ACM/IEEE 16th International Conference on
Model Driven Engineering Languages and Systems

September 29, 2013 – Miami, Florida (USA)

XM 2013 – Extreme Modeling Workshop Proceedings

Juan de Lara, Davide Di Ruscio, Alfonso Pierantonio (Eds.)

Published on Nov 23, 2013 v1.0

Editors' addresses:

Juan de Lara
Escuela Politécnica Superior
Departamento de Ingeniería Informática
Universidad Autónoma de Madrid (Spain)

Davide Di Ruscio
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila (Italy)

Alfonso Pierantonio
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila (Italy)

Organizers

Juan De Lara	Universidad Autonoma de Madrid (Spain)
Davide Di Ruscio	Università degli Studi dell'Aquila (Italy)
Alfonso Pierantonio	Università degli Studi dell'Aquila (Italy)

Program Committee

Colin Atkinson	University of Mannheim (Germany)
Nelly Bencomo	Inria (France)
Jordi Cabot	INRIA-École des Mines de Nantes (France)
Antonio Cicchetti	Mälardalen University (Sweden)
Tony Clark	Middlesex University (UK)
Gregor Engels	University of Paderborn (Germany)
Jean-Marie Favre	University of Grenoble (France)
Jesus Garcia-Molina	Universidad de Murcia (Spain)
Cesar Gonzalez-Perez	Incipit, CSIC (Spain)
Jeff Gray	University of Alabama (USA)
Esther Guerra	Universidad Autónoma de Madrid (Spain)
Robert Hirschfeld	Hasso-Plattner-Institut (Germany)
Gerti Kappel	Vienna University of Technology (Austria)
Philipp Kutter	Montages AG (Switzerland)
Stephen Mellor	Freeter (UK)
Richard Paige	University of York (UK)
Bran Selic	Malina Software Corp. (Canada)
Jim Steel	University of Queensland (Australia)
Jesús Sánchez Cuadrado	Universidad Autónoma de Madrid (Spain)
Antonio Vallecillo	Universidad de Málaga (Spain)
Mark Van Den Brand	TU/e (The Netherlands)
Vadim Zaytsev	CWI (The Netherlands)

Preface

Increasingly, models are starting to become commonplace and Model-Driven Engineering is gaining acceptance in many domains, including

- Automotive Software Engineering
- Business applications and financial organizations
- Defense / aerodynamics / avionic systems
- Telecommunications domain

Raising the level of abstraction and using concepts closer to the problem and application domain rather than the solution and technical domain, requires models to be written with a certain agility. This is partly in contrast with MDE whose conformance relation is analogous to a very strong and static typing system in a current programming language. For instance EMF does not permit to enter models which are not conforming to a metamodel: on one hand it allows only valid models to be defined, on the other hand it makes the corresponding pragmatics more difficult. In this respect, there is a wide range of equally useful artefacts between the following extremes

- Diagrams informally sketched on paper with a pencil
- Models entered in a given format into a generic modeling platform, e.g., EMF.

At the moment MDE encompasses only the latter possibility, while depending on the stage of process it might make sense to start with something closer to the former to eventually end up with the latter one. For instance, this clearly requires different notions of conformance and the possibility to even have a method for user-defined conformance relations depending on the scope. In other words, we do need different forms of agility in terms of both artefacts (the way they are conforming to metamodels) and processes (the way they are created and whose subsequent versions linked together in a consistent and uniform framework).

The Extreme Modeling Workshop 2013 has been co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages & Systems. It provided a forum for researchers and practitioners where to discuss different forms of agility as demonstrated by the technical program, including

- model/metamodel co-design
- domain-specific agility
- agility and languages

We thank the authors for their submissions and the program committee for their hard work.

November 2013

Juan de Lara, Davide Di Ruscio, and Alfonso Pierantonio

Table of Contents

Keynote

Agile versus MDE - Friend or Foe?

Jon Whittle

Lancaster University

Agile methods and MDE are often considered to be polar opposites of each other. Agile methods are fast, adaptive, responsive; MDE is heavyweight, plan-first, and requires high up-front investment. But are agile methods and MDE really so contradictory? Or could they co-exist and even work together to maximize each other's benefits? In this talk, I will reflect upon our recent study with two large companies that have both recently introduced agile methods into a model-driven environment. I will describe where they succeeded, and failed. And discuss how MDE tools do or do not support working in an agile manner. This is joint work with Hakan Burden and Rogardt Heldal.

Jon Whittle *is Full Professor and Chair of Software Engineering at Lancaster University's School of Computing and Communications. He is well known for his work in software modeling, scenario-based requirements engineering, aspect-oriented software development, and, more recently, empirical studies of model-driven development in practice. He leads two research groups at Lancaster - one on software engineering and one on social computing. He has been on the PC of most major software engineering conferences, including ICSE, ASE, MODELS, AOSD, RE, SPLC. He chaired the MODELS Steering committee from 2006-2008, was PC Chair for MODELS 2011, and Experience Track Chair for MODELS 2006.*

Programmatic Muddle Management

Dimitrios S. Kolovos, Nicholas Matragkas,
Horacio Hoyos Rodríguez, and Richard F. Paige

Department of Computer Science, University of York,
Deramore Lane, York, YO10 5GH, UK
{dimitris.kolovos, nicholas.matragkas,
hhr502, richard.paige}@york.ac.uk

Abstract. In this paper we demonstrate how diagrams constructed using general-purpose drawing tools in the context of agile language development processes can be annotated and consumed by model management programs (such as simulators, model-to-model and model-to-text transformations). The aim of this work is to enable engineers to engage in programmatic model management early in the language development process, so that they can explore whether or not the languages and models constructed are fit for purpose. We demonstrate a proof-of-concept prototype developed atop the Epsilon platform and a flexible graph definition language (GraphML).

1 Introduction

The quality and usefulness of a Domain Specific Language (DSL) depends on accurately identifying the domain concepts, their features and relationships. As such, the involvement of domain experts in the language development process is crucial. In the early stages of the language development process, domain experts often provide informal example diagrams/sketches from which engineers can infer a first version of the metamodel of the envisioned language. To obtain additional feeback, engineers then need to develop an initial version of a language-specific modelling tool that enables domain experts to further experiment with the language. This typically constitutes the first step of an iterative process during which the metamodel of the language can undergo several revisions. When 3-layer modelling frameworks such as MOF/EMF are used, for each change in the metamodel, language engineers need to update and re-deploy a new version of the modelling tool, and for non-additive changes to the metamodel they also need to provide support for automated migration of older models.

To achieve shorter and more efficient iteration cycles, several techniques that challenge this top-down metamodel-centric approach have recently been proposed. In such approaches, the early phases of the language development process involve the construction of *example diagrams* using flexible drawing tools, which can be used to (semi-)automatically infer a rigid metamodel only once sufficient confidence in the completeness and maturity of the language has been developed.

In this paper we argue that example diagrams constructed in the context of this process should also be processable by model management programs (such

as simulators, model-to-model and model-to-text transformations) so that engineers can develop additional and early confidence that the constructed language is fit for purpose. The rest of the paper is organised as follows. In Section 2 we provide an overview of related work in the field of bottom-up and agile metamodelling. In Section 3 we illustrate an approach for enabling engineers to engage in programmatic model management activities early in the language development process, and demonstrate a proof-of-concept prototype developed atop the Epsilon platform and a flexible graph definition language (GraphML). In Section 4 we conclude and provide directions to further work.

2 Background and Motivation

In [1], the authors propose an example-driven approach where users are able to construct informal diagrams using the Dia drawing tool, and these diagrams are then used to infer appropriate metamodels in an interactive manner. Similarly, in [2] the authors introduce a systematic semi-automated approach to create visual DSLs from a set of domain model examples provided by an end-user. The MetAmodel Recovery System (MARS) [3] is a semi-automatic inference-based system for recovering a metamodel from a set of instance models through application of grammar inference algorithms. This approach does not rely on example models provided by end-users, but it relies on models, which no longer conform to a metamodel due to its evolution. In [4], the authors present a tool (GraCoT) that supports co-development of EMF models and metamodels, in a loosely-coupled manner that promotes agility and simplifies the process of co-evolution.

To our knowledge, research in this area so far has focused solely on agile model construction and automated metamodel inference. In our view, to further validate the maturity and completeness of a metamodel, it is also important for language engineers to develop some confidence that models conforming to this metamodel can support the automated model management operations involved in the envisioned MDE workflow (simulation, model-to-model and model-to-text transformation etc.)

3 Proposed Approach

In this paper we illustrate an approach for rendering diagrams constructed using general-purpose drawing tools amenable to programmatic model management. An overview of the proposed approach is illustrated in Figure 1. Consistently with previously-proposed bottom-up metamodelling techniques, in this approach language engineers and domain experts can start the language development process by drawing diagrams depicting example models, which (conceptually) conform to the envisioned language, using a general purpose diagram drawing tool.

In the next stage, engineers can augment these conceptual diagrams using a set of predefined textual annotations (discussed in Section 3.2) to specify the

types and features of diagram elements of interest in an agile manner. Annotated diagrams are then automatically transformed into an intermediate representation (*muddle*) that can be programmatically managed using existing model management languages.

In this work we use GraphML, the conceptual metamodel of which is illustrated in Figure 2, for diagram drawing, and languages of the Epsilon platform [5] for automated model management, but in principle this approach should be applicable to other diagram formats and model management languages.

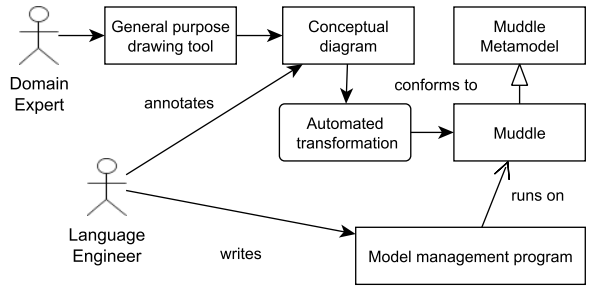

Fig. 1. Process Overview

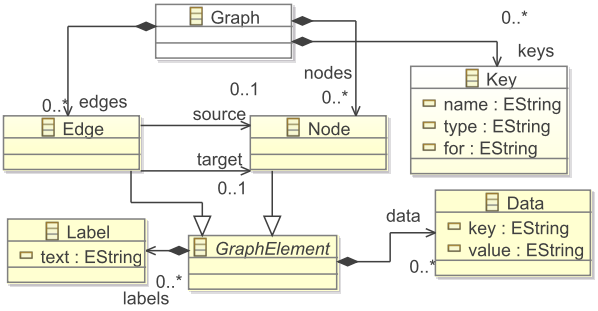

Fig. 2. GraphML Metamodel

3.1 Running Example

We illustrate the process of constructing, annotating, and programmatically managing GraphML diagrams through a running example. In this example, our aim is to define a flowchart language that supports timed events and delays. To develop some confidence that the envisioned language is feature-complete, we also need to implement a proof-of-concept program that can execute/simulate models that conform to the language.

We start by using the yEd[1] GraphML-compliant tool to draw an example diagram that conceptually conforms to the envisioned flowchart language. The diagram, illustrated in Figure 3 consists of labeled rectangles which conceptually represent actions, a diamond which represents a decision, directed edges which represent transitions, a hexagon that represents the triggering event, a circle which represents a delay, and a hexagon which represents the time at which the attached event should fire for the first time.

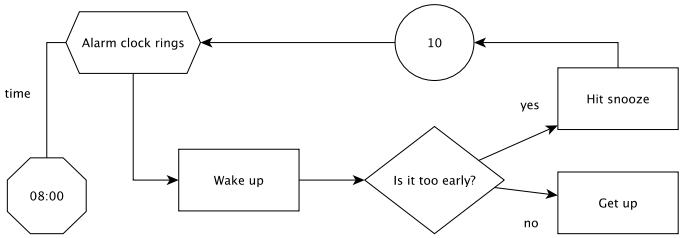

Fig. 3. Flowchart Diagram

We now take a leap and in Listing 1.1 we present the implementation of a simple simulator for such flowcharts, expressed in the Epsilon Object Language [6], an imperative OCL-based model query and transformation language. We provide a brief overview of the behaviour and the organisation of the simulator and then demonstrate how we need to annotate the diagram of Figure 3 so that the simulator program can use it as an input model that can drive its execution.

```
1  var event = Event.all.selectOne(e|e.entryPoint = true);
2  var time = event.time.hours.toMinutes();
3  event.process();
4
5  operation Event process() {
6    ("Event: " + self.name + " at " + time.toHours()).println();
7    self.outgoing.at(0).target.process();
8  }
9
10 operation Action process() {
11   ("Action: " + self.name).println();
12   if (not self.outgoing.isEmpty()) {
13     self.outgoing.at(0).target.process();
14   }
15 }
16 operation Decision process() {
17   ("Decision: " + self.name).println();
18   var random = self.outgoing.random();
19   ("Chose: " + random.name).println();
20   random.target.process();
21 }
22
23 operation Delay process() {
24   time = time + self.mins;
25   ("Waited for " + self.mins + "mins").println();
```

[1] http://www.yworks.com/en/products_yed_about.html

```
26    self.outgoing.at(0).target.process();
27  }
28
29  operation String toMinutes() : Integer {
30    var parts = self.split(":");
31    return parts[0].asInteger() * 60 + parts[1].asInteger();
32  }
33
34  operation Integer toHours() : String {
35    return (self / 60).asString().pad(2, "0", false) +
36      ":" + (self - (self / 60)*60).asString().pad(2, "0", false);
37  }
```

Listing 1.1. Simple flowchart simulator

- Assuming that a flowchart can contain many events, in line 1 we select one event that has its *entryPoint* attribute set to true;
- In line 3, we keep a copy of the time (converted to minutes) at which this event is fired for the first time;
- In line 4, we process the target of the first outgoing transition of the event; Calls to *process()* operations are dynamically dispatched depending on the type of their context object, and behave as discussed below;
- The *Event.process()* operation prints a message and processes the target of its first outgoing transition;
- The *Action.process()* operation prints a message and then, if the action has any outgoing transition, it processes the target of the first of them;
- The *Decision.process()* operation chooses a random outgoing transition, prints its name and processes its target;
- The *Delay.process()* operation adds the delay time to the global time, prints a message and then processes the target of its first outgoing transition;
- The *toMinutes()* and *toHours()* operations can convert HH:MM-formatted time strings to integers (number of minutes) and vice versa.

A sample execution trace of the simulator appears below.

```
1  Event: Alarm clock rings at 08:00
2  Action: Wake up
3  Decision: Is it too early?
4  Chose: yes
5  Action: Hit snooze
6  Waited for 10mins
7  Event: Alarm clock rings at 08:10
8  Action: Wake up
9  Decision: Is it too early?
10 Chose: no
11 Action: Get up
```

3.2 Annotating GraphML Diagrams

To facilitate the execution of model management programs such as the one illustrated in Listing 1.1, we need to annotate diagram elements with additional information. For example, we need to declare that the type of all rectangle nodes in this diagram is *Action*, and that the type of directed edges is *Transition*. As GraphML does not provide built-in support for capturing type-related information for nodes and edges, we need to use GraphML's extensibility facilities[2] to define *Type* extension fields for nodes and edges.

