issue and in parallel on separate related issues within the same environment.

4 Conclusion and Future Work

We have discussed the current state of debugging for MDE. We then summarized the challenges presented when applying these techniques in a globally distributed, heterogeneous modeling environment. Several challenges were discussed for each technique. Some challenges were caused by the application of many formalisms to describe a single system, contrasting with the existing work that typically focuses on a single formalism. Other challenges were the result of applying these techniques to a distributed environment. In these environments, the system must support both collaborative processes where models and control is shared for a common goal, and parallel processes where both models are shared and control is independent simultaneously. These concerns were presented and discussed, but left as future work to solve. We believe debugging support is a requirement in a modern software development environment and these issues are therefore of vital concern for the future of this growing and evolving paradigm.

References

1. Combemale, B., Deantoni, J., Baudry, B., France, R., Jézéquel, J.M., Gray, J.: Globalizing Modeling Languages. Computer (June 2014) 10–13
2. Mirko Seifert, Stefan Katscher: Debugging triple graph grammar-based model transformations. In: Proceedings of 6th International Fujaba Days, Dresden, Germany (2008)
3. Adrian Lienhard, Julien Fierz, Oscar Nierstrasz: Flow-centric, back-in-time debugging. In: Proc. of Objects, Components, Models and Patterns, Zurich, Switzerland (2009) 272–288
4. Johannes Schoenboeck, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Wieland Schwinger, Manuel Wimmer: Catch me if you can – debugging support for model transformations. In: Proc. of 12th Int'l Conf. on Model-Driven Engineering, Languages, and Systems, Denver, CO, USA (2009) 5–20
5. Andrew J. Ko, Brad Myers: Debugging Reinvented: Asking and answering why and why not questions about program behavior. In: Proceedings of the 30th International Conference on Software Engineering (ICSE '08), Leipzig, Germany (2008) 301–310
6. Lencevicius, R., Hölzle, U., Singh, A.: Dynamic Query-Based Debugging of Object-Oriented Programs. Automated Software Engineering **10** (2003) 39–74
7. Mannadiar, R., Vangheluwe, H.: Debugging in domain-specific modelling. In: Software Language Engineering. Volume 6563 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 276–285
8. Selic, B.: The pragmatics of model-driven development. IEEE Software **20**(5) (2003) 19–25
9. Agrawal, A.: Graph rewriting and transformation (great): a solution for the model integrated computing (mic) bottleneck. In: Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on. (Oct 2003) 364–368
10. Jouault, F., Kurtev, I.: Transforming models with atl. In: Satellite Events at the MoDELS 2005 Conference. Volume 3844 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2006) 128–138

Understanding Metalanguage Integration by Renarrating a Technical Space Megamodel

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. There are many software languages which are not exposed protocols, exchange formats, interfaces and storage formats, and are only used for intermediate representation, runtime data manipulation and tool-specific serialisation. Yet, they can be important for technology comprehension, since such internal implementation details may have indirect impact on some aspects of the externally observed behaviour of the system. In this paper, we show a concrete example of how various tools and their technological differences can be explained based on one abstract megamodel and its different renarrations.

1 Introduction

Megamodels are used for modelling complex systems involving many artefacts, each of which is also in turn a model or a transformation [3,4]. For instance, they can help represent an entire technological space or a technical space in order to expose its components and to explain them to previously unaware audience (such as students) [6,10]. The main focus of megamodelling is usually on externally observable (meta)languages: communication protocols, data interchange formats, application programming interfaces, algebraic data types, public library interfaces, serialisation formats, etc. Yet there are a lot of (meta)language used behind the scenes for internal presentation of data structures — and we all know very well how much of an impact can a different data structure have on performance of an algorithmically nontrivial application. As it turns out, megamodelling can be very helpful here as well.

Megamodels (also called linguistic architecture models [6,10], macromodels [15], technology models, etc) come in a great variety of forms and approaches and are theoretically useful for solving many problems of different stakeholders. However, one of the main showstoppers is their overwhelming complexity: not only a typical megamodel requires considerable investment in deep domain analysis, exploratory experimentation, modelling and metamodelling; but also the result thereof is a towering monolith easily intimidating any possible users. At the same time, simplification is possible yet often undesirable, for the devil lurks in the details. One of the existing solutions is investing in packaging the megamodel as well as in its development. We can slice the megamodel into digestible parts and navigate stakeholders through them, possibly through various itineraries depending on the priorities — this process is referred to as *megamodel renarration* [18].

