Proceedings of the DEMONSTRATIONS track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (Models 2014)

Valencia, Spain, October 1st and 2nd, 2014

Edited by

Tao Yue, Simula Research Laboratory

Benoit Combemale, IRISA, University of Rennes1, France

Table of Contents

Preface **3**
Tao Yue and Benoit Combemale

Bridging Java Annotations and UML Profiles with JUMP **5**
Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer and Gerti Kappel

DPLfw: a Framework for the Product-Line-Based Generation of Variable Content Documents **10**
Abel Gómez, Pau Martí, Ma. Carmen Penadés, José H. Canós

Combining Textual and Web-Based Modeling **15**
Martin Haeusler, Matthias Farwick and Thomas Trojer

Tool support for Collaborative Software Quality Management **20**
Philipp Kalb and Ruth Breu

DSLFORGE: Textual Modeling on the Web **25**
Amine Lajmi, Jabier Martinez and Tewfik Ziadi

Umple: An Open-Source Tool for Easy-To-Use Modeling, Analysis, and Code Generation **30**
Timothy Lethbridge

Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools **35**
Michaela Rindt, Timo Kehrer and Udo Kelter

From Pen-and-Paper Sketches to Prototypes: The Advanced InteractionDesign Environment **40**
Harald Störrle

Concern-Driven Software Development with jUCMNav and TouchRAM **45**
Nishanth Thimmegowda, Omar Alam, Matthias Schöttle, Wisam Al Abed,
Thomas Di'Meco, Laura Martellotto, Gunter Mussbacher, Jörg Kienzle

Preface

Model-driven engineering (MDE) and modeling in general has matured substantially over the past decade and is increasingly finding its way into industrial practice. As a consequence, it is now more important than ever to demonstrate the value of modeling and MDE through research prototypes as well as polished tools covering a broad variety of modeling-related activities.

Therefore, MoDELS 2014 features a track of tool demonstration sessions, in parallel with other conference sessions, to serve as a channel for recognizing and appreciating teams building tools and also a venue for sharing experience of building and applying tools both in academia and industry. This track constitutes the premier venue for researchers and practitioners to showcase their projects and tools in areas that are relevant to the MODELS community. We encouraged submissions, which are either early research prototypes or mature tools not yet been commercialized. Innovative uses of existing tools and tool chains from industrial practice were also included in the scope of this year's demonstration track.

The track welcomed any submission within the scope of the topics of interest of the main conference, including (but not limited to) modeling tools supporting one or more model-based activities in different phases of software/system development lifecycles, domain specific language modeling environment, automated solutions for integrating different modeling tools.

Demonstrations were selected on the basis of technical merit, novelty, and relevance to the MoDELS community. All the submitted proposals were peer-reviewed by at least three reviewers. Nine accepted demonstrations were individually presented by technical members of the team, and focused on technical content and practical issues of modeling tools and environment, analysis and model management. Each paper in this volumn contains a link to a video giving a small insight into the actual demonstration.

We would like to thank the authors for submitting their papers to the demonstration track. We are also grateful to the members of the Selection Committee for their efforts in the reviewing process and to the MoDELS 2014 organizers for their support and assistance during the demonstration organization.

September 2014

Tao Yue (tao@simula.no),
Simula Research Laboratory,
Oslo, Norway
Benoit Combemale
(benoit.combemale@irisa.fr),
University of Rennes 1 and
Research Scientist at Inria,
France

Program Committee

Shaukat Ali	Simula Research Laboratory
Benoit Baudry	INRIA
Benoit Combemale	IRISA, Université de Rennes 1
Juan De Lara	Universidad Autonoma de Madrid
Robert France	Colorado State University
Sebastien Gerard	CAE LIST
Jeff Gray	University of Alabama
Jörg Kienzle	McGill University
Bran Selic	Malina Software Corp.
Juha-Pekka Tolvanen	MetaCase
Antonio Vallecillo	Universidad de Málaga
Marc-Florian Wendland	Fraunhofer Institut FOKUS
Tao Yue	Simula Research Laboratory
Steffen Zschaler	King's College London

Bridging Java Annotations and UML Profiles with JUMP*

Alexander Bergmayr[1], Michael Grossniklaus[2], Manuel Wimmer[1], and Gerti Kappel[1]

[1] Vienna University of Technology, Austria
{bergmayr,wimmer,kappel}@big.tuwien.ac.at
[2] University of Konstanz, Germany
michael.grossniklaus@uni-konstanz.de

Abstract. UML profiles support annotations at the modeling level. However, current modeling tools lack the capabilities to generate such annotations required for the programming level, which is desirable for reverse engineering and forward engineering scenarios. To overcome this shortcoming, we defined an effective conceptual mapping between Java annotations and UML profiles as a basis for implementing the *JUMP* tool. It automates the generation of profiles from annotation-based libraries and their application to generate profiled UML models. In this demonstration, we (i) compare our mapping with the different representational capabilities of current UML modeling tools, (ii) apply our tool to a model-based software modernization scenario, and (iii) evaluate its scalability with real-world libraries and applications.