[2] `http://docs.yworks.com/yfiles/doc/developers-guide/graphml.html`

The value of the *Type* extension field of a node/edge needs to adhere to the `name (> name)*` pattern, where > is used to denote inheritance. For example, by setting the *Type* field of the *Wake up* node to *Action* > *FlowchartElement*, we define that the node is an instance of the *Action* type, and that the *FlowchartElement* type is a super-class of *Action*. All types are unique by name and are created the first time they are encountered in the diagram. For example, by subsequently setting the *Type* field of *Hit snooze* to *Action*, we are reusing the *Action* type defined in *Wake up* instead of creating a new one. Beyond type-related information, we also need to capture additional information using the following GraphML extensions summarised in Table 1.

Table 1: GraphML extensions

Extension	**For**	**Description**	**Pattern**
Properties	Node, Edge	Descriptors and values for primitive attributes of nodes/edges	(int|String|boolean|real)? (*)? name (= value)?
Default	Node, Edge	Descriptor of the slot under which the first label of the node/edge should be made accessible	(int|String|boolean|real)? name
Source role	Edge	Descriptor of the role of the source end of the edge	name (*)?
Target role	Edge	Descriptor of the role of the target end of the edge	name (*)?
Role in source	Edge	Descriptor of the role of the edge in its source node	name (*)?
Role in target	Edge	Descriptor of the role of the edge in its target node	name (*)?

The value of the *Properties* field of a node/edge can contain zero or more lines of text. Each line needs to adhere to the pattern above and define the type, multiplicity, name and value of the property. For example, by setting the value of the *Type* field of the *Alarm clock rings* node to *Event* and the text of its *Properties* field to *boolean startEvent = true*, we define that the node has a single-valued boolean *startEvent* property, with a value set to *true*.

The value of the *Default* field should conform to the pattern above and define the name of the default slot of the node/edge and, optionally, its primitive type (defaults to String). For example, by setting the *Default* field of the *Wake up* node to *name*, the first label of the node that does not match the property descriptor pattern (in this case, the *Wake up* label), will be made accessible through a *name* property of type String.

The values of the *Source role*, *Target role*, *Role in source*, and *Role in target* fields of an edge define the name and multiplicity of the respective roles. For example, in the *yes* transition we define the following values for these properties:

Source role: *source*, Target role: *target*, Role in source: *outgoing **, Role in target: *incoming **.

3.3 Deriving a Muddle

The next step of the process is to parse the annotated GraphML diagram and construct an intermediate model (*muddle*) that conforms to the metamodel of Figure 4. This is achieved through a multi-pass transformation which is transparent to the end-user and which comprises the following steps.

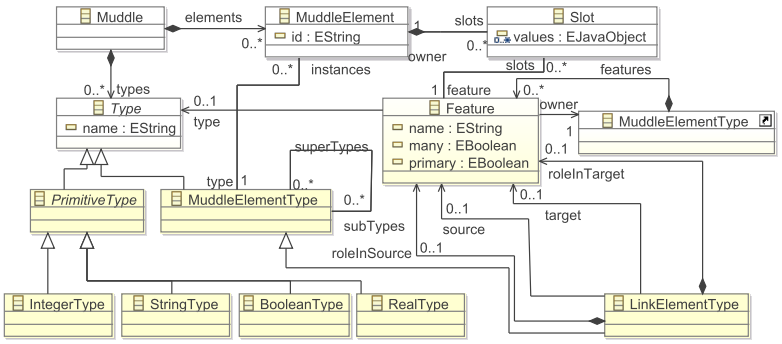

Fig. 4. Intermediate (Muddle) Metamodel

1. For every typed node in the graph, it creates an empty *MuddleElement* in the intermediate model and its corresponding *MuddleElementType* (if the latter does not already exist). It also looks for nodes for which the *Default* field has a valid value. When this happens, the value of the *Default* field is used to produce a primary *Feature* which is added to the type of the created *MuddleElement*;
2. Iterates through the created elements and creates/populates their *slots*, based on the descriptors provided in the *Properties* field of the node. Again, for each new property a *Feature* is created and added to the type of the element. As such, by setting the value of the *Properties* field of *Alarm clock rings* to *boolean startEvent = true*, all model elements of type *Event* obtain a single-valued *startEvent* boolean feature;
3. Iterates through the labeled and untyped edges of the graph (e.g. the *time* edge in the diagram of Figure 3). For each edge, it adds an untyped *Feature* to the type of its source muddle element, a respective *Slot* to the source muddle element, and adds the target of the edge to the values of the slot;
4. Iterates through the unlabeled and untyped edges of the graph and attempts to fit their targets into appropriate slots of the source muddle elements (i.e. slots that already contain at least one value of the same type);

5. For every typed edge of the graph it creates an empty *MuddleElement* and its corresponding *LinkElementType*, similar to what was discussed for nodes in step 1. It also attempts to create primary, role in source, role in target, source and target *Features* for the created *LinkElementType*s;
6. Iterates through the typed edges of the graph and creates/populates their slots similar to what was discussed in step 2;
7. Adjusts the multiplicities of features based on the maximum number of values of their slots. In this process, single-valued features, slots of which contain more than one values become multi-valued (but not the opposite).

3.4 Consuming Muddles in Epsilon Programs

Epsilon provides an abstraction layer (Epsilon Model Connectivity – EMC[3]) that shields the languages of the platform from the intricacies of concrete model representations and enables them to access models conforming to a wide range of technologies. To enable Epsilon languages to access muddles, we have developed a new driver that implements the set of interfaces required by EMC. Due to space restrictions, a detailed discussion on the new driver is beyond the scope of this paper.

The driver enables all languages in Epsilon to query muddles. For example, in addition to the simulator of Listing 1.1, Listing 1.2 demonstrates an exemplar constraint written in the validation language of the platform (EVL[4]), and Listing 1.3 demonstrates an exemplar model-to-text transformation written in EGL[5].

```
1  context Decision {
2    constraint HasMoreThanOneOutgoingTransitions {
3      check: self.outgoing.size() > 2
4      message: "Decision " + self.name + " needs to have at least 2 outgoing
           transitions"
5    }
6  }
```

Listing 1.2. Validation constraint for flowchart models

```
1  The flowchart has [%=Action.all.size()%] actions:
2    [%for (action in Action.all) {%]
3    - [%=action.name%]
4    [%}%]
```

Listing 1.3. Model-to-text transformation for flowchart models

4 Conclusions and Further Work

In this paper we have argued for the importance of enabling engineers to engage in exploratory model management operations early on in the language development process and demonstrated an approach and a prototype that enables

[3] http://www.eclipse.org/epsilon/doc/emc
[4] http://www.eclipse.org/epsilon/doc/evl
[5] http://www.eclipse.org/epsilon/doc/egl

engineers to annotate and programmatically manage GraphML diagrams using languages of the Epsilon platform. In the future, we plan to investigate supporting additional GraphML constructs such as subgraphs and hyperedges.

In our view, while constructing diagrams using using general-purpose drawing tools can be very useful in the early phases of the language development process, it can become cumbersome and error-prone as the example diagrams and the DSL become larger and more mature - at which stage a transition to a language-specific modelling tool should be consider. To reduce the overhead of this transition, we plan to investigate inferring annotated metamodels that can then be consumed by tools such as Eugenia[6] to automatically generate language-specific model editors.

Acknowledgements

This research was part supported by the EPSRC, through the Large-Scale Complex IT Systems project (EP/F001096/1) and by the EU, through the Automated Measurement and Analysis of Open Source Software (OSSMETER) FP7 STREP project (318736).

References

1. Jesús Sánchez-Cuadrado, Juan Lara, and Esther Guerra. Bottom-up metamodelling: An interactive approach. In Robert France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 3–19. Springer Berlin Heidelberg, 2012.
2. Hyun Cho, J. Gray, and E. Syriani. Creating visual domain-specific modeling languages from end-user demonstration. In *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*, pages 22–28, 2012.
3. Faizan Javed, Marjan Mernik, Jeff Gray, and Barrett R. Bryant. Mars: A metamodel recovery system using grammar inference. *Inf. Softw. Technol.*, 50(9-10):948–968, August 2008.
4. Villalobos J. Gómez P., Sánchez M. Gracot, a tool for co-creation of models and metamodels in specific domains. In *Proc. Academics Tooling with Eclipse (ACME 2013) at European Conference on Object-Oriented Programming (ECOOP2013)*. ACM, 2013.
5. Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, Fiona A.C. Polack. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Proc. 14th IEEE International Conference on Engineering of Complex Computer Systems*, Potsdam, Germany, 2009.
6. Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.

[6] `http://www.eclipse.org/epsilon/doc/eugenia`

Extending Agile Practices in Automotive MDE

Ulf Eliasson[1] and Håkan Burden[2]

[1] Volvo Car Corporation, Sweden
ulf.eliasson@volvocars.com
[2] University of Gothenburg, Sweden
burden@cse.gu.se

Abstract. The size and functionality of the software in a modern car increases with each new generation. To stay competitive automotive manufactures must deliver new and better features to their customers at the same speed or faster than their competitors. A traditional waterfall process is not suitable for this speed challenge - a more agile way of working is desirable. By introducing MDE we have seen how individual teams at Volvo Cars adopt agile practices, resulting in tensions while the organization at large still uses a waterfall process. In an exploratory case study we interviewed 17 engineers to better understand how agile practices can be extended beyond individual teams. Where the tensions have their source in the technical specification of the software components and their interfaces, it turns out that it is company culture and mindsets that are the main hurdles to overcome.

Keywords: Interface Specification, Tailoring, Exploratory Case Study

1 Introduction

The size and functionality of the software in a modern car increases with each new generation [1] and the software can be executing on in the order of 100 Electronic Control Units (ECUs) spread out in the car. This causes software development to take an increasing part of the total R&D budget for new car models [2]. Automotive manufacturers are traditionally mechanical and hardware oriented companies. The software processes often resemble the traditional waterfall since that works for mechanical development - but it might not necessary be the best for developing software. In a world that moves faster and faster it is crucial to push new features out faster than the competitors. One way that automotive manufactures can speed up their software process is to develop more software in-house, making it possible to iterate their software and introduce new features faster than when ordering the same from a supplier. By introducing MDE we have seen that the domain experts are directly involved in the software implementation and how lead times become shorter.

2 Automotive Software Development at VCC

The system development process at Volvo Car Corporation (VCC) is a waterfall process with a number of integration points throughout where artifacts should

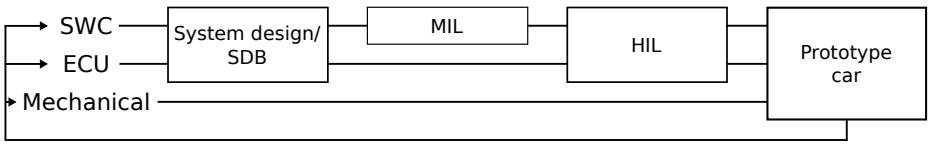

Fig. 1. Overall process view.

be delivered. There are three tracks of parallel development - the software components (SWC), the hardware (known as electronic control units or ECU) and the mechanical parts. Parts of the software is developed in-house while the rest of the software, and most hardware and mechanical systems are developed by sub-contractors. Each iteration of the process has a number of integration steps, as shown in Fig. 1. The system design and signal database (SDB) is the first integration step where the software and its interfaces are designed. Model-In-the-Loop (MIL) is where the implementation Simulink-models are integrated together with other models, either single components together with test models for testing, or in complete MIL with models of different components from different teams. After code generation from the implementation models they are integrated in Hardware-In-the-Loop (HIL) executing on hardware but with the environment around it simulated with models executing in real-time. Finally everything is integrated in a prototype car.

Early on, all the requirements and the system design for the next iteration is captured as a model inside a custom-made tool, referred to as SysTool. The model contains, among many things, software components, their responsibilities as requirements, the software components deployment on ECUs, and the communication between them as ports and signals. The signals are basically periodic data that is sent, and they are described down to the byte size of their data types. The signal definition also includes timing requirements, such as frequency and maximum latency. At a certain point in time the model is frozen and no new changes are allowed until the next iteration starts. These freezes usually last for 20 weeks.

The SysTool model is used for two things. Firstly, signals, ports and their connections and timing requirements are used to implement the SDB. The SDB is used by the software on the ECUs and the internal network nodes to schedule all the signals passing through the networks in the car. The packing and scheduling of the signals is done manually. It takes 9 weeks from a freeze until the SDB is delivered and ready to be used. Secondly, the component model is transformed into Simulink model skeletons. Each Simulink model represents an ECU with skeletons of the deployed software components, including ports and connections. These models are then filled in with functionality by the developers as specified by textual requirements. If the system model changes the Simulink models are updated to reflect this by a click of a button and any implementation already done is kept intact. The implementation must be updated manually if it is dependent on signals that are removed or changed.

The executable Simulink models are tested together with plant models. The plant models represent the environment surrounding the ECUs, including electrical, physical and mechanical systems. This enables the developer to get instant feedback by running and testing the models s/he developed in isolation on their own PCs. The models are also integrated in a virtual car MIL environment where all models developed in-house are integrated and executed. The time frame for getting feedback after delivering a model to the virtual car is counted in days. This gives the developers a possibility to do requirements and design exploration and validation on component level. However, sub-contractors developing software do not normally provide models of their software meaning that where suppliers are developing functionality there are holes in the MIL environment. These holes are filled in by models describing how to develop their software as they see fit, as long as they can fit their work in to the larger overall process and deliver in time. Therefore development within the Simulink models for one ECU can and is conducted agile. However, as soon as there is a need to extend or modify the SDB the developers need to adapt to the overall waterfall process.

The suppliers deliver their software as binaries. Code is generated from the in-house models, built to binary and then the two are linked together. The finished software is loaded on hardware and is then tested on HIL rigs, see Fig 1. Because the supplier only delivers binaries this is the first time that in-house and supplier developed software can be integrated and validated together. Later, software and hardware is integrated with the mechanical systems in a complete prototype vehicle and tested. This is the first time that the whole system is tested together. Each full iteration in Fig 1 has a deadline referred to as a gate and the time between the gates is referred to as E-series.

The use of MDE in the teams makes it possible to break free from the suppliers making it possible to execute and test the in-house software without having to wait for the hardware and software from the suppliers. This enables the team to be agile and work in short iterations.

3 Method

Given that agile practices have shown to be possible at the level of individual teams, we wanted to answer the question:

RQ: Which are the challenges and possibilities for a more agile software development process on a system level?

The question was answered by two exploratory case studies [3]. The main data was collected through two parallel interview series, conducted by the two authors independently of each other. The first set of interviews was launched internally by Volvo Cars in order to identify factors that could improve the existing process. The second study was initiated as a collaboration between Volvo Cars and academia to evaluate and improve the ongoing transition into Model-Driven Engineering, MDE. Both interview sets included eight interviewees, without an overlap between interviewees. In general the interviewees had a background in

electronics, physics, automation or mechanical engineering with a limited training in software development from VCC. All interviews were conducted at VCC in Gothenburg, Sweden, and complemented by active participation by one of the authors and on-site observations by the other.