A renarration of a megamodel is a story that traverses the elements of this megamodel in order to guide the users through it and to gradually introduce them to the model elements and thus to domain concepts. Formally, a renarration relies on operators for addition/removal, restriction/generalisation, zoom in/zoom out, instantiation/parametrisation, connection/disconnection and can make use of backtracking [11]. In prior work we have shown renarrations as annotated megamodel transformations, but have not used them in multiple scenarios based on one original model.

The approach we propose in this paper involves investing in a global megamodel of an entire technical space, and then using renarrations of it to demonstrate each existing technology. Thus, the contribution of the paper is mainly the focus on using one baseline white box megamodel for establishing a common ground for explaining various subtly different technologies of the same domain by renarrating it repeatedly.

Specifically in the context of the GEMOC initiative, megamodelling addresses the second focus (integration of heterogeneous model elements), while renarration treats the first issue (catering various stakeholder concerns). So far this material has been (in a more volatile form) used in teaching courses on software language engineering [2], software evolution, software construction and in supervision of Master students.

2 Parsing with many faces

For a demonstration of the proposed approach let us consider a megamodel for parsing in the broad sense that we presented in earlier work [21]. The model includes twelve kinds of artefacts commonly found in software language enginering (as well as commonly encountered mappings among them, see Figure 1):

- ◊ Str — a string, a file, a byte stream.
- ◊ Tok — a finite sequence of strings (called *tokens*) which, when concatenated, yields Str. Includes spaces, line breaks, comments, etc — collectively, *layout*.
- ◊ TTk — a finite sequence of typed tokens, possibly with layout removed, some classified as numbers, strings, etc.
- ◊ Lex — a lexical source model [19] that addes grouping to typing; in fact a possibly incomplete tree connecting most tokens together in one structure.
- ◊ For — a forest of parse trees, a parse graph or an ambiguous parse tree with sharing; a tree-like structure that models Str according to a syntactic definition.
- ◊ Ptr — an unambiguous parse tree where the leaves can be concatenated to form Str.
- ◊ Cst — a parse tree with concrete syntax information. Structurally similar to Ptr, but abstracted from layout and other minor details. Comments could still be a part of the Cst model, depending on the use case.
- ◊ Ast — a tree which contains only abstract syntax information.
- ◊ Pic — a picture, which can be an ad hoc model, a “natural model” or a rendering of a formal model.