1 Introduction

The value of UML profiles is a major ingredient for model-based software engineering [3] as they provide features supplementary to the UML metamodel in terms of lightweight extensions. This powerful capability of profiles can be employed as annotation mechanism [9], where stereotypes show similar capabilities as annotations in Java. In the ARTIST project [1], we exploit these capabilities, as we work towards a model-based engineering approach for modernizing applications by novel cloud offerings, which involves representing platform-specific models (PSM) that refer to the platform of existing applications, e.g., the Java Persistence API (JPA)[3], when considering persistence, and the platform of "cloudified" applications, e.g., Objectify[4], when considering cloud datastores. Clearly, the modernization process relies on the availability of the profiles that correspond to the used Java libraries.

Manually developing a rich set of profiles demands a huge effort when considering the large number of possible annotations in Java. To automate the generation of UML profiles requires an effective conceptual mapping of the two languages. In recent work, we defined such a mapping [2] based on which we implemented the *JUMP* tool. Hence, we continue with the long tradition of investigating mappings between Java and

* This work is co-funded by the European Commission, grant no. 317859.
[3] http://oracle.com/technetwork/java/javaee/
[4] https://code.google.com/p/objectify-appengine/

UML [7, 8], though in this work we also consider Java annotations in the mapping.

The *JUMP* tool is intended to be used by (i) developers that produce platform-specific profiles to support transformations for reverse engineering and forward engineering scenarios or to enable platform-independent profiles abstracted from such platform-specific profiles and (ii) modelers that directly use the produced profiles to document important design decisions at the modeling level or to easier understand Java libraries by visualizing provided annotations in terms of UML profile diagrams.

In this demonstration, we discuss the benefits of *JUMP* compared to existing solutions of modeling tools that support annotations. Furthermore, we emphasize the unique capabilities of the *JUMP* tool to automatically generate profiles from annotation-based libraries, which are collected in the *UML-Profile-Store*. It leverages the generation of profiled models from applications. To report on the scalability, we measured the performance of *JUMP* tool by applying it to large Java code bases.

2 Bridging Java Annotations and UML: Profiles to the Rescue

UML profiles enable systematically introducing new language elements [5] without the need to adapt the underlying modeling environment, such as editors, model transformations, and model APIs [6]. UML provides a dedicated language to precisely define profiles and how stereotypes are applied on models. Similarly, Java provides an annotation language to declare annotation types that can be applied on the targeted code elements. Figure 1 demonstrates the relationship between the two languages based on the Objectify framework. On the left side, the *application* of annotation types, among them `Cache`, to the `Customer` class and the respective *declaration* of the `Cache` annotation type is shown. The corresponding UML-based representation shown on the right side demonstrates the stereotype application to the `Customer` class and the declaration of `Cache` by a `Stereotype`, which is part of the Objectify profile. To ensure that the `Cache` stereotype provides at least similar capabilities as the corresponding annotation type, the extension relationship references the UML meta-class `Type`. The Objectify profile generated by the *JUMP* tool enables modelers to refine UML class di-