The study focusing on process conducted the first interview in May 2012, the study on MDE started in January 2013. Both sets of interviews were finished by April 2013. Since the interviews were conducted over a longer time frame, collecting and analyzing the data was done in an incremental fashion [3]. From the first interviews of each study a preliminary analysis emerged, identifying themes and concepts that the engineers found challenging in the current combination of process and model-driven implementation. Using a semi-structured format allowed the interviewers to explore new topics as they arose but also to see if spiring hypotheses could be confirmed [4, 5].

The two studies were aligned in May 2013 by a manager who recognized that both lines of inquiry had come to the same conclusion independently of each other, despite the fact that one study was an internal effort to improve the existing process and the other study was an academic collaboration investigating MDE. After the analysis of the interviews had been completed (see section 4) a system architect involved in the scheduling of the SDB was interviewed to reflect on the outcome of the analysis (presented in section 5).

4 Challenges for Agile MDE Practices

From our interviews we discovered a number of issues that have their roots in the waterfall process used on a system level. The most prominent issue, repeated by different interviewees, concerned the SDB in the SysTool. By freezing requirements and system design such a long time before delivery developers are forced to take premature decisions on what data they need to receive or send and where. Because the signals are the interfaces between different components, developed by different teams or sub-contractors, any negative impacts caused by premature decisions are not discovered until late in integration and therefore expensive to change.

The practice of freezing the interface implies that the engineers have to specify the interface they need before they fully understand the internal behaviour of the component being developed. This means that the interfaces are defined based on the assumptions the developer have at that time and subsequently there are signals that will never be used but still have a share of the limited capacity. This causes two problems. First extra signals need to be scheduled on the network, wasting time for the SDB group in their work as well as causing extra congestion on the network. It also makes it difficult for a developer on an ECU to know which signals are used or not used.

Q: So do you overload the interface? Throw in a signal just in case?

A: Yes, that is what we do. At least I do it [. . .] and then you end up with the problem knowing which signal it is you should actually use.

As with the spare signals above, developers add extra data-elements to their signals they create to future-proof them. This causes the same problems as with the spare signals but is also harder to redeem because one can't just remove a signal, the signal needs to be modified. It is also harder to check if a data element is used or not compared to seeing if a whole signal is used.

A: Also an old problem we have here at Volvo is that when someone wants to add a new signal, they know it will be hard to change the signal later. So to be prepared they add a few extra data bits to the signal, just in case.

Developers that figure they need to send or receive some data that they did not think about before the freeze do not want to halt their implementation until the next freeze, instead they use existing signals in creative ways. This includes using signals and data-elements in ways that are not described in the requirements making them behave differently than intended. When other groups depend on these signals misinterpretations occur which causes problems.

A: But we have a text document that's about 300 or 400 pages in total if you take all the documents. And that hasn't been updated for a couple of years. So this is wrong. This document is not correct.

Since the textual documentation is inconsistent with the implementation and the interface is overloaded the engineers start to mistrust the artifacts that are supposed to support them in their development. As a consequence one of the other interviewees had developed a work-around for handling that the interface specification was constantly outdated. The solution is to sieve through a second document after the information that concerns the interface being developed and translate that information into a new, temporary, specification.

A: We have in our requirements a list of signals used in the requirement. Now that list is seldom updated. It's hardly ever, so they're always out of date. So I don't actually read them anymore. I just go in through the specific sub-requirements and I read what is asked for my functionality. This is asked. What do I need? I need this and this. So, yeah, so I do it manually, I guess.

The reoccurring theme behind these issues are that developers are forced to make unfounded assumptions about what the SDB will or needs to contain and how the signals in it will be used. Also a shared view between the developers was that it is the development of mechanical and hardware systems and the MDE tools that forced the use of a waterfall process. The issues caused by these assumptions are not found until late in the process with a considerable cost in both time and money as the result.

5 Possibilities for Extending Agile MDE Practices

Based on the results from previous interviews we interviewed an architect responsible for tool and process development at VCC and brought up the identified challenges. He did not see the technical problems of the SysTool and its models to be the main obstacle to overcome, rather it is the culture and the way of thinking in the organization that needs to change.

Q: Why do we have these freezes?

A: We have a traditional gated process, and then you have freezes and gates for everything, period. [...] If we think waterfall it is very logical that we have these freezes, that is what we have to rethink completely. What I'm thinking is that we need to change our system definition process so that it is agile and we can make changes whenever we want [...] and then we can drive this in different speed with different suppliers but it shouldn't be our process that is stopping us.

He also did not think that the hardware development done in parallel requires a waterfall system development process with gates. However, working with suppliers is one reason for having the gated system development process.

Q: How much does it have with working against suppliers and developing hardware?

A: Yes, that is part of the answer. The connection to hardware I do not see as very strong, because the hardware development is not really in the same cycles anyway. Of course, we have hardware upgrades and when it affects the software then it has that connection, if there is some sensor or something that is replaced, but many of these problems are about changes to signals on the network and that is not connected to hardware at all really, at least not at that level. So the hardware is not a big factor. But supplier interaction is of course a factor, because we have a way of working with the supplier where we tell them that on this day we will send the specification and this day you will deliver, and there is a number of weeks between when we send the requirements and they should deliver and then we need a freeze so that we have something to send.

The time between freezing the database until there is a finished implementation of that version, is 9 weeks. Because the developers know that this is their last chance for a while to change the signals they wait until the last minute to put any requests for new or changing signals as close to the freeze as possible. Obviously this results in a lot of change requests to be processed at the same time. It is not until after the freeze that signals and other changes are checked for compatibility and consistency, which might result in a lot of work to make the system design model consistent again.

A: If we say that at this point in time you should submit all your change requests then you get all of these the Friday before instead of getting them more continuously and then you have these weeks of job ahead of you. [...] We need to start thinking about the SDB and frame packing as a product, like any node. So the nodes deliver their software and the SDB delivers the frame packing as a component that integrates with the other stuff. So instead of having a process where the frame packing and everything needs to be finished and done before you can start making a node it should be something that you integrate with the other stuff. [...] Integration is something that already today is happening continuously. It is not a specific day we integrate it is something you try during the E-series and in the end you make it work and then it is time for the next one. [...] At the gate between E-series we do a refactoring of the SDB, clean it up and pack it. What I think is that, we should continuously allow ourselves to add signals, and also allow each other to add redundant signals. If one signal is wrong we do not remove that signal, because the components are expecting to get this signal.

But we can all the time allow ourselves to add signals and therefore we can get double, triple or quadruple signals when we find our way forward. This is roughly as you work with product lines, you allow yourself to over-specify interfaces and so on.

In this way the SDB is tidied up at each gate instead of defined at the system design phase within each iteration.

A: Working like this we can basically end up at releasing a SDB every day throughout the series and as long as a test or E-series environment is alive we can release a SDB daily where we can add new signals to test. Then we can allow ourself to deliver in what frequency we want.

Q: It sounds like most things that need to change are soft issues, are there no technical obstacles for this?

A: No, not really. There is a need for more support in SysTool to make sure that additions that you make are backward compatible. Today this is mostly a manual process. So you need to build in locks so, for this E-series you can only introduce backward compatible changes so that we all the time can export a correct export. And a lot of the stuff that we manually need to clean up is checked automatically. So there are some small tool changes. But this is not the big thing, it is how we think and how we work.

The system model doesn't have to be executable, a non-executable model is enough for the checks that are needed. Also, because the system model is used to generate shells that are filled with executable Simulink models, developers might not see a need or purpose with having more executable models.

A: We will still have these E-series where there will be some refactoring, and to release such an E-series will still take two to three weeks to pack, so it might not be nine weeks but it might be four to five weeks before an E-series release that you need to say what you want in that release. However, the difference from now is that this is not your last chance to get things in it, it is just what will be in it from day one in this series. Then after this you will be able to get things into it as long as it is living.

Because the software development cycles are not bound to the hardware or mechanical development there is no need to follow a similar waterfall process. Also, a more agile system development process would not force the suppliers to be more agile, instead it would make it possible for the teams and their suppliers to work out the best way of working between them. Therefore we can use a more agile process, as already practiced by some of the ECU teams, on a system level. To enable this transition the SysTool needs to support static consistency checks, as proposed by [6], to give the developers and architects confidence in that the changes they make are backward compatible and will not break the integration.

6 Related work

Pernstål [7] has conducted a literature study of lean and agile practices in large-scale software development, such as in the automotive industry, concluding that there is a lack of empirical research within this area.

Eklund and Bosch [8] have developed a method and a set of measures to introduce agile software development in mass-produced embedded systems development, including automotive development. They have discovered that part of introducing agile methods is to gain acceptance in the organization for gradual growth and polishing of requirements instead of using a waterfall approach. They also say that it is possible to have agile software development even though the product as a whole is driven by a plan-driven process.

Kuhn et al. [9] and Arnanda et al. [10] have investigated MDE at General Motors. However, they have not looked at how MDE could change the process and help overcome some of the problems with the traditional process for developing automotive systems.

7 Conclusion and Future Work

The interviewees often thought that the agile challenges were related to MDE or the used tools. However, during the interview with one of the responsible architects for the tools and processes he identified them as caused by the process used on a system level. MDE would be the enabler for a more agile way of working as it provides the teams with a way of testing and iterate their design without having to wait for supplier software or hardware. To get the suppliers on board in such a way of working will take time. But a more agile system development process would not force the teams and suppliers to change their current way of working, but it enables teams and suppliers to agree on a way of working that suits them best instead of forcing them to fit in to the waterfall development process.

We have during our research discovered that agile MDE can be beneficial for automotive development. Using a language close to the domain, such as Simulink, enables engineers trained in the specific domain to work in an environment they recognize and can express their solution in. A model based environment where one can do physical modeling also enables the developers to quickly test their solutions on their own PCs. These are enablers for a more agile development process than is currently the norm in the automotive industry.

We have also discovered that the hardware or mechanical development does not force the software development to follow a waterfall process. The software development is so independent it could have its own process on top of hardware and mechanical development. A more agile software development process would also make it easier to adapt to changes in hardware or mechanical systems.

The interaction with sub-contractors is one of the obstacles that needs to be bridged before agile can happen on all levels. However, a more agile software process on the system level would permit the individual teams and sub-contractors to interact in any way they would think is best instead of forcing them to follow and fit it in to the waterfall process. Some systems are naturally less appropriate to develop in a completely agile way, such as brakes, and others are so stable and well known that there is little benefit or need for agile development. But having

the software process on the system level agile doesn't mean that these domains have to be developed agile, the sub-processes can still be as strict as they need.

The natural thing would be to try to spread the agile MIL environment in all directions and tailor the process [11] to utilize the possibilities of the tools. A first step to enable such a transition would be to allow for faster iterations of the SDB and extending the SysTool to do static consistency checks.

For the future, we plan to implement some of the proposed changes in section 5 to the process at parts of VCC and evaluate them, including looking at how external actors can be integrated in a more agile way of working. Changing a large organization will take time [12]. By starting bottom up we hope that the acceptance for change will be easier to achieve in respect to in-house developers [8] but also for building trust with sub-contractors [13].

References

1. Ebert, C., Jones, C.: Embedded software: Facts, figures, and future. IEEE Computer **42** (2009) 42–52
2. Broy, M.: Challenges in automotive software engineering. In: Proceedings of the 28th international conference on Software engineering. ICSE '06, New York, NY, USA, ACM (2006) 33–42
3. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering **14** (2008) 131–164
4. Robson, C.: Real World Research. 2nd edn. Regional Surveys of the World Series. Blackwell Publishers (2002)
5. Seaman, C.: Qualitative methods in empirical studies of software engineering. IEEE Transactions on Software Engineering **25** (1999) 557–572
6. Rumpe, B.: Agile modeling with the UML. In Wirsing, M., Knapp, A., Balsamo, S., eds.: Radical Innovations of Software and Systems Engineering in the Future. Number 2941 in Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 297–309
7. Pernstål, J.: Towards Managing the Interaction between Manufacturing and Development Organizations in Automotive Software Development. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden (2013)
8. Eklund, U., Bosch, J.: Applying agile development in mass-produced embedded systems. In Wohlin, C., ed.: Agile Processes in Software Engineering and Extreme Programming. Number 111 in Lecture Notes in Business Information Processing. Springer Berlin Heidelberg (2012) 31–46
9. Kuhn, A., Murphy, G.C., Thompson, C.A.: An exploratory study of forces and frictions affecting large-scale model-driven development. In France, R.B., Kazmeier, J., Breu, R., Atkinson, C., eds.: Model Driven Engineering Languages and Systems. Number 7590 in Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 352–367
10. Aranda, J., Damian, D., Borici, A.: Transition to model-driven engineering: what is revolutionary, what remains the same? In: Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems. MODELS'12, Berlin, Heidelberg, Springer-Verlag (2012) 692–708
11. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In:

MODELS 2013, 16th International Conference on Model Driven Engineering Languages and Systems, Miami, USA (2013)
12. Aaen, I., Börjesson, A., Mathiassen, L.: SPI agility: How to navigate improvement projects. Software Process: Improvement and Practice **12** (2007) 267–281
13. Christopher, M.: The agile supply chain: Competing in volatile markets. Industrial Marketing Management **29** (2000) 37–44

Supporting Agility in MDE Through Modeling Language Relaxation

Rick Salay and Marsha Chechik

University of Toronto
Toronto, Canada
{rsalay, chechik}@cs.toronto.edu

Abstract. Agility requires expressive freedom for the modeler; however, automated MDE processes such as transformations require models to conform to strict constraints (e.g. well-formed rules). One way out of this apparent conflict is to allow a "relaxed" version of a modeling language to be used by modelers and then use tool support to "tighten" such models so that they are conformant to the original constraints. In this paper, we explore the issues of relaxation and tightening of modeling languages and discuss the possibilities for tool support.

1 Introduction

The problem of agility in MDE arises because graphical models have two very different kinds of users: humans and programs. Humans use models to express themselves and communicate with each other. Programs manipulate models to do analyses or to transform them into other models. These two types of users give rise to a modeling dilemma: humans want expressive freedom and can cope with relaxed rules while programs need models to conform to precise constraints. How can this agility conflict be reconciled?

In this paper, we propose a transformation-based framework for addressing the agility conflict for a given modeling language by meeting the needs of both kinds of users. Human needs are satisfied by relaxing the language to permit greater expressive freedom. Program needs are satisfied by defining a *tightening* transformation that converts the model in the relaxed language back into the original, more strict, language.

We limit our scope by focussing on supporting two kinds of agility: *omission agility* – allowing the modeler to omit information in the model in order to express uncertainty, irrelevance, etc., and *clarity agility* – allowing the modeler to express information in the model more concisely or differently to improve clarity. Although our scope is limited, the usefulness of these forms of agility is justified by work in the philosophy of language relating to human communication. For example, Grice defines a "cooperative principle" that gives four maxims that hold in effective human communication [4]: *quantity* – making the contribution as informative as is required but no more informative than required; *quality* – being truthful; *relation* – being relevant; and *manner* – being clear. Both types

of agility we handle address quantity, relation and manner, whereas quality (i.e., truthfulness) is an orthogonal issue and is independent of language relaxation.