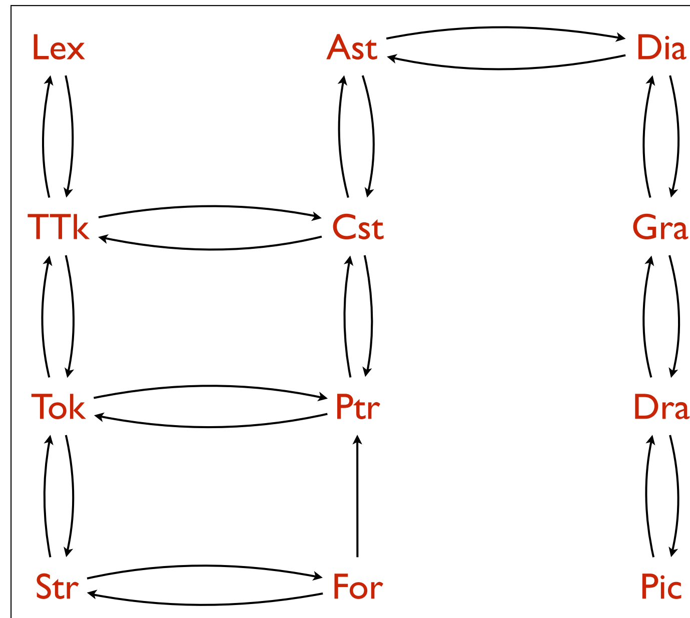
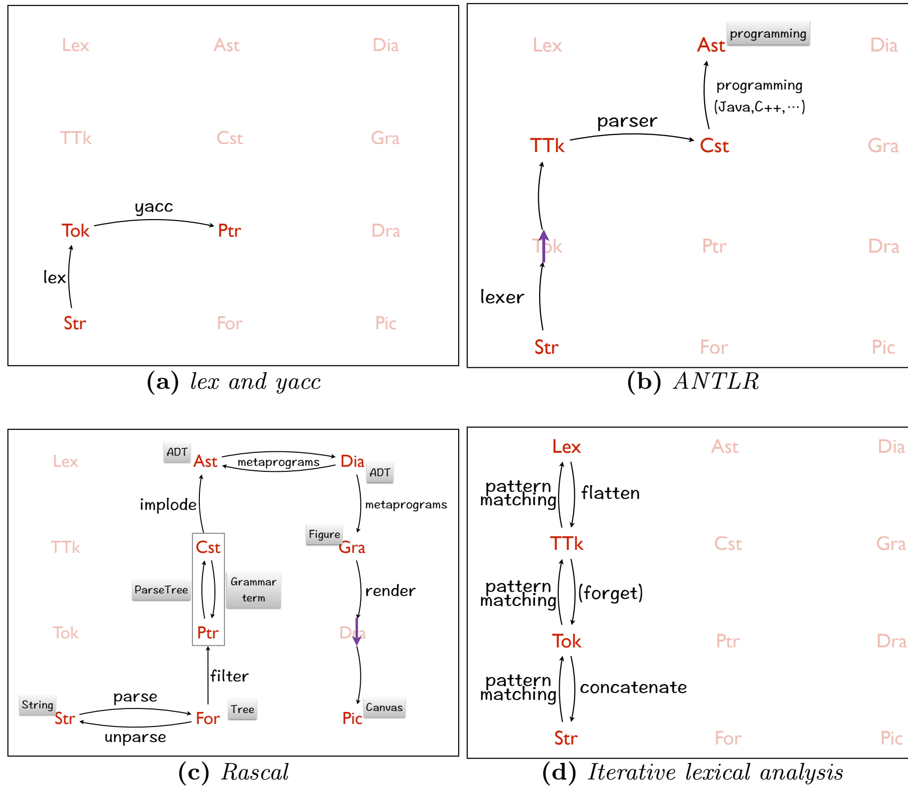


Fig. 1. A megamodel of parsing in a broad sense — see [21] for detailed definitions and descriptions of these kinds of software artefacts and mappings.

- ◊ Dra — a graphical representation of a model (not necessarily a tree), a drawing in the sense of GraphML or SVG, or a metamodel-indepenent syntax but metametamodel-specific syntax like OMG HUTN.
- ◊ Gra — an entity-relationship graph, a categorical diagram or any other primitive "boxes and arrows" level model.
- ◊ Dia — a diagram, a graphical model in the sense of EMF or UML, a model with an explicit advanced metamodel.

The megamodel from Figure 1 provides a unique uniform view on parsing, unparsing, formatting, pretty-printing, disambiguation, visualisation and related activities — it is a big step from heterogeneous discordant papers originating from relevant technical spaces toward general understanding of the field. Yet, as we have claimed before [18,11], a monolithic megamodel can play a role of a knowledge container, but cannot be used directly as the deployed artefact. (As a side remark, this corresponds to the claim by Bézivin et al that a megamodel as a model of models should not be used as a reference model [3]). Hence, we need a renarration of a megamodel to successfully deliver the knowledge behind it. A renarration can happen naturally (e.g., as a lecture for students) or be formally inferred with megamodel transformation operators for addition, connection, instantiation, etc [11].



(a) *lex and yacc* **(b)** *ANTLR*
(c) *Rascal* **(d)** *Iterative lexical analysis*

Fig. 2. Four illustrated renarrations of the (slices of the) megamodel from Figure 1

In this paper, we use English for the narrative, and the models themselves are available at ReMoDD: `http://www.cs.colostate.edu/remodd/v1/content/renarrating-metalanguage-integration`. In the following sections, we demonstrate several renarrations of the megamodel from Figure 1.