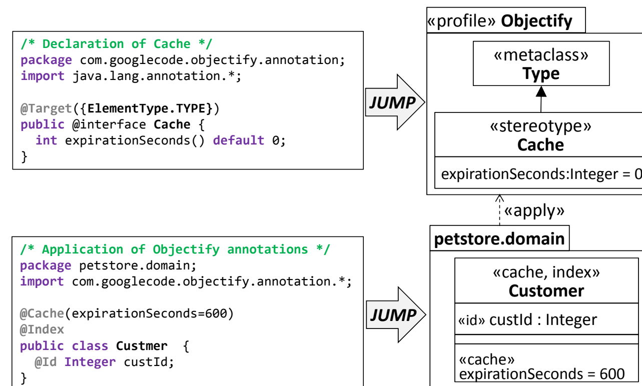


Fig. 1: JUMP in Action

Modeling Tool Name	Ver	Mapping (Java -> UML) Annotation Application	Mapping (Java -> UML) Annotation Declaration	UML Profile Generation
Altova UML http://altova.com/umodel.html	2013	Generic Java Profile	Interface	-
ArgoUML http://argouml.tigris.org	0.34	Generic Java Profile	Interface	-
Enterprise Architect http://sparxsystems.com	9.3	Built-in Tool Feature	Interface	-
Magic Draw http://nomagic.com	17.0.4	Generic Java Profile	Interface	-
Rational Software Architect http://ibm.com/developerworks/rational	8.5.1	Specific Profiles	Stereotype	-
Visual Paradigm http://visual-paradigm.com	10.2	Built-in Tool Feature	Class	-
JUMP Tool	1.0.0	Specific Profiles	Stereotype	+

Table 1: Comparison of Modeling Tools

agrams towards the Google App Engine (GAE) and supports developers to realize code generators that produce richer program code.

Mapping Java Annotations to UML. Currently, three significantly different solutions exist to support Java annotations for UML models: (i) *built-in* annotation feature of modeling tools, (ii) *generic* profile for Java, which enables capturing annotations and their type declarations, and (iii), profiles which are *specific* to a Java library or even an application with custom annotation type declarations. The first solution is certainly the most tool specific one as it goes beyond Java and UML. It facilitates to capture Java annotations, though the type declaration of an annotation and its applications are not connected. A generic profile for Java emulates the representational capabilities of Java's annotation language. Although with this approach the connection of annotation type declarations and their applications can be ensured, the native support of UML for annotating elements with stereotypes is still neglected. However, stereotypes specifically defined for annotation types would facilitate their application in a controlled UML standard-compliant way as they extend only the required UML metaclasses. From a language engineering perspective, such stereotypes facilitate defining constraints and model operations, such as model analysis or transformations, because they can directly be used in terms of explicit types similar to a metaclass in UML. Therefore, the *JUMP* tool is based on a mapping between Java's annotation language and UML's profile language [2], which enables the generation of specific stereotypes for corresponding annotation types that in turn leverage platform-specific profiles.

Existing Modeling Tools. Several commercial and open-source modeling tools support Java annotations at the modeling level as summarized in Table 1. While all evaluated modeling tools support the generation of annotated UML class diagrams from Java applications, none of them is capable of generating profiles for Java libraries, and so exploiting the powerful capabilities of stereotypes and profiles.

3 JUMP Tool

The *JUMP* tool envisages two main scenarios: *UML Profile Generation* and *Profiled UML Model Generation*. The first scenario is executed on a Java-based Eclipse project that covers the library from which a profile with the corresponding stereotypes is generated. Optionally, the generated profile is added to a local copy of the *UML-Profile-*

Store, which exposes frequently used profiles as plug-ins to facilitate their reuse, thereby avoiding to regenerate them again and again. The practical application of profiles is employed in the second scenario, which is integrated into the generation of UML class diagrams from Java applications. Annotation applications are replaced by the corresponding stereotypes applied to the reverse-engineered UML elements. The applied stereotypes are imported from the *UML-Profile-Store*. If user-defined annotation types are declared in the application, the respective profile is generated in a pre-processing step as they need to be defined prior their application. Such application-specific profiles are provided together with the generated UML class diagram rather than added by default to the *UML-Profile-Store*. Similarly to the first scenario, the second scenario is also executed on Java-based Eclipse projects.

Prototypical Implementation. To realize *JUMP*, we developed three transformation chains, i.e., *JavaCode2UMLProfile*, *JavaCode2ProfiledUML*, and *ProfiledUML2-JavaCode*. For injecting Java code to our chains, we reuse MoDisco [4], which generates a Java model that is considered as input for the *JUMP* tool. Hence, it can be considered as a specific model discoverer to extract annotation types from Java libraries in terms of profiles. To generate Java code from such models we extended the code generator provided by Obeo Network[5]. The prototype and the collected profiles from 20 Java libraries with over 700 stereotypes are available at our project web site[6].

Scalability Evaluation.To report on the scalability of the *JUMP* tool, we measured the execution time of applying the *JavaCode2UMLProfile* and *JavaCode2ProfiledUML* transformation chain to real-world libraries and applications. For obtaining the measures, we executed them in Eclipse Kepler SR2 with Java 1.7 on commodity hardware: Intel Core i5-2520M CPU, 2.50 GHz, 8,00 GB RAM, Windows 7 Professional 64 Bit.