The paper is structured as follows. In Section 2, we illustrate different aspects of the two agility types using five examples. In Section 3, we give a preliminary framework for language relaxation and tightening and show how it can address our examples. Section 4 explores possible tool support for the framework. We discuss related work in Section 5 and conclude in Section 6.

2 Language relaxation and tightening by example

A modeling language can be relaxed in several different ways. In this section, we explore some of these possibilities using the examples depicted in Figure 1 (A-E), referring to these as Examples A-E, respectively. All of the examples use the language of UML class diagrams (CD). In the discussion below, assume that the metamodel of CD consists of a *vocabulary* defining the element and relation types in the language and a set of *constraints* defining well-formedness – i.e., a well-formed model must conform to the constraints. For each example, we first state what the modeler is attempting to express and the type of agility required, then describe the relaxation aimed to achieve this and finally, introduce the tightening transformation required.

Example A. The modeler wants to express that she doesn't yet know what sits on the other end of the `controlledBy` association (omission agility). To do this, she weakens the well-formedness constraint that a binary association must have a class on both ends. The tightening transformation assigns a class to the target of the `controlledBy` association. Since there is choice here (i.e., an existing class or a new class), this choice must be resolved.

Example B. The modeler wants to express that in the parallel inheritance hierarchies, the classes `Car`/`Driver` and `Plane`/`Pilot` are the intended pairings with the `controlledBy` association (clarity agility). To do this, she uses the vertical alignment in the layout to indicate the correspondences. Note that neither the vocabulary nor the constraints of the concrete syntax are affected, but the expressive power of the language is extended by giving the spatial relation of vertical alignment a special meaning. The tightening transformation defines an OCL constraint for each occurrence of the vertical alignment of a pair of classes that extend `Vehicle`/`Operator` to enforce the intended constraint.

Example C. The modeler wants to indicate that she isn't sure which class should hold the `park`() operation (omission agility). To express this, she wants to link `park`() to both classes but to do that, it would have to be simultaneously contained in two boxes. This "physical" constraint, which enforces the well-formedness constraint that an operation is owned by one class, cannot be weakened unless the boxes are made to overlap. Instead, for clarity, she opts for extending the vocabulary to allow operations to be specified externally to a class, using an ellipse and linked to the class with a dashed line (clarity agility). The tightening transformation adds an operation to a class for each ellipse linked to

the class. Since in this example, two owners of the operation `park`() are specified and this violates a well-formedness constraint, there is a choice (i.e., which class is the owner?) that needs to be resolved.

Example D. To reduce clutter, the modeler wants to put the name of the class outside, but close to, its box (clarity agility). To do this, she weakens the constraint that the class name is inside the box at the top. The tightening transformation defines text close to a class box as being the name of the class. To operationalize it, the definition of "closeness" must be given.

Example E. The modeler wants to express the fact that certain classes are "connected" without being specific about the type of connection – it can be a generalization, an association, etc. (omission agility). To do this, she extends the vocabulary with a special dashed line to indicate this relation. The tightening transformation resolves the dashed line to one of the class diagram relations that can hold between classes. Since there is choice here, someone needs to make it.

3 Towards a framework for relaxation and tightening

Our ultimate goal is to develop a framework for the relaxation and tightening of modeling languages to address the agility conflict. In this section, we use the examples of Section 2 to discuss the characterizing features of relaxation and tightening that could be parts of such a framework.

The approach is given schematically in Figure 2. We assume that a modeling language has a transformation $c2a$ that generates the abstract syntax for a model expressed using its concrete syntax. Modeling agility is supported by allowing the modeler to relax the concrete syntax to a new syntax, as needed, to provide the required expressive power. Then, when the model must be used for MDE operations, the tightening transformation T that transforms the model back to the more strict concrete syntax is constructed. The composition $c2a \circ T$ takes the relaxed model to the original abstract syntax, making it amenable to MDE operations such as transformation and analysis.

The approach is motivated by the observation that human and program (MDE) users of models have different foci: humans deal with concrete syntax while MDE primarily deals with the abstract syntax of a model[1]. Thus, all of our examples are in concrete syntax. Correspondingly, our transformation-based approach to relaxation and tightening is centered around concrete syntax rather than abstract syntax. The focus on concrete syntax does not limit the expressive power of the language relaxation; on the contrary, it is greater than if the relaxation were applied to abstract syntax. While all user-relevant information from the abstract syntax is preserved in the concrete syntax, the reverse is not true, e.g., Example D in Figure 1. *One of the contributions of the present work is to bring attention to the fact that extending MDE to address human issues such as agility requires transformations on the concrete syntax.*

[1] Model editors and model layout algorithms are notable exceptions to this.

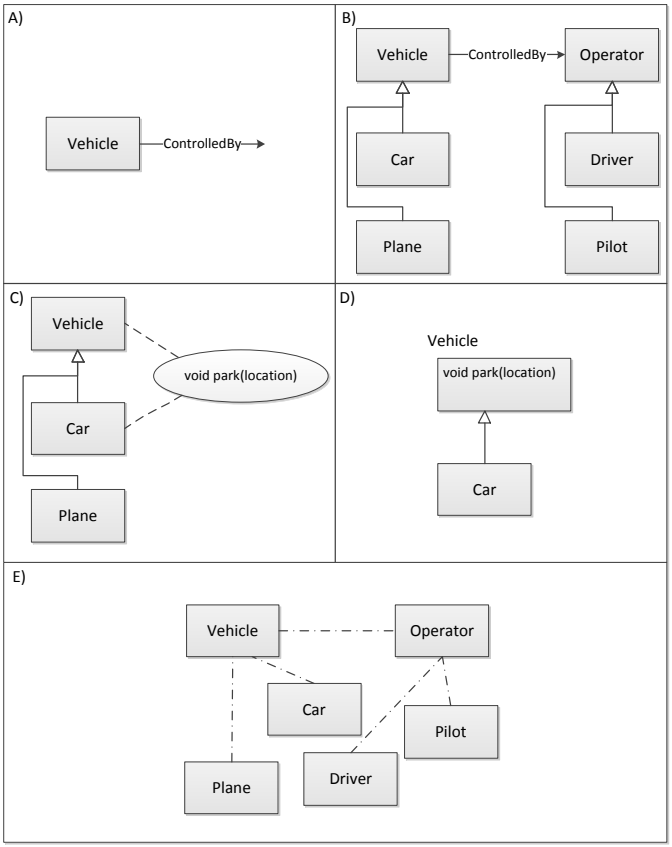

Fig. 1. Examples of language relaxation.

The motivation in Section 1 for limiting our scope to omission and clarity agility was to ensure that a tightening transformation T always exists (though it might not be necessarily unique). Relaxation to omit information can be tightened by adding back information; while relaxation to express information differently for clarity is tightened by defining an alternate expression in terms of native constructs in the original language.

3.1 Implementing relaxation and tightening

We now consider the ways in which elements of Figure 2 are affected by the relaxation and tightening process. The concrete syntax can be affected in two ways: *extending the vocabulary* (Examples C and E) or *weakening the well-formedness constraints* (Examples A, C and D). When the vocabulary is extended, the interpretation transformation `c2a` must be correspondingly broadened; but the

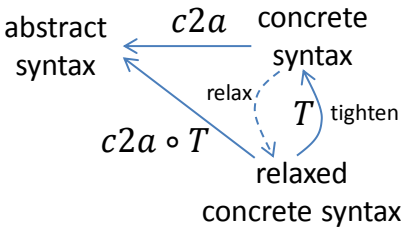

Fig. 2. Transformation-based approach to address model agility.

broadening of `c2a` may still be required even if the concrete syntax is unaffected – this is the case with Example B.

The language aspects involved in relaxation have corresponding tightening actions. Relaxing by extending the vocabulary requires tightening by redefining these extensions in terms of existing constructs. Relaxing by weakening constraints requires tightening by repairing the violations of the constraints that were weakened. In Section 4, we discuss these actions in more detail.

Support for agility. The examples in Figure 1 show how our approach applies to both clarity and omission agility. Clarity agility uses vocabulary extension in Example C and constraint weakening in Example D. In addition, Example B illustrates clarity agility when no language changes are made and only `c2a` is broadened.

Omission agility uses vocabulary extension in Examples C and E and constraint weakening in Example A. Furthermore, three different ways of omitting information are illustrated: *dropping information* (Example A), *providing alternatives* (Example C) and *using abstraction* (Example E).

Whenever omission agility is being addressed, choice may occur in the tightening process, and there are different ways to address this choice. One alternative is to elicit a decision from the modeler. Another possibility is to make a "systematic" decision (e.g., always create a new class in Example A). Yet another possibility is to defer the decision and keep all choices. We discuss this last possibility in Section 3.2.

Special characteristics of concrete syntax. The physical nature of concrete syntax makes it different from abstract syntax and this has two important implications for the framework. First, existing spatial relations that are "unused" can be appropriated for increasing expressiveness without changing the concrete syntax. This is the case with Example B where vertical alignment is given a meaning, and in Example D where closeness is given a meaning. Other relations that can be used are overlap, containment, horizontal alignment, clustering, radial alignment, etc. Second, some well-formedness constraints are enforced by the physical world, and so weakening them requires an alternative representation. This is what motivates the vocabulary extension in Example C.

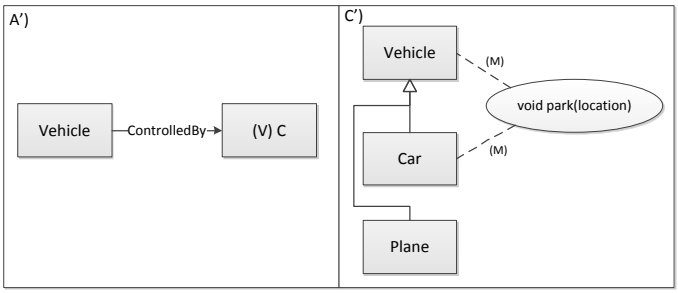

Fig. 3. Using partial modeling to express choice in Examples A and C from Figure 1.

3.2 Partial modeling

When tightening due to information omission yields a choice of alternatives, the modeler may not be comfortable having to choose, either because she doesn't yet know which choice is correct or because she wants to consider all alternatives. In this case, the technique of *partial modeling* allows the modeler to defer the decision and provides an alternative to tightening.

A partial model can express a set of possible models through the use of model annotations and is typically used to express model uncertainty. For example, Figure 3 shows the use of the *MAVO* partial modeling approach to express the choices due to the omission of information in Examples A and C of Figure 1, resulting in A' and C', respectively. The `V` annotation in Example A' means that the class `C` is a "variable" and so this represents a set of different models according to how the variable is instantiated. The `M` annotations indicate "maybe" so Example C' represents the set of models in which only one of the operation ownership relations exists. Due to lack of space, we omit a detailed description of *MAVO* partial modeling – interested readers are directed to [11]. The benefit of using partial models is that the annotations have formal semantics and thus partial models can be used in place of ordinary models in MDE operations such as property checking [11,2] and transformation [3].

4 Towards tool support

Our strategy relies on tool support. In this section, we discuss some of the possibilities for this in terms of existing technologies.

Relaxation. The relaxation tool may be a general drawing tool (e.g., Visio) with a predefined template for the concrete syntax to allow models expressed in the original language to be drawn. The relaxation mechanism of vocabulary extension is achieved by allowing other shapes to be drawn as well. The relaxation mechanism of constraint weakening is achieved by supporting an operating mode that does not enforce (selectable) constraints (e.g., see "soft validation" in

Section 5). Note that physical constraints imposed by the concrete syntax cannot be weakened, so these are addressed by vocabulary extension as in Example C.

Tightening. Constructing the tightening transformation is the more difficult part of the approach. There are two steps involved in the construction:

(1) Identify an occurrence of a language relaxation. Instances of vocabulary extension or constraint weakening (i.e., violation) are easy to detect automatically. Instances of broadening the interpretation may be impossible to detect without the modeler "pointing it out". One clue may be to detect occurrences of spatial relations (e.g., vertical alignment).
(2) Construct the appropriate tightening depending on the type of relaxation:
   – For *weakened constraints*, we must fix model inconsistencies relative to the original constraints. To do this, we can rely on existing computational approaches for computing *minimal model repairs*. The objective here is to search the space of possible changes to the model to find the minimal changes that fix a constraint violation. See Section 5 for a discussion of this work in the literature. If there is still a choice left after the repair process (i.e., there are several possible minimal repairs), then a strategy for dealing with choice must be followed, e.g., to elicit the decision from the modeler, follow a predefined choice policy or use a partial modeling mechanism as described in Section 3.2.
   – For *vocabulary extensions* and other interpretation broadening, a definition of the new elements/information in terms of constructs in the original language must be elicited from the modeler. Clearly, this requires the use of a transformation language for expressing this redefinition, and we rely on existing solutions for this, e.g., ATL[2].

5 Related Work

The use of relaxation to increase agility has been proposed in various contexts. There is a long tradition of work on relaxing the input method by allowing freehand sketching of models. See [8] for a recent example and [5] for a survey. Support for conversion of sketches to the "computer" form of the concrete syntax has been developed in commercial tools and explored in research (e.g., [1]). In contrast to this work, our focus is on agility through the relaxation of the concrete syntax rather than the input method.

Some modeling tools allow "soft validation" where the satisfaction of well-formedness constraints can be deferred until a more appropriate time when the modeler is finished with a unit of work (e.g., see Microsoft modeling tools[3]). This mechanism can be used as a limited form of language relaxation but it only addresses constraint weakening and not vocabulary extension.

[2] http://www.eclipse.org/atl/
[3] http://msdn.microsoft.com/en-us/library/bb245773.aspx

Metamodel relaxation has been used for purposes other than increasing the expressive power of models. For example, in [6], the authors propose a way of automatically constructing a transformation language from the concrete syntax of a modeling language. Since transformation rules must work with non-well-formed model fragments, the creation of the transformation language requires a relaxation of the original language. Both vocabulary extension and constraint weakening are used to achieve this. Further, since transformation languages have a different use than the original language they are based on, language modifications are required as well. In other work, Ramos et. al. [9] use model fragments as a way of specifying model patterns for pattern matching. They use constraint weakening to relax a metamodel so that model fragments (called "model snippets" here) become acceptable instances. Similarly, Levendovszky et. al. [7] use constraint weakening to construct metamodels that allow design patterns to be defined, while Sen et. al. [12] use constraint weakening to define metamodels of models containing partial knowledge in order to support transformation testing. In most of this work, the focus is on creating a metamodel that can accept model fragments as instances, with constraint weakening being the primary mechanism to achieve this. In our work, the goal is richer and hence vocabulary extension plays a larger role.

As discussed in Section 3, the issue of "model tightening" is dependent on mechanisms for model repair. Due to lack of space, we omit a thorough review of work in this area and instead only mention recent examples. Many approaches focus on attempting to formulate repair rules representing various change scenarios where specific repair actions are performed in response to detected changes, e.g., [13]. Others automatically infer the needed repairs directly from the well-formedness constraints and the violation, e.g., [10]. Many of these approaches also handle the elicitation of a decision from the user when a choice of multiple repairs is available. Our tightening transformations can work with either of these techniques.