2.1 Parsing in a narrow sense: lex + yacc

One of the textbook approaches to parsing is using two tools to obtain a parse tree from the input string: one for lexical analysis and one for syntactic analysis. In many classic compiler construction courses lexical analysis is done with `lex` [12] or one of its successors. The tokens that are obtained by lexical analysis, are in fact typed, but the type information is not necessarily used for anything, so we can model the result of the lexical analysis with Tok. The next step is handled by a compiler compiler like `yacc` [7] or its more modern counterparts (but not too innovative — we want to stick to the classic DragonBook-like view [1]). This syntax analysis tool consumes Tok and produces a parse tree — Ptr. This can be seen on a rather simple Figure 2(**a**).

2.2 Advanced parsing technology: ANTLR

Consider ANTLR [14], a state of the art compiler compiler that can be used for the same purpose as lex+yacc, but incorporates the results of several decades of research on parsing, compiler construction and interactive programming environments. Both a lexer and a parser are generated from the uniform syntactic definition (grammar). Lexical nonterminals, usually written in CAPSLOCK, define a grammar used for lexical analysis. Most of them are preterminals — their definitions contain only terminals, combined sequentially, with disjunction, Kleene closure and other combinators typical for regular expressions [8]. As shown on Figure 2**(b)**, the output of the lexer is TTk, a stream of strongly typed tokens — each token has to either belong to one of the lexical categories (be parsed as a lexical nonterminal) or match one of the terminal symbols used in the rest of the grammar — they are turned into preterminals automatically by ANTLR. The untyped version of the same representation (Tok) is not available directly: if needed, one could possibly either disregard the typing information (e.g., by using code duplicates inside semantic actions) or plug in into the internals of the generated lexer.

A typed token stream is processed by a parser which ANTLR generates from the input grammar. The result is Cst, a parse tree that abstracts from some details like layout and comments. It is important to note that ANTLR generates the definition of the Cst and provides means to traverse them. However, if one still desires to use an abstract syntax tree, both Ast itself and the mapping from Cst to Ast need to be programmed explicitly in the base language of ANTLR (Java, C++, C#, Python, etc). The mapping can be scattered among the nonterminal definitions directly in the grammar (as semantic actions), or it can be written as a separate program that traverses the ANTLR Cst with the ANTLR visitor and constructs a specific Ast. The class structure of the Ast itself always needs to be defined and processed independently from ANTLR.

2.3 Rascal metaprogramming language

Rascal [9] is another state of the art piece of grammarware — however, an important difference from ANTLR is that Rascal is advertised as a "one-stop-shop" for software analysis, transformation and visualisation. Let us try to understand this difference from Figure 2**(c)**.

Rascal uses generalised parsing (more specifically, GLL), which yields a parse forest instead of a parse tree, if the grammar is ambiguous. Such parse forests (For) are represented internally with the same structure — a term representation that is allowed to explicitly contain ambiguity node. Thus, in order to decide if a given tree is For or Ptr, we need to perform a deep match on an `amb(_,_)` pattern (since pattern matching is one of the basic constructions in Rascal, this operation is trivial to express, even though it might become a performance bottleneck).

By Rascal design, there is no observable distinction between Ptr and Cst. All trees are stored internally as Ptr, but all pattern matching behaves as if both the pattern and term is Cst (with the pattern allowed to be incomplete). Each

unambiguous tree conforms to the grammar (a syntax specification) that was used to parse it. A grammar is defined in Rascal within the same module or imported as a separate one. Relying on such grammatical structure can simplify pattern matching immensely: instead of checking for a term which is an application of a particular production rules with certain arguments, we write the same intent down with a term on the left hand side, typed to a particular nonterminal and thus fully conforming to its structure (modulo intended gaps to be skipped during unification).