Table 2 summarizes our obtained results by emphasizing (i) the number of *code elements* in the intermediate Java model, (ii) the number of *declared* and *applied stereotypes* and (iii) the measured *execution times*. The rationale behind our selection of libraries (JPA, Objectify, Spring[7], and EclipseLink[8]) and applications (Petstore[9], DEWS-Core[10], Findbugs[11], and once more EclipseLink) is to consider small-sized to large-sized libraries and applications with varying number of declared and applied stereotypes. Clearly, the size of the input models passed to the transformation

Library	Code Elements	Declared Stereotypes	Execution Time in Sec
JPA[1]	20K	84	2.362
Objectify[2]	40K	18	1.842
Spring[6]	500K	63	10.292
EclipseLink[7]	700K	127	29.614
Application		**Applied Stereotypes**	
Petstore[8]	10K	287 (12 Profiles)	4.581
DEWS-Core[9]	30K	253 (2 Profiles)	3.116
Findbugs[10]	100K	1808 (3 Profiles)	26.620
EclipseLink	700K	7117 (3 Profiles)	199.028

Table 2: Performance Measures

[5] http://marketplace.eclipse.org/content/uml-java-generator
[6] http://code.google.com/a/eclipselabs.org/p/uml-profile-store
[7] http://projects.spring.io/spring-framework
[8] https://www.eclipse.org/eclipselink
[9] http://oracle.com/technetwork/java/index-136650.html
[10] Distant Early Warning System (DEWS), a use-case of the ARTIST project [1]
[11] http://findbugs.sourceforge.net

chains has a strong impact on the execution time of the *JUMP* tool as they are traversed throughout the generation of profiles and profiled models. Regarding the profile generation, the number of generated stereotypes is another main factor that impacts on the execution time. The more stereotypes are generated the more extensions to UML metaclasses need to be created. For instance, even though the JPA is compared to Objectify smaller in size the execution time is higher because a lot more transformation rules are applied when considering the number of declared stereotypes. Similarly, the number of applied stereotypes and their respective profiles impacts on the execution time. For instance, in the Petstore application, stereotypes are applied from 12 different profiles, which explains the higher execution time compared to DEWS-Core, even though the latter is larger in size. Finally, the execution time of generating profiled models is generally higher compared to profiles because the class structure of the former is much larger in size compared to the latter. For instance, considering EclipseLink and the number of generated stereotypes compared to classes the factor is almost 30.

4 Future Work

The *JUMP* tool aims at closing the gap between programming and modeling concerning annotation mechanisms. Still, open challenges remain to further integrate the two areas. For instance, with the latest Java version (1.8) we have repeating annotations that enable the same annotation to be repeated multiple times in one place which is currently not supported by stereotypes in UML. Furthermore, we plan to incorporate the production of UML activity diagrams for Java method implementations in JUMP in order to represent also annotation applications on the statement level in UML. Another line of research we plan to investigate is to further evaluate the JUMP tool in the context of the ARTIST project by empirical studies with our use case providers to determine the practical benefits for understanding and migrating legacy applications.

References

1. A. Bergmayr et al. Migrating Legacy Software to the Cloud with ARTIST. In *Proc. of CSMR*, pages 465–468, 2013.
2. A. Bergmayr, M. Grossniklaus, M. Wimmer, and G. Kappel. JUMP—From Java Annotations to UML Profiles. In *Proc. of MODELS*, 2014.
3. M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan&Claypool, 2012.
4. H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In *Proc. of ASE*, pages 173–174, 2010.
5. L. Fuentes-Fernández and A. Vallecillo. An Introduction to UML Profiles. *European Journal for the Informatics Professional*, 5(2):5–13, 2004.
6. P. Langer, K. Wieland, M. Wimmer, and J. Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *JOT*, 11(1):1–29, 2012.
7. U. Nickel, J. Niere, and A. Zündorf. The FUJABA Environment. In *Proc. of ICSE*, pages 742–745, 2000.
8. R. F. Paige and L. M. Rose. Lies, Damned Lies and UML2Java. *JOT*, 12(1), 2013.
9. B. Selic. The Less Well Known UML: A Short User Guide. In *Proc. of SFM*, pages 1–20, 2012.

DPLFW: a Framework for the Product-Line-Based Generation of Variable Content Documents

Abel Gómez[1], Pau Martí[2], M. Carmen Penadés[2], and José H. Canós[2]

[1] AtlanMod team (Inria, Mines