6 Conclusion

Models are used by humans and programs in different ways, giving rise to what we have called *the agility conflict*: humans require expressive freedom while programs require strict conformance to constraints. In this paper, we outlined the beginnings of a framework to address the agility conflict with a focus on two types of agility: *omission agility* which gives the modeler the freedom to omit information, and *clarity agility* which allows the modeler the ability to rephrase information to improve clarity. Our approach involves relaxing the modeling language to support these types of agility and then constructing a tightening transformation to put the relaxed model back into a form that can be accepted by MDE processes. We explored the approach through a series of examples, discussing its characteristics and potential tool support. Our next steps are to further develop the theoretical details of this approach and prototype tool support for it.

References

1. Th. Buchmann. Towards Tool Support for Agile Modeling: Sketching Equals Modeling. In *Proc. of XM'12 Wksp*, pages 9–14, 2012.
2. M. Famelis, M. Chechik, and R. Salay. Partial Models: Towards Modeling and Reasoning with Uncertainty. In *Proc. of ICSE'12*, 2012.
3. M. Famelis, R.Salay, A. Di Sandro, and M. Chechik. Transformation of Models Containing Uncertainty. In *Proc. of MODELS'13*, 2013.
4. H. P. Grice. Logic And Conversation. In Cole et al., editor, *Syntax and Semantics 3: Speech arts*, pages 41–58. Elsevier, 1975.
5. G. Johnson, M. Gross, J. Hong, and E. Yi-Luen Do. Computational Support for Sketching in Design: a Review. *J. Foundations and Trends in HCI*, 2(1):1–93, 2009.
6. Th. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. Explicit Transformation Modeling. In *Proc. of MODELS'10*, pages 240–255, 2010.
7. T. Levendovszky, L. Lengyel, and T. Mészáros. Supporting domain-specific model patterns with metamodeling. *Software & Systems Modeling*, 8(4):501–520, 2009.
8. N. Mangano, A. Baker, M. Dempsey, E. Navarro, and A. van der Hoek. Software Design Sketching with CALICO. In *Proc. of ASE'10*, pages 23–32, 2010.
9. R. Ramos, O. Barais, and J. Jézéquel. Matching model-snippets. In *Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2007.
10. A. Reder and A. Egyed. Computing Repair Trees for Resolving Inconsistencies in Design Models. In *Proc. of ASE'12*, pages 220–229, 2012.
11. R. Salay, M. Famelis, and M. Chechik. "Language Independent Refinement using Partial Modeling". In *Proc. of FASE'12*, 2012.
12. S. Sen, J. Mottu, M. Tisi, and J. Cabot. Using models of partial knowledge to test model transformations. In *Theory and Practice of Model Transformations*, pages 24–39. Springer, 2012.
13. Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting Automatic Model Inconsistency Fixing. In *Proc. of ESEC/FSE'09*, pages 315–324, 2009.

Pending Evolution of Grammars

Vadim Zaytsev

Software Analysis & Transformation Team,
Centrum Wiskunde & Informatica,
Amsterdam, The Netherlands

Abstract. The classic approach to grammar manipulation is based on instant processing of grammar edits, which limits the kinds of grammar evolution scenarios that can be expressed with it. Treating transformation preconditions as guards poses limitations on concurrent changes of the same grammar, on reuse of evolution scripts, on expressing optionally executed steps, on batch processing and optimization of them, etc. We propose an alternative paradigm of evolution, where a transformation can be scheduled for later execution based on its precondition. This kind of extreme evolution can be useful for expressing scenarios that are impossible to fully automate within the classic or the negotiated transformation paradigms.

1 Introduction

A colloquial expression *'consider it done'* means that the subject of the conversation is either indeed already done, or will be done in the very near future — in either case, the receiver of such a message can rest assured that the subject will take place if it has not already, and is expected to act as if it has indeed happened. The technique of *pending evolution* that we introduce in this paper, is similar to that expression, and the benefits of it are not unlike the subtle differences between considering something done and it having been done.

As it turns out, the pending evolution scheme allows us to efficiently model scenarios of grammar evolution, deployment and maintenance that are impossible to model within the traditional grammar transformation paradigm, which is briefly explained in §2. The method is introduced in §3. Since the most profits hide deep in the details, we spend the rest of the paper (§4) by motivating the use of pending evolution for grammars instead of classical evolution scripts, by concrete examples. §5 concludes the paper by summarizing its contributions and discussing related work.

2 GrammarLab

GrammarLab is a codename for a grammar manipulation project that is currently being migrated from the Software Language Processing Suite[1] initiative

[1] V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, G. Wachsmuth. Software Language Processing Suite, 2008–2013. http://slps.github.io.

to its own repository[2]. It is centered around the concept of a grammar in a broad sense [5], which can be *extracted* by abstracting away the idiosyncratic details that we see on class diagrams, in algebraic data type definitions, object grammars, concrete syntax specifications, database schemata and exposed library interfaces — all these are 'grammars in a broad sense', since they model *commitment to grammatical structure*.

Beside extraction, GrammarLab is good in dealing with programmable grammar *transformations* — a disciplined method of grammar evolution, where every change is expressed as a call to a transformation operator with a well-defined semantics; as well as grammar *mutations* — large scale strategies for changing one simple thing in a priori unknown number of places. GrammarLab also includes library for grammar *analysis* and *metrics*, but they are less relevant for this paper.

3 Pending evolution

In GrammarLab, the evolution of a grammar is specified by a sequence of steps, each referring to a transformation operator or a mutation, with proper parameterization: e.g., first we **rename** a nonterminal, then we **factor** its definition and then **extract** a part of it into a new nonterminal. Each of the possible transformation operators and mutations in the library, have preconditions that determine their applicability and postconditions that demonstrate their successful execution. Whenever a postcondition of a step or a precondition of the next step fails, the transformation sequence is interrupted and an error occurs instead.

When a negotiated transformation paradigm [13] is explored, failure of a pre- or a postcondition means the start of a negotiation: e.g., if a **rename** fails, the engine can propose alternative name pairs that would enable its execution. A clever strategy for negotiations can drastically increase applicability and reuse of a transformation script, while still allow for full automation.

The *pending evolution* paradigm that we propose here, can hold any transformation step pending until its precondition becomes enabled. As will become apparent from the following examples in §4, it is possible to: (1) push the pending steps all the way to the end of the evolution sequence and then disregard them; or (2) leave them forever pending and always ready to be applied any number of times necessary; or (3) relax the constraints about the order of steps; or (4) collect and log the information about all the possibly non-sequential failures in a system; or even intentionally decide that particular steps must be taken by defer their actual execution until later. Only the simplest local cases can be expressed in terms of negotiations. On the other hand, only simplest negotiations can be expressed as pending changes. In short, negotiated transformations enables flexibility with the *outcome* of one step, while pending evolution enables flexibility with the *order* of multiple steps.

[2] V. Zaytsev. GrammarLab, 2013. http://grammarware.github.io/lab.

4 Scenarios

The paradigm of pending evolution probably has much wider applicability, but here we sketch at least four user stories for it, inspired by the problems in the grammarware technological space that can be addressed and solved there.

4.1 Optional execution

In the classic grammar transformation engine of GrammarLab, any grammar transformation step that changes nothing in the grammar (we call them 'vacuous transformations'), is considered erroneous, since in most reasonable contexts — correction, adaptation, evolution, etc — a *change* that *changes* nothing, is meaningless. However, with some negotiated transformation schemes [13], one could find it sensible to ignore the fact that a transformation step brought no actual changes, if considered in a broader context. In particular, consider the following scenarios:

– Suppose we have a repository of grammars, such as the Grammar Zoo[3] [15]. The repository is highly heterogeneous and contains 'grammars in a broad sense' extracted from parser specifications, compiler sources, readable documentation, privately created webpages, community contributed wikis, generated and manually built artifacts. However, one of the steps known from grammar research [8] to increase the quality of a grammar, is resolving all 'bottom' nonterminals — the ones that are used within the grammar but never defined (a grammar with no bottom nonterminals is called a 'level 3 grammar' by Lämmel and Verhoef in [8]). While some definitions are simply missing from the grammar due to development mistakes, quite commonly these are lexical, or character-level, definitions, containing the rules about how an identifier name or a numeric literal should look in a language being defined. A big fraction of these, as becomes apparent after mining Grammar Zoo, have meaningful names such as 'string', 'identifier', 'integer', 'id', etc, and can be matched to a small library of predefined production rules such as 'a string is a symmetrically quoted sequence of one or more characters' or 'an integer value is an optional sign followed by one or more digits, the first of which is not zero'. This can be automated and ran over the whole repository, which can of such substantial size that prevents its manual verification[4] — however, it would be desirable for the framework to introduce the missing definitions only if they are truly missing, and allow individual grammars to retain their specific views of what a string or a boolean looks like. Hence, we allow the **introduce** operator[5] to be left pending, and disregard it at the end of the transformation application.

[3] Grammar Zoo, http://slps.github.io/zoo

[4] Grammar Zoo contains 569 grammars at the day of paper submission.

[5] **Introduce** and other grammar transformation operators are documented at http://github.com/grammarware/slps/wiki/introduce and similar URIs.

– Consider another scenario. In grammarware technological space, there are two most common styles of production rules, that we will traditionally call *horizontal* and *vertical*. A horizontal definition says that the nonterminal N is 'either X or Y or Z', while the vertical one makes three statements that 'N is X', as well as 'N is Y', as well as 'N is Z'. These can be formally proven to be equivalent. There are many exceptions, but most language documents prefer horizontal definitions (e.g., Java Language Specification [3]), while language workbenches tend toward vertical ones (e.g., The Meta-Environment [4]) or make no distinction between them (e.g., Rascal [6]). Some transformation operators also expect their arguments to be either horizontal or vertical, which leads to the evolution scenarios specified in such a way where some of the operator calls are preceded by the calls of **horizontal** or **vertical** operators, while others are not. Obviously, this excessive versatility hinders maintainability and changeability of the transformations. It would be better to write these transformation steps as assertions. For instance, we can specify that the definition must by vertical before we **deyaccify** it, and this step would be optional, requiring no action if the original definition is already vertical.

4.2 Error handling

In GrammarLab, transformations are stopped whenever an error occurs, and an error message is displayed. Within the negotiated paradigm [13], it is possible to negotiate for another outcome. One of the rather complex strategies for achieving that, is the one that skips over the failing transformation step and proceeds with the rest of the script, and then displays all the error messages at the end of the computation. To demonstrate the usefulness of this approach, consider the following detailed scenarios.

– In the context of grammar recovery, suppose that we want to extract several grammars in bulk — they are written in the same style, in the same metalanguage, with some a priori unknown differences between them (perhaps they are different versions or dialects of the same language). After carefully considering one of them, a grammar engineer develops a post-extraction transformation script that makes the grammar maximally connected, adds missing definitions, fixes misspelt nonterminal names and corrects other problems. Naturally, we want to reuse the same transformation script for recovering the rest of these grammars. However, in the traditional setup, most of the automated reuse cases will fail because some of the extracted grammars will have some misspellings already fixed, others will lack the part that concerns the fixes, etc. Advanced error handling (or ignoring) can help greatly with scalability in this case, by skipping over inapplicable fixes, applicability classification, etc.
– Imagine another scenario concerning maintenance of grammar transformation scripts. Suppose that we have several grammars that are being converged

together in multiple steps — e.g., the case study converging six Java grammars found in different editions of the Java Language Specification book, consisted of 1611 transformation steps arranged in 70 different scripts [10]. When an error is spotted in one of the existing steps, or when another step needs to be added in the middle of the transformation chain, or when the order of existing steps needs to be adjusted, it becomes a very labor-intensive task since every failure stops the transformation computation — having the luxury of recovering after a failure noticeably increases debugging capabilities.

4.3 Pending fixes

Both the traditional programmed and the negotiated transformation models delegate the decision about the transformation order to the original script: any transformation step takes place after the one that precedes it in the specification and is followed by the one after it. However, there are situations when we can develop certain transformation scenarios and leave them pending so that they can be executed when (if) the times comes and they become applicable. In general this is useful in case of preserving any kind of normal form properties, but we provide two detailed cases taken from practice:

- Recall the difference between horizontal and vertical definitions that we have explained in the previous section. Suppose that our grammar uses vertical definitions exclusively — this is easy to achieve by grammar transformations or mutations, and easy to validate with a metaprogramming formula or by micropatterns. However, if we would like to specify that the dominance of vertical definitions is not incidental and that we would like to preserve it, it is not possible to express this constraint within the straightforward grammar programming approach. With pending evolution, we could leave the verticalizing mutation pending. Then, if someone introduces a new nonterminal to the grammar, and that nonterminal is found to be horizontal, the mutation becomes enabled, is executed and recharged for next use.
- Grammar recovery is a process of extracting a grammar from an existing software artifact that may not be of perfect quality. Automated grammar recovery methodology [14] is based on a collection of heuristics that are partly configurable and partly inferred from the notation specification. One of such heuristics is splitting composite terminals: for instance, if a terminal like '`);`' is found, it can be broken into two consecutive terminals: '`)`' and '`;`' — simply because the resulting atomic terminals are more helpful for other heuristics (like matching parentheses). A grammar mutation that breaks up composite terminals, can be programmed and left pending, such that under any circumstances that would bring such terminals to the grammar (such as importing another grammar, introducing a new nonterminal definition, folding/unfolding, projecting symbols, etc), it becomes enabled and is immediately fired to split such terminals as desired.

These scenarios are sufficiently different from the ones in the previous section not only in motivation, but also in realization, since we speak of pending mutations (which are large scale transformations) and recharging them after transformation.

4.4 Intentional pending

Since we have discussed preserving the grammar already being in the normal form, another scenario deserves mentioning where the grammar is normalized — or rather, when such a normalizing mutation is left pending. Below there are two use cases for this situation, but any normalization could possibly spawn another one.

- Suppose that we have a grammar written in a specific notation (usually a dialect of EBNF). Suppose also that a notation evolves, and the grammar is required to coevolve in order to preserve conformance to the metalanguage. This scenario is called 'metalinguistic evolution' [12] and has been studied sufficiently to be applied in an automated fashion. One of such applications involves a grammar being exported to a particular notation, which it might not perfectly fit. For instance, the grammar may use an explicit repetition (usually denoted with '∗') or other metaconstructs which are lacking from the notation. Another typical case is that the target metalanguage insists on a particular naming convention for the nonterminal (e.g., all must be written in capitals). In that case, the grammar needs to coevolve with the 'change' of notation from its original one to the one that it is being exported to. However, this coevolution is essentially a part of the exporting process, and as such must always take place after all the other evolution steps. Hence, it can be left pending until the very end of the transformation script, and be executed last, removing the use of excessive metalanguage elements, changing the naming convention and adjusting grammar before the actual export mapping.
- Grammars in a broad sense can be observed in very different environments and extracted from artifacts hailing from different technological spaces: XML schemata, Ecore models, class diagrams, parser specifications, data types, etc. Even when these define one intended language, they are different in many ways. A technique called grammar convergence [9] is used to reverse engineer the real relationships between such grammars: based on expert-written transformation scripts, it can show which grammars define the same language, which define languages that are subsets or supersets of one another, and which are incomparable. It is also possible to automate the creation of such scripts, but the inference algorithm performs best when grammars are in so called 'abstract normal form'. Many constraints of the abstract normal form contradict the practice of grammar engineering, so it would be most desirable to continue working with the non-normalized grammar and then perform the pending normalization right before the guided grammar convergence algorithm is applied. Then, the obtained result can be traced back to the original grammar by reversing the bidirectional transformation chain produced by the normalizer.