A Cst can be mapped to Ast explicitly by writing a pattern-matching visitor, which is done in some cases that require sophisticated compulations as a part of the mapping. However, an easier way is to use an `implode()` library function that has a set of stable heuristic rules for finding bidirectional correspondence between a given syntax definition and a given algebraic data type. The ADT itself (the structure of Ast) must still be programmed manually, which is traditionally not considered to be a burden since one wants to have full control about the way abstract syntax is defined. (When this is not the case, it can be inferred from the grammar by grammar mutations [20] of GrammarLab, a Rascal library for manipulating grammars in a broad sense[1]). `implode()` is not shipped with a reverse function, so any derivation from Ast to Cst/Ptr, if needed, must be programmed manually.

High level abstract diagrams (Dia) are also modelled in Rascal by algebraic data types managed by the (meta)programmer. A universal yet still a high level visual model (Gra) is provided in the standard Rascal library and contains elements like boxes, grids, graphs, trees, plots. A `render()` function, however, positions all these elements automatically and only outputs the final picture (Pic) on screen or to a file, effectively skipping over Dra — for a Rascal programmer it means having no control over the exact positioning of most elements on canvas, except for general constraints which are a part of the metamodel of Gra.

2.4 Semiparsing: building lexical models with ILA

Semiparsing [19] is an umbrella term for techniques of imprecise manipulation of source code (its variations are known as agile modeling, robust parsing, lightweight processing, error repair, etc). They are inherently very different because usually come into existence for solving a very particular practical problem — we have claimed recently that Boolean grammars [13] and parsing schemata [16] can be helpful in modelling all possible variations of semiparsing. However, as useful as these two formalisations could be in deep understanding of the methods, relating them and positioning among themselves, they are not always as effective for their implementation-driven comprehension, especially by software engineering practitioners without background in formal methods.

Consider Figure 2**(d)**, which demonstrates a semiparsing technique called iterative lexical analysis [5] (a similar technique has recently emerged in a more modern framework called TEBA for analysis of tokenised syntax trees [17]). The

[1] GrammarLab: `http://grammarware.github.io/lab`.

technique relies mainly on patterns which are classified in levels: the higher the level, the more unstable and the less desirable the pattern is. So, on the first level there are strict matches for terminals such as keywords, and on the last level there are "desperate" heuristics that are meant more to ensure that the process produces some kind of result than to actually claim any correctness. Hence, we only work with the left column of our megamodel: the higher we are in the model, the more abstract and imprecise patterns are applied. There is no direct correspondence between pattern levels and layers of the megamodel, but for each concrete pattern we can easily find a place. For example, a pattern that detects strings and demotes the role of tokens inside a string from possible metasymbols (e.g., so that a curly bracket in `a = "b{";` is never used for block identification) clearly works on TTk, while a pattern that matches an identifier followed by a bracketed comma-separated list of identifiers followed by a block of statements and promotes it to a function definition, naturally produces Lex.

Operations for descending from Lex to TTk to Tok to Str are not explicitly described in the paper about iterative lexical analysis, but are certainly available in any sensible framework: we need to flatten (unfold) all hierarchical constructs to get down to TTk, disregard type information to get down to Tok and concatenate all tokens to get all the way to Str.

3 Conclusion

Megamodels are used as an understanding aid in complex scenarios involving various technologies, software languages, methodologies, approaches and transformations [3,4]. Renarrations of megamodels improve their usefulness by guiding megamodel consumers through the forest of immanently complicated artefacts and mappings [11,18]. Megamodels, whether ad hoc (a sentence "model M conforms to a metamodel MM" is in fact a tiny megamodel) or formal (AMMA, MEGAF, SPEM, MCAST, MegaL, CT), perform undeniably well for teaching purposes when introducing students to a new technology and explaining subtle differences between two almost identical technologies. In this paper, we have claimed that the same approach can be used for internal "languages", the ones that are hiding behind the scenes inside our tools. For this purpose, we propose to have one baseline megamodel of the domain — in a formal sense, it will include a lot of abstract entities, unbounded elements, constraints based on roles, etc — and use its refined renarrations for each of the concrete technologies that need to be explained and understood. We have demonstrated this approach with our megamodel for parsing in a broad sense [21], which we have used as a baseline model for four renarrations: classic lex+yacc parsing [1,7,12], ANTLR language workbench [14], Rascal one-stop-shop [9] and iterative semiparsing [5,17,19].