5 Concluding remarks

There are some techniques similar to pending evolution in the inconsistency management, most notably with concurrent transformation schemes. Such inconsistencies can be represented as separate first-class entities [2] and incorporated directly to the resulting model [7], which enables efficient handling of inconsistency detection and resolutions as graph transformation rules [11] in a much less extreme way than the one proposed in this paper. The fact that these approaches of inconsistency modeling and resolution are not entirely covered by negotiated grammar transformation, has inspired us to look for common schemes of advanced change impact propagation, importing ideas from modelware to grammarware and adapting them to the domain.

To summarize, we have proposed the following use cases for the technique of pending grammar evolution:

- optional execution (§4.1)
  - optionally complementing the grammar with missing definitions
  - using optional transformations as assertions
- error handling (§4.2)
  - reusing transformations for bulk extraction
  - debugging transformations
- pending fixes (§4.3)
  - persistent commitment to a normal form
  - pending recovery heuristics
- intentional pending (§4.4)
  - pre-export processing
  - pre-convergence normalization

Pending evolution for grammars (either in a broad sense [5] or in the classic sense [1]) has never been considered before. Investigating the impact and opportunities for pending evolution schemes in other fields like program transformation remains future work. In transaction handling domains both of great strictness (such as database management and mainframe job processing) and persistent inconsistency (such as managing wiki contents with a bot) one will be able to find techniques somewhat similar to the one we have proposed here.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, Oct. 2007. TOOLS EUROPE 2007 — Objects, Models, Components, Patterns.
3. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.
4. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF—Reference Manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

5. P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 14(3):331–380, 2005.
6. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2009)*, volume 6491 of *LNCS*, pages 222–289, Berlin, Heidelberg, Jan. 2011. Springer-Verlag.
7. M. Kögel, H. Naughton, J. Helming, and M. Herrmannsdörfer. Collaborative Model Merging. In *Companion of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, SPLASH '10, pages 27–34, New York, NY, USA, 2010. ACM.
8. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, Dec. 2001.
9. R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In M. Leuschel and H. Wehrheim, editors, *Proceedings of the Seventh International Conference on Integrated Formal Methods (iFM 2009)*, volume 5423 of *LNCS*, pages 246–260, Berlin, Heidelberg, Feb. 2009. Springer-Verlag.
10. R. Lämmel and V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal (SQJ)*, 19(2):333–378, Mar. 2011.
11. T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *LNCS*, pages 200–214. Springer, 2006.
12. V. Zaytsev. Language Evolution, Metasyntactically. *Electronic Communications of the European Association of Software Science and Technology (EC-EASST)*, 49, 2012.
13. V. Zaytsev. Negotiated Grammar Transformation. In J. De Lara, D. Di Ruscio, and A. Pierantonio, editors, *Post-proceedings of the Extreme Modeling Workshop (XM 2012)*. ACM Digital Library, Nov. 2012. In print, currently available at http://www.di.univaq.it/diruscio/sites/XM2012/xm2012_submission_11.pdf. An extended version is currently under major revision to the Special issue on Extreme Modeling of The Journal of Object Technology (JOT).
14. V. Zaytsev. Notation-Parametric Grammar Recovery. In A. Sloane and S. Andova, editors, *Post-proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA 2012)*. ACM Digital Library, June 2012.
15. V. Zaytsev. Grammar Zoo: A Repository of Experimental Grammarware. Under major revision for the Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5), 2013.

Language Support for Megamodel Renarration

Ralf Lämmel[1] and Vadim Zaytsev[2]

[1] Software Languages Team, Universität Koblenz-Landau, Germany
[2] Software Analysis & Transformation Team, CWI, Amsterdam, The Netherlands

Abstract. Megamodels may be difficult to understand because they reside at a high level of abstraction and they are graph-like structures that do not immediately provide means of order and decomposition as needed for successive examination and comprehension. To improve megamodel comprehension, we introduce modeling features for the recreation, in fact, renarration of megamodels. Our approach relies on certain operators for extending, instantiating, and otherwise modifying megamodels. We illustrate the approach in the context of megamodeling for Object/XML mapping (also known as XML data binding).

Keywords: megamodeling, linguistic architecture, renarration, software language engineering, XML data binding

1 Introduction

Models (of all kinds) may be difficult to understand when they reside at a high level of abstraction and when they are not structured in a way to serve successive examination and comprehension. In this paper,[3] we are specifically concerned with the modeling domain of the *linguistic architecture* of software systems [5] and a corresponding form of *megamodels* [3]. These are highly abstract models about software systems in terms of the involved languages, technologies, concepts, and artifacts. We aim to improve understanding of such models by means of renarration such that a megamodel is described (in fact, recreated) by a 'story' as opposed to a monolithic, highly abstract graph.

Contribution of this paper We enrich the megamodeling language MegaL [5] with language support for renarration such that megamodels can be developed in an incremental manner, subject to appropriate operators such as 'addition', 'restriction', or 'instantiation', also subject to an appropriate notion of megamodel *deltas*. In previous work [16], we have introduced the notion of renarration of megamodels in an informal manner as the process of converting a collection of facts into a story, also inspired by natural language engineering [15], computer-assisted reporting [13] and database journalism [8]. In this paper, we take the next step: we enrich megamodeling with proper language support for renarration.

[3] The paper's website: http://softlang.uni-koblenz.de/megal-renarration

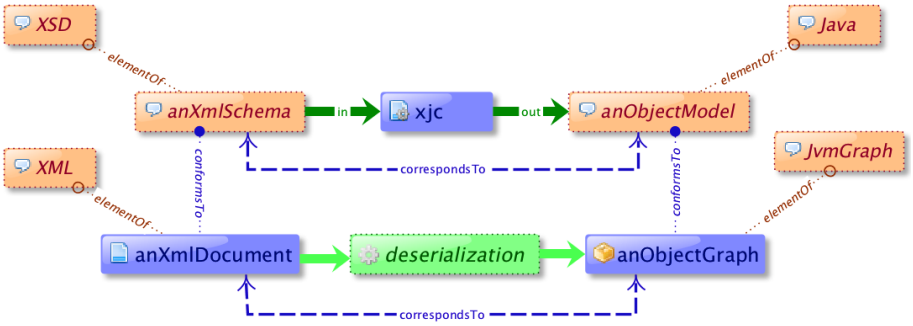

Fig. 1. A megamodel for Object/XML mapping (also known as XML data binding)

Roadmap §2 provides background on megamodeling and motivates the need for renarration. §3 recalls the MegaL language. §4 describes the specific approach to renarration. §5 provides a catalogue of operators that are used to express steps of renarration. §6 validates the approach in the context of megamodeling for Object/XML mapping. §7 discusses related work. §8 concludes the paper.

2 On the need for megamodel renarration

"A megamodel is a model of which [...] some elements represent and/or refer to models or metamodels" [3]—we use this definition by interpreting the notion of (meta)models in a broad sense to include programs, documents, schemas, grammars, etc. Megamodeling is often applied in the context of model-driven engineering while we apply it in the broader software engineering and software development context.

That is, we use megamodels to model the *linguistic architecture* of software systems [5]. By linguistic architecture of software systems or technologies, we mean their architecture expressed in terms of the involved software languages, software technologies, software concepts, software artifacts, and the explicit relationships between all these conceptual and actual entities. In our recent work, we have shown the utility of megamodels for understanding the linguistic architecture of diverse software (language) engineering scenarios [5,16].

Consider Figure 1 for an illustrative megamodel rendered in the visual syntax MegaL/yEd [5]. The nodes represent entities (languages, schemas, tools, etc.). The edges represent relationships ('elementOf', 'conformsTo', 'correspondsTo', etc.). The megamodel sketches basic aspects of Object/XML mapping according to the JAXB technology for XML data binding in the Java platform. Specifically, there is the aspect of deriving an object model (i.e., Java classes) from an XML schema (see the upper data flow in the figure) and the aspect of de-serializing an XML document to an object graph in the JVM (see the lower data flow in the figure).

One impediment to megamodel comprehension is the abstraction level of megamodels. In particular, the role and the origin of the entities as well the meaning of the relationships may not be evident. In recent work [5], we have

proposed a remedy for this problem. Our proposal involves linking megamodel entities and relationships to proper artifacts or extra resources for conceptual entities.

This paper focuses on another impediment to megamodel comprehension: megamodels are essentially just graph-like structures that do not immediately provide means of order and decomposition as needed for successive examination and comprehension. Consider the figure again. The following kinds of questions naturally arise. Where to start 'reading' in the figure? Are there any subgraphs that can be understood independently? Do any of the entities arise as instantiations of more general entities that may be worth mentioning to facilitate understanding?

The latter impediment to comprehension is not unique to megamodeling, of course. Various modeling or specification languages are prone to the same problem. Various remedies exist, e.g., based on modularization, abstraction, refinement, annotation, and slicing. In this paper, we put to work renarration which is indeed inspired by existing ideas on refinement, modularization, and slicing.

In general, renarration is the process of creating different stories while reusing the same facts (cf. narration[4]). In literature, for example, renarration is a technique to create a story by the narrator based on fixed plot elements; the story itself can be adapted to the audience and other circumstances—we refer to [2] for more background information. In megamodeling, renarration is a process of creating stories for the recreation of a megamodel. Recreation may cater for the audience's technical background and interest, time available and yet other factors. In our experience, the process of recreating a megamodel is needed to make megamodels meaningful to humans. Recreation may be interactive, e.g., by renarrating megamodels on the whiteboard, encouraging questions from the audience, and responding to these questions in the continuation of the story. This paper provides language support for the process of renarration.

3 Megamodeling with MegaL

Figure 1 provided a first illustration of the MegaL [5] language for megamodeling. In the rest of the paper, we use the textual MegaL syntax, i.e., MegaL/TXT. A megamodel is a collection of declarations of the following kinds.

Entity declarations A name is introduced for a conceptual entity, an actual entity, or a parameter thereof; an entity type (e.g., *Language* or *File*) is assigned. For instance:

```
Java : Language // "Java" as a language entity
JavaGrammar : Artifact // the "JavaGrammar" as an artifact entity
BNF : Language // "BNF" as a language entity
?aLanguage : Language // parameter "aLanguage" for a language entity
?aProgram : File // parameter "aProgram" for a file entity
```

[4] According to Merriam-Webster: *narration*: the act or process of telling a story or describing what happens http://www.merriam-webster.com/dictionary/narration Visited 14 September 2013.

We speak of a conceptual entity, if it exits in our mind, as in the case of a language. We speak of an actual entity (or simply an artifact), if it is manifest in some way: it exists on the file system (e.g., a language description) or as data structure at runtime (e.g., a parse tree).

Relationship declarations Two declared entities (or parameters thereof) are related by a binary relationship (e.g., 'elementOf' or 'conformsTo'). For instance:

```
aProgram elementOf Java // a program of the Java language
JavaGrammar elementOf BNF // the Java grammar is a BNF−style grammar
JavaGrammar defines Java // the Java grammar defines the Java language
aProgram conformsTo JavaGrammar // a program conforming to the Java grammar
```

Entity-type declarations There is a number of predefined, fundamental entity types, as exercised in the earlier examples, but new entity types can be defined by specialization. For instance:

```
OopLanguage < Language // an entity type for OO programming languages
FpLanguage < Language // an entity type for functional programming languages
```

Relationship-type declarations Likewise, there is a number of predefined, fundamental relationship types, as exercised in the illustrations above, but new relationship types can be defined on predefined as well as explicitly declared entity types. We do not further discuss such expressiveness in this paper.

The declarations simply describe a graph as illustrated in Figure 1. The order of all declarations of a megamodel is semantically irrelevant. The lack of any intrinsic notion of order (as in an imperative setting) or decomposition (as in substitution or function composition in functional programming) feeds into the comprehension challenge to be addressed by renarration. We mention in passing that megamodels have an interesting evaluation semantics. That is, declared relationships may be checked by applying some programmatic relationship-specific check on resources linked to declared entities.

4 Megamodel renarration

We add language support for renarration to the megamodeling language MegaL. We commit to a specific view on renarration such that megamodel *deltas* are used in the recreation of a megamodel through a sequence of steps with each step being effectively characterized by ingredients as follows:

- An informative *label* of the step, also serving as an 'id' for reference.
- The actual *delta* in terms of added and removed declarations (such as entity and relationship declarations). Added declarations are prefixed by '+'; removed declarations are prefixed by '−'. Deltas must preserve well-formedness of megamodels. In particular:
  - Entities are declared uniquely.
  - All entities referenced by relationship declarations are declared.
  - Relationships are applied to entities of suitable types.

Consider the following megamodel (in fact, megamodeling pattern) of a file and a language being related such that the former (in terms of its content) is an element of the latter.

```
[Label="File with language", Operator="Addition"]
+ ?aLanguage : Language // some language
+ ?aFile : File // some file
+ aFile elementOf aLanguage // associate language with file
```

In a next step, let us instantiate the language parameter to actually commit to the specific language *Java*. Thus:

```
[Label="A Java file", Operator="Instantiation"]
+ Java : Language // pick a specific language
+ aFile elementOf Java // associate the file with Java
- ?aLanguage : Language // removal of language parameter
- aFile elementOf aLanguage // removal of reference to language parameter
```

Fig. 2. An illustrative renarration

– An *operator* to describe the intent of the step. Each operator implies specific constraints on the delta, as discussed below.

The steps are interleaved with *informal explanations*.

See Figure 2 for a trivial, illustrative renarration. The first step introduces some entities and relates them. Nothing is removed; thus, the use of the operator 'Addition'. The second step instantiates the megamodel to a more concrete situation. The more general declarations are removed according to the delta and more specific declarations are added; thus, the use of the operator 'Instantiation'. Arguably, the instantiation could be characterized more concisely than by listing the delta, but we like to emphasize the utility of deltas for at least explaining the intended semantics of the renarration operators.

5 Renarration operators

The illustrative renarration of Figure 2 has started to reveal some operators: *addition* and *instantiation*. In this section, we provide a catalogue of operators. In the next section, the operators will be illustrated by a larger renarration.

– *Addition*: declarations are exclusively added; there are no removals. Use this operator to enhance a megamodel through added entities and to constrain a megamodel through added relationships.
– *Removal*: the opposite of *Addition*.
– *Restriction*: net total of addition and removal is such that entities may be restricted to be of more specific types. Also, the set operand of 'elementOf' and the super-set operand of 'subsetOf' relationships may be restricted.
– *Generalization*: the opposite of *Restriction*.

– *ZoomIn*: net total of addition and removal is such that relationships are decomposed to reveal more detail. Consider, for example, the relationship type mapsTo, which is used to expressed that one entity is (was) transformed into another entity. When zooming in, a relationship x mapsTo y could be expanded so as to reveal the function that contributes the pair $\langle x, y \rangle$.
– *ZoomOut*: the opposite of *ZoomIn*.
– *Instantiation*: parameters are consistently replaced by actual entities. We may describe such instantiation directly by a mapping from parameters to entities as opposed to a verbose delta. (A delta is clearly obtainable from such a mapping.)
– *Parameterization*: the opposite of *Instantiation*.
– *Connection*: convert an entity parameter into a *dependent entity*, which is one that is effectively determined by relationships as opposed to being yet available for actual instantiation. Such a dependency often occurs as the result of adding other parameters, e.g., a parameter for the definition of a language. We prefix dependent entity declarations by '!' whereas '?' is used for parameters, as explained earlier.
– *Disconnection*: the opposite of *Connection*.
– *Backtracking*: return to an earlier megamodel, as specified by a label. This may be useful in a story, when a certain complication should only be temporarily considered and subsequent steps should relate again to a simpler intermediate state.

6 An illustrative renarration

We are going to renarrate a megamodel for Object/XML mapping. We begin with the introduction of the XML schema which is the starting point for generating a corresponding object model:

```
[Label="XML schema", Operator="Addition"]
+ XSD : Language // the language of XML schemas
+ ?anXmlSchema : File // an XML schema
+ anXmlSchema elementOf XSD // an XML schema, indeed
```

On the OO side of things, we assume a Java-based object model:

```
[Label="Object model", Operator="Addition"]
+ Java : Language // the Java language
+ ?anObjectModel : File+ // an object model organized in one or more files
+ anObjectModel elementOf Java // a Java−based object model
```

The entities *anXmlSchema* and *anObjectModel* are parameters (see the '?' prefix) in that they would only be fixed once we consider a specific software system. We assume that schema and object model are related to each other in the sense that the former is mapped to ('transformed into') the latter; these two data models also correspond to each other [5].

```
[Label="Schema first", Operator="Addition"]
+ anXmlSchema mapsTo anObjectModel // the schema maps to the object model
+ anXmlSchema correspondsTo anObjectModel // the artifacts are "equivalent"
```

The ‘mapsTo’ relationship is helpful for initial understanding, but more details are needed eventually. Let us reveal the fact that a ‘type-level mapping’ would be needed to derive classes from the schema; we view this as ‘zooming in’: one relationship is replaced in favor of more detailed declarations:

```
[Label="Type−level mapping", Operator="ZoomIn"]
+ ?aTypeMapping : XSD -> Java // a mapping from schemas to object models
+ aTypeMapping(anXmlSchema) |-> anObjectModel // apply function
- anXmlSchema mapsTo anObjectModel // remove too vague mapping relationship
```

It is not very precise, neither is it suggestive to say that type-level mapping results in arbitrary Java code. Instead, we should express that a specific Java *subset* for simple object models (in fact, POJOs for data representation without behavioral concerns) is targeted. Thus, we restrict the derived object model as being an element of a suitable subset of Java, to which we refer here as *OxJava*:

```
[Label="O/X subset", Operator="Restriction"]
+ OxJava : Language // the O/X−specific subset of Java
+ OxJava subsetOf Java // establishing subset relationship, indeed
+ anObjectModel elementOf OxJava // add less liberal constraint on object model
- anObjectModel elementOf Java // remove too liberal constraint on object model
```

We have covered the basics of the type level of Object/XML mapping. Let us look at the instance level which involves XML documents and object graphs (trees) related through (de-)serialization. Let us assume an XML input document for de-serialization which conforms to the XML schema previously introduced:

```
[Label="XML document", Operator="Addition"]
+ XML : Language // the XML language
+ ?anXmlDocument : File // an XML document
+ anXmlDocument elementOf XML // an XML document, indeed
+ anXmlDocument conformsTo anXmlSchema // document conforms to schema
```

The result of de-serialization is an object graph that is part of the runtime state. We assume a language for Java’s JVM-based object graphs. The object graph conforms to the object graph previously introduced:

```
[Label="Object graph", Operator="Addition"]
+ JvmGraph : Language // the language of JVM graphs
+ ?anObjectGraph : State // an object graph
+ anObjectGraph elementOf JvmGraph // a JVM−based object graph
+ anObjectGraph conformsTo anObjectModel // graph conforms to object model
```

De-serialization maps the XML document to the object graph:

```
[Label="Instance−level mapping", Operator="Addition"]
+ ?aDeserializer : XML -> JvmGraph // deserialize XML to JVM graphs
+ aDeserializer(anXmlDocument) |-> anObjectGraph // map via deserializer
```

At this point, the mappings both at type and the instance levels (i.e., *aTypeMapping* and *aDeserializer*) are conceptual entities (in fact, functions) without a trace of their emergence. We should manifest them in relation to the underlying mapping technology. We begin with the type level.

```
[Label="Code generator", Operator="Addition"]
```

```
+ ?anOxTechnology : Technology // a technology such as JAXB
+ ?anOxGenerator : Technology // the generation part
+ anOxGenerator partOf anOxTechnology // a part, indeed
```

By relating generator and type mapping, we stop viewing the (conceptual entity for the) mapping as a proper parameter; rather it becomes a dependent entity.

```
[Label="Dependent type−level mapping", Operator="Connection"]
+ anOxGenerator defines aTypeMapping // mapping defined by generator
+ !aTypeMapping : XSD -> Java // this is a dependent entity now
- ?aTypeMapping : XSD -> Java // Ditto
```

Likewise, de-serialization is the conceptual counterpart for code that actually constructs and runs a de-serializer with the help of a designated library, which is another part of the mapping technology:

```
[Label="O/X library", Operator="Addition"]
+ ?anOxLibrary : Technology // the O/X library
+ anOxLibrary partOf anOxTechnology // an O/X part
+ ?aFragment : Fragment // souce code issuing de−serialization
+ aFragment elementOf Java // source code is Java code
+ aFragment refersTo anOxLibrary // use of O/X library
```

Again, we eliminate the parameter for the de-serializer:

```
[Label="Dependent instance−level mapping", Operator="Connection"]
+ aFragment defines aDeserializer // fragment "constructs" de−serializer
+ !aDeserializer : XML -> JvmGraph // this is a dependent entity now
- ?aDeserializer : XML -> JvmGraph // Ditto
```

Let us instantiate the mapping technology and its components to commit to the de-facto platform standard: JAXB [9]. We aim at the following replacements of parameters by concrete technology names:

```
[Label="JAXB", Operator="Instantiation"]
anOxTechnology => JAXB // instantiate parameter ... as ...
anOxGenerator => JAXB.xjc // ditto
anOxLibrary => JAXB.javax.xml.bind // ditto
```

Thus, we use qualified names for the component technologies of JAXB, thereby reducing the stress on the global namespace. We omit the the lower level meaning of the instantiation in terms of a delta.

Let us now generalize rather than instantiate. To this end, we first backtrack to an earlier state—the one before we instantiated for JAXB:

```
[Label="Dependent instance−level mapping", Operator="Backtracking"]
```

Now we can generalize further by making the language a parameter of the model. (Again, we show the concise mapping of actual entities to parameters as opposed to the delta for all the affected declarations.)

```
[Label="Beyond Java", Operator="Parameterization"]
Java => anOopLanguage // replace ... by parameter ...
OxJava => anOxLanguage // ditto
```

Arguably, we should use more specific entity types to better characterize some of the parameters of the model. For instance, the intention of the language

parameter to be an OOP language is only hinted at with the parameter's name; we could also designate and reference a suitable entity type:

```
[Label="Taxonomy", Operator="Restriction"]
+ OopLanguage < Language // declare entity type for OOP languages
+ ?anOopLanguage : OopLanguage // limit entity type of language
+ ?anOxLanguage : OopLanguage // limit entity type of language
- ?anOopLanguage : Language // remove underspecified declaration
- ?anOxLanguage : Language // remove underspecified declaration
```

7 Related work

In the presentation of actual megamodels, e.g., in [5,6,7,12,14], arguably, elements of renarration appear, due to the authors' natural efforts to modularize their models, to relate them, and to develop and describe them in piecemeal fashion. Renarration as an explicit *presentation* technique in software engineering was introduced in previous work [16]. Renarration as an explicit *modeling* technique is the contribution of the present paper.

It may seem that the required language support is straightforward, if not trivial. For instance, one may compare delta-based megamodel recreation with language support for model construction (creation), e.g., in the context of executable UML, as supported by action languages [11]. However, the renarration operators are associated with diverse constraints, as hinted at in their description, which brings them closer to notions such as refactoring or refinement or, more generally, model evolution. Deltas are used widely in model evolution; see, for example, [4]. In this context, it is also common to associate low-level deltas, as observed form the change history, with high-level intents in the sense of model-evolution operators.

A more advanced approach to the renarration of megamodels may receive inspiration from, for example, model management in MDE with its management operators (e.g., for composition [1]) and grammar convergence [10] with its rich underlying operator suite of (in this case) grammar modifications.

The field of natural language engineering contains many problems such as deriving a syuzhet from a fabula, a plot from genre elements, or a story from a plot. Recent solutions to these problems are advanced, formal and automated [15], and can be reused for software language engineering to facilitate semi-automatic or genetic inference of megamodel renarrations based on given constraints.

8 Concluding remarks

We have introduced language support for renarrating megamodels. With a relatively simple language design, we have made it possible to recreate (renarrate) megamodels in an incremental manner, while expressing intents by means of designated operators along the way.

In future work, we plan to provide a precise semantics of the operators. Further, by applying renarration to a number of different megamodeling scenarios, we also hope to converge on the set of operators needed in practice. Deltas,

as such, are fully expressive to represent any sort of recreation, but the suite of operators needs to be carefully maintained to support enough intentions for convenient use and useful checks on the steps. Yet another interesting area of future work is the animation of renarrations for a visual megamodeling language; we use the visual approach already informally on the whiteboard. Finally, the improvement of megamodel comprehension through renarration should be empirically validated.

References

1. A. Anwar, T. Dkaki, S. Ebersold, B. Coulette, and M. Nassar. A Formal Approach to Model Composition Applied to VUML. In *Proc. of ICECCS 2011*, pages 188–197. IEEE, 2011.
2. M. Baker and A. Chesterman. Ethics of Renarration. *Cultus*, 1(1):10–33, 2008. Mona Baker is interviewed by Andrew Chesterman.
3. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. *OOPSLA & GPCE, Workshop on best MDSD practices*, 2004.
4. A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, 2007.
5. J.-M. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proc. of MODELS 2012*, volume 7590 of *LNCS*, pages 151–167. Springer, 2012.
6. J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *ENTCS*, 127(3), 2004.
7. R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. Realizing Architecture Frameworks Through Megamodelling Techniques. In *Proc. of ASE 2010*, pages 305–308. ACM, 2010.
8. A. Holovaty. A Fundamental Way Newspaper Sites Need to Change, Sept. 2006. http://www.holovaty.com/writing/fundamental-change/.
9. JCP JSR 31. JAXB 2.0/2.1 — Java Architecture for XML Binding, 2008. http://jaxb.dev.java.net/.
10. R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In *Proc. of IFM 2009*, volume 5423 of *LNCS*, pages 246–260. Springer, 2009.
11. C.-L. Lazar, I. Lazar, B. Pârv, S. Motogna, and I. G. Czibula. Using a fUML Action Language to Construct UML Models. In *Proc. of SYNASC 2009*, pages 93–101. IEEE Computer Society, 2009.
12. B. Meyers and H. Vangheluwe. A Framework for Evolution of Modelling Languages. *Science of Computer Programming*, 76(12):1223–1246, 2011.
13. L. C. Miller. *Power Journalism: Computer-Assisted Reporting*. Harcourt Brace College Publishers, 1997.
14. J.-S. Sottet, G. Calvary, J.-M. Favre, and J. Coutaz. Megamodeling and Metamodel-Driven Engineering for Plastic User Interfaces: MEGA-UI. In *Human-Centered Software Engineering*, pages 173–200. Springer, 2009.
15. K. Wang, V. Q. Bui, and H. A. Abbass. Evolving Stories: Tree Adjoining Grammar Guided Genetic Programming for Complex Plot Generation. In *Proc. of SEAL 2010*, pages 135–145. Springer, 2010.
16. V. Zaytsev. Renarrating Linguistic Architecture: A Case Study. In *Proc. of MPM 2012*, pages 61–66. ACM, 2012. http://dx.doi.org/10.1145/2508443.2508454.

An Approach for Efficient Querying of Large Relational Datasets with OCL-based Languages

Dimitrios S. Kolovos, Ran Wei, and Konstantinos Barmpis

Department of Computer Science, University of York,
Deramore Lane, York, YO10 5GH, UK
{dimitris.kolovos, rw542, kb634}@york.ac.uk

Abstract. Relational database management systems are used to store and manage large sets of data, subsets of which can be of interest in the context of Model Driven Engineering processes. To enable seamless integration of information stored in relational databases in an MDE process, the technical and conceptual gap between the two technical spaces needs to be bridged. In this paper we investigate the challenges involved in querying large relational datasets using an imperative OCL-based transformation language (EOL) through a running example, and we propose solutions for some of these challenges.

1 Introduction

Information that can potentially be of interest in the context of a Model Driven Engineering process is often located within non-model artefacts such as spreadsheets, XML documents and relational databases. As such, model management languages and tools would arguably benefit from extending their scope beyond the narrow boundaries of 3-level metamodelling architectures such as EMF and MOF for MDE.

In previous work, we have demonstrated how OCL-based model management (e.g. model validation, model-to-text and model-to-model transformation) languages of the Epsilon platform [1] can be used to interact with plain XML documents [2] and spreadsheets [3]. In this work we investigate the challenges involved in using such languages to query large relational datasets and extract abstract models that can be then used (e.g. analysed, validated, transformed) in the context of MDE processes. In particular, we identify the challenges imposed by the size of such datasets and the conceptual gap between the organisation of relational databases and the object-oriented syntax of OCL-based languages, and we propose some solutions.

The rest of the paper is organised as follows. In Section 2 we present a running example that involves querying a real-world large relational dataset and extracting an EMF model from it using an OCL-based imperative transformation language, we identify the performance challenges involved in doing so, and propose a run-time query translation approach that addresses some of these challenges. In Section 3, we review previous work on using OCL to query relational datasets and compare our approach to it, and in Section 4 we conclude the paper and provide directions for further work.

2 Querying Large Relational Datasets: Challenges and Solutions

The Epsilon Object Language [4] is an OCL-based imperative model query and transformation language. EOL is the core language of the Epsilon platform and underpins a number of task-specific languages for model management tasks including model validation, model-to-model and model-to-text transformation. As such, by adding support for querying relational datasets to EOL, this capability is automatically propagated to all task-specific languages of the platform. While the discussion in the rest of the paper focuses on EOL, in principle the discussion and solutions proposed are also relevant to a wide range of OCL-based model management languages such as QVTo, ATL and Kermeta.