Beside the obvious future work claims such as promises of (mega)modelling different domains and perhaps even megamodelling relations among such domains, some other open questions remain. For example, some megamodels require explicit distinction between kinds of mappings they express (injective, bijective, monomorphic, isomorphic, asymmetric bidirectional, symmetric bidirec-

tional, etc), and such distinctions would also have to be properly specified and renarrated. In other cases, the modelling framework may already have a metamodel suitable for expressing typical renarrations, and the megamodel navigating arsenal would need to be adjusted with respect to the language it must be expressed in (instead of the opposite situation which we always assume).

As any other modelling method which introduces unification and heterogenuity, (mega)modelling different technologies with renarrations of the same baseline megamodel can help not only in explaining the actual state of the art, but also in spotting singularities. Anything irregular could be a signal of a bug, a not yet implemented feature or a comprehension mistake. Why is there a mapping from Cst to Ast in Rascal but no universtal mapping from Ast to Cst? Perhaps we should include one! Is there a good reason for Dra to not be accessible in Rascal? Having it explicitly as a (possibly optional) first class entity could allow us to do things we otherwise cannot! Would it help organising patterns for ILA/TEBA based not on their "desperation", but on the kind of artefacts they are actually dealing with (untyped tokens, typed tokens, grouped tokens)? Exploration of the extent of usefulness of such conclusions remains future work.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. A. H. Bagge, R. Lämmel, and V. Zaytsev. Reflections on Courses for Software Language Engineering. In *Tenth Educators Symposium (EduSymp 2014)*, 2014.
3. J. Bézivin, S. Gérard, P.-A. Muller, and L. Rioux. MDA components: Challenges and Opportunities. In *Proceedings of the First International Workshop on Metamodelling for MDA*, pages 23–41, Nov. 2003.
4. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. *OOPSLA & GPCE, Workshop on best MDSD practices*, 2004.
5. A. Cox and C. Clarke. Syntactic Approximation Using Iterative Lexical Analysis. In *Proceedings of the International Workshop on Program Comprehension (IWPC 2003)*, pages 154–163, 2003.
6. J.-M. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *MoDELS*, pages 151–167, 2012.
7. S. C. Johnson. YACC—Yet Another Compiler Compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, 1975.
8. S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies*, pages 3–42, 1956.
9. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *GTTSE 2009*, volume 6491 of *LNCS*, pages 222–289. Springer, Jan. 2011.
10. R. Lämmel and A. Varanovich. Interpretation of Linguistic Architecture. In *ECMFA*, pages 67–82, 2014.
11. R. Lämmel and V. Zaytsev. Language Support for Megamodel Renarration. In *XM 2013*, volume 1089 of *CEUR*, pages 36–45. CEUR-WS.org, Oct. 2013.
12. M. E. Lesk. LEX—A Lexical Analyzer Generator. Computer Science Technical Report 39, AT&T Bell Laboratories, 1975.
13. A. Okhotin. Conjunctive and Boolean Grammars: The True General Case of the Context-Free Grammars. *Computer Science Review*, 9:27–59, 2013.

14. T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.
15. R. Salay, J. Mylopoulos, and S. Easterbrook. Using Macromodels to Manage Collections of Related Models. In *CAiSE*, pages 141–155. Springer, 2009.
16. K. Sikkel. *Parsing Schemata — a Framework for Specification and Analysis of Parsing Algorithms*. Springer, 1997.
17. A. Yoshida and Y. Hachisu. A Pattern Search Method for Unpreprocessed C Programs based on Tokenized Syntax Trees. In *SCAM*, 2014.
18. V. Zaytsev. Renarrating Linguistic Architecture: A Case Study. In *MPM 2012*, pages 61–66. ACM, Nov. 2012.
19. V. Zaytsev. Formal Foundations for Semi-parsing. In *CSMR-WCRE 2014 ERA*, pages 313–317. IEEE, Feb. 2014.
20. V. Zaytsev. Software Language Engineering by Intentional Rewriting. *EC-EASST; Software Quality and Maintainability*, 65, Mar. 2014.
21. V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In *MoDELS*, volume 8767 of *LNCS*, pages 50–67. Springer, Oct. 2014.