To experiment with querying relational datasets with EOL, we selected a large real-world publicly-available dataset from the US Bureau of Transportation Statistics[1] that records all domestic flights in the US in January 2013. The dataset consists of one table (Flight) with 223 columns and 506,312 rows and is 221MB when persisted in MySQL. Each row of the table records the details of a domestic flight during that month, including the short codes of its origin and destination airports, the flight's departure and arrival time etc. An excerpt of the Flight table appears in Figure 1.

Our aim in this running example is to transform this dataset into an EMF model that conforms to the metamodel of Figure 2 and which captures the incoming and outgoing routes for each airport as well as the volume of traffic on these routes, so that we can then further process the EMF model to discover interesting facts about the structure of the US airport network.

origin	dest	depTime	arrTime	...
ABE	ATL	1557	1812	...
ABQ	BWI	0735	1252	...
ANC	ADQ	0804	0915	...
AZA	DEN	1556	1731	...
...	...	...	...	...

Fig. 1. Excerpt of the Flight table

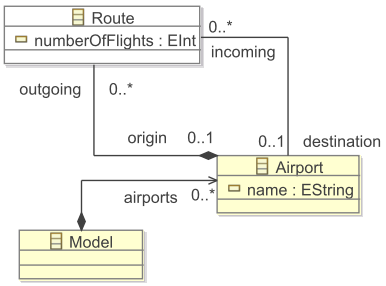

Fig. 2. Simple ATM System Metamodel

[1] http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time

2.1 Finding the number of airports in the network

A reasonable OCL-like expression[2] that can be used to retrieve the number of all distinct airports[3] in the Flight table would be:

```
Flight.allInstances.origin.asSet().size()
```

When such an expression is evaluated against an in-memory model (e.g. an EMF model) the EOL execution engine performs the following steps:

1. It inspects the model and computes a collection of all model elements of type Flight;
2. It iterates through the contents of the collection computed in step 1 and collects the values of their *origin* properties in a new collection;
3. It removes all duplicates from the collection computed in step 2;
4. It computes the size of the collection computed in step 3.

To evaluate the same expression against the relational database discussed above, we can assume that each table in the database is a type and each row in the table is a model element that is an instance of that type. Under these assumptions, the following issues emerge:

1. To compute the *Flight.allInstances* collection, the engine needs to execute the following SQL query: *select * from Flight*. Due to the size of the Flight table, the returned result-set cannot fit in a reasonable amount of heap space (we experimented with up to 1GB), and as such it needs to be streamed from the database to the engine instead. Streamed result-sets demonstrate the following challenges:
   – They support forward-only iteration;
   – To calculate the size of a streamed result-set it needs to be exhaustively iterated (in which case it becomes unusable as only forward iteration is permitted);
   – Each database connection cannot support more than one streamed result-sets at a time.
2. Iterating through all rows of the Flight table through a streamed result set and collecting the values of the *origin* column of each row is particularly inefficient given that the same result can be achieved at a fraction of the time using the following SQL statement: *select origin from Flight*;
3. Eliminating duplicates by iterating the collection computed in step 2 is also inefficient as the same result can be achieved using the following – more efficient – SQL statement: *select distinct origin from Flight*;
4. Finally, calculating the size of a streamed result-set is not trivial without invalidating the result-set itself. By contrast, this could be computed in one step using the following SQL statement: *select count(distinct origin) from Flight*.

[2] EOL does away with the ocl- prefixes (e.g. oclAsSet()) and the → OCL operator and uses . instead for all property/method calls.

[3] We assume that there are no airports with only incoming or outgoing flights and as such, looking into one of *origin*, *dest* should suffice.

2.2 Finding adjunct airports

Assuming that we have computed a set containing the short codes of all airports in the table, the next task is to find for each airport, which other airports are directly connected to it, and then compute the volume of traffic between each pair of adjunct airports. An imperative EOL program that can be used to achieve this follows:

```
1  var origins = Flight.allInstances.origin.asSet();
2  for (origin in origins) {
3    var destinations = Flight.allInstances.dest.asSet();
4    for (destination in destinations) {
5      var numberOfFlights = Flight.allInstances.
6        select(f|f.origin = origin and f.dest = destination).
7          size();
8    }
9  }
```

The following observations can be made for the program above:

- Although the *destinations* result-set computed in line 3 does not change, it needs to be re-computed for every nested iteration as the result of the computation is streamed, and therefore only permits forward navigation;
- Unless care is taken to evaluate the right-hand side expressions in lines 1 and 3 using different database connections, the program will fail (as discussed above, each MySQL connection only permits at most one streamed result-set at a time);
- Iterating through all the rows of the Flight table in the *select(. . .)* method in lines 5-7 is inefficient, particularly as the same result can be computed using the following SQL statement *select count(*) from Flight where origin=? and destination=?* (where *?* should be replaced every time with the appropriate origin/destination values).

2.3 Runtime SQL Query Generation

In this section we argue that while the naive way of evaluating OCL-like queries on relational datasets can dramatically degrade performance (as shown in the previous section), there are certain runtime optimisations that the execution engine can perform to significantly reduce the execution time and memory footprint of some types of queries.

After applying such optimisations, the following EOL transformation, can transform the complete dataset (DB) in question to an EMF-based model (ATMS) that conforms to the metamodel of Figure 2 in less than 45 seconds on average hardware[4]. A visualisation of an excerpt of the extracted model appears in Figure 3.

The functionality of the transformation is outlined below:

- In line 1 it creates a new instance of the *Model* EClass in the ATMS EMF (target) model;

[4] CPU: 2.66 GHz Intel Core 2 Duo, RAM: 8 GB DDR3.

- In line 2 it computes a set of all origin airports in the Flight table;
- In line 4 it iterates through the set of strings computed in line 2;
- In line 5 it invokes the *airportForName* method defined in lines 21-30 which returns an instance of the *Airport* EClass in the target model with a matching name;
- In lines 6-7 it computes a set of adjunct airports to the origin airport;
- In lines 10-12 for each adjunct airport (destination), it computes the number of flights between the two airports;
- In lines 13-16 it creates a new instance of the *Route* EClass in the target model and populates its origin, destination and numberOfFlights properties.
- The *airportForName()* method in lines 21-30 is responsible for preventing the creation of airports with duplicate names in the target model.

```
1  var m : new ATMS!Model;
2  var origins = DB!Flight.allInstances.origin.asSet();
3
4  for (origin in origins) {
5    var originAirport = airportForName(origin);
6    var destinations = DB!Flight.allInstances.
7      select(f|f.origin = origin).dest.asSet();
8
9    for (destination in destinations) {
10     var numberOfFlights = DB!Flight.allInstances.
11       select(f|f.origin = origin and f.dest = destination)
12         .size();
13     var route = new ATMS!Route;
14     route.origin = originAirport;
15     route.destination = airportForName(destination);
16     route.numberOfFlights = numberOfFlights.asInteger();
17   }
18
19 }
20
21 operation airportForName(name : String) {
22   var airport = ATMS!Airport.allInstances.
23     selectOne(a|a.name = name);
24
25   if (airport.isUndefined()) {
26     airport = new ATMS!Airport;
27     airport.name = name;
28     m.airports.add(airport);
29   }
30   return airport;
31 }
```

Listing 1.1. EOL transformation

To achieve an acceptable level of performance, we have extended the EOL execution engine to use streamed *lazy collections* and a runtime OCL to SQL query translation strategy for certain types of OCL expressions when the latter are evaluated against relational datasets. Each lazy collection acts as a wrapper for an SQL query generated at runtime and only starts streaming data from the database if/when it needs to be iterated. This prevents unnecessary database queries and enables multi-step query translation at runtime. An example of the query translation process is illustrated in Figure 4 which calculates the average delay of flights flying from JFK to LAX on Sundays. In particular, the following OCL expressions are rewritten as SQL queries.

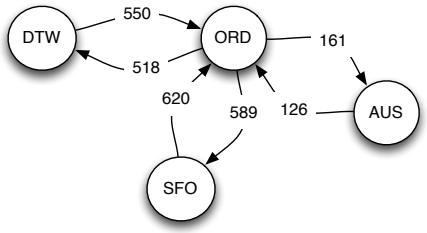

Fig. 3. Visualisation of an excerpt of the model extracted using the transformation in Listing 1.1

.allInstances Retrieving all the rows of a table in the database returns a streamed lazy collection (*ResultSetList*) that is backed by a *select * from <table>* SQL expression. For example, *Flight.allInstances* is translated to *select * from Flight*.

.select(<iterator>|<condition>) *ResultSetList* overrides the built-in OCL select operation, translates the EOL condition to an SQL expression, and returns a new *ResultSetList* constrained by the latter. For example, *Flight.allInstances.select(f|f.origin = "JFK" and f.dayOfWeek=1)* is translated into *select * from Flight where origin = ? and dayOfWeek = ?* (the values of the parameters – i.e. JFK and 1 – are kept separately and are only used if the query needs to be executed). The condition can contain references to the columns of the table, arithmetic, and logical operators at arbitrary levels of nesting. The *exists()*, *forAll()* and *reject()* OCL operations behave similarly.

.collect(<iterator>|<expression>) *ResultSetList* overrides the built-in OCL collect() operation to return a streamed lazy collection of primitive values (*PrimitiveValuesList*). For example, *Flight.all.collect(f|f.dest + "-" + f.origin)* is translated to *select origin + "-" + dest from Flight*. In the spirit of OCL, retrieving properties of collections is a short-hand notation for collect(). For example, *Flight.allInstances.dest* is shorthand for *Flight.allInstances.collect(f|f.dest)*.

.size() Calls to the size() method of a *ResultSetList*/*PrimitiveValuesList* are interpreted as *count* SQL queries. For example *Flight.allInstances.size()* is translated to *select count(*) from Flight*.

asSet() Calls to asSet() method of a *PrimitiveValuesList* return a new *PrimitiveValuesList* backed by a *distinct* SQL query. For example, *Flight.allInstances.dest.asSet()* returns a new *PrimitiveValuesList* backed by the following SQL query: *select distinct(dest) from Flight*.

Streamed lazy collections also provide a *fetch()* method that executes their underpinning query and returns a complete in-memory result-set which is navigable in both directions. This is useful for small result-sets where the overhead of maintaining the entire result-set is memory is preferable to the performance overhead of streaming. To address the limitation of one streamed result-set per

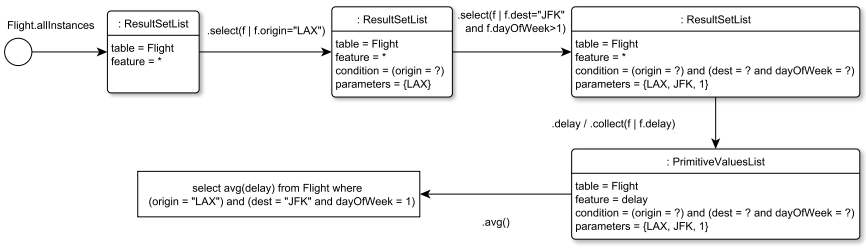

Fig. 4. Multi-step SQL query generation process

connection, we are using a pool of connections for streamed result-sets: each streamed result-set requests a connection from the pool when it needs to be computed and returns it back to the pool when it has been fully iterated.

3 Related Work

Several researchers have proposed solutions for translating OCL to SQL. For example, in [5], the authors demonstrate an approach for generating event-condition-action (ECA) rules comprising SQL triggers and procedures from OCL constraints attached to a UML class diagram, when the latter is translated into a relational schema. In [6], the authors propose using OCL-derived views in relational databases designed using UML, to check the integrity of the persisted data. In this work each OCL constraint is translated into a view in the relational database that contains reports of integrity violations. Such violations can be handled using different strategies including rolling back the offending transaction, triggering a data reconciliation action, or simply reporting the violation to application users. This approach has been implemented in the context of the OCL2SQL prototype[5]. A similar approach is proposed by the authors of [7]. In [8], the authors propose a framework for translating OCL invariants into multiple query languages including SQL and XQuery using model-to-text transformations.

All the approaches above propose compile-time translation of OCL to SQL. By contrast, our approach proposes run-time generation and lazy evaluation of SQL statements. While compile-time translation is feasible for side-effect free OCL constraints that are evaluated against a homogeneous target (e.g. a relational database), for use-cases that involve querying and modifying models conforming to different technologies (e.g. a relational database and an EMF model), this approach is not applicable. Another novelty of the approach proposed in this paper is that it does not require a UML model that specifies the schema of the database, and as such, it can be used on existing databases that have not been developed in a UML-driven manner.

[5] `http://dresden-ocl.sourceforge.net/usage/ocl22sql/`

4 Conclusions and Further Work

In this paper we have argued that it is important for model management languages to extend their scope beyond the narrow boundaries of 3-level metamodelling architectures such as MOF and EMF. In this direction, we have experimented with using an OCL-based imperative transformation language to query data stored in relational databases. We have reported on the identified challenges and proposed an approach for improving the performance of some types of queries using run-time query translation.

In future iterations of this work, we plan to investigate the extent to which compile-time static analysis and query rewriting can deliver additional benefits in terms of performance. An obvious target is to use static analysis to limit the number of columns returned by queries by excluding any columns that are never accessed in the model management program, but additional optimisations are also envisioned. We also plan to investigate supporting queries spanning more than one tables by exploiting foreign keys.

Acknowledgements

This research was part supported by the EPSRC, through the Large-Scale Complex IT Systems project (EP/F001096/1) and by the EU, through the Automated Measurement and Analysis of Open Source Software (OSSMETER) FP7 STREP project (318736).

References

1. Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, Fiona A.C. Polack. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Proc. 14th IEEE International Conference on Engineering of Complex Computer Systems*, Potsdam, Germany, 2009.
2. Dimitrios S. Kolovos, Louis M. Rose, Nicholas Matragkas, James Williams, Richard F. Paige. A Lightweight Approach for Managing XML Documents with MDE Languages. In *Proc. 8th European Conference on Modeling Foundations and Applications*, Copenhagen, Denmark, July 2012.
3. Martins Francis, Dimitrios S. Kolovos, Nicholas Matragkas, Richard F. Paige. Adding Spreadsheets to the MDE Toolbox. In *Proc. ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Miami, USA, October 2013.
4. Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.
5. D. Berrabah and F. Boufares. Constraints checking in uml class diagrams: Sql vs ocl. In Roland Wagner, Norman Revell, and Ganther Pernul, editors, *Database and Expert Systems Applications*, volume 4653 of *Lecture Notes in Computer Science*, pages 593–602. Springer Berlin Heidelberg, 2007.

6. Birgit Demuth, Heinrich Hussmann, and Sten Loecher. Ocl as a specification language for business rules in database applications. In *Proc. 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, UML '01, pages 104–117, London, UK, UK, 2001. Springer-Verlag.
7. U. Marder, N. Ritter, H.-P. Steiert. A DBMS-based Approach for Automatic Checking of OCL Constraints. In *Proc. "Rigourous Modeling and Analysis with the UML: Challenges and Limitations, OOPSLA workshop*, 2009.
8. Heidenreich, F. and Wende, C. and Demuth, B. A Framework for Generating Query Language Code from OCL Invariants. *Electronic Communication of the European Association of Software Science and Technology*, 9, 2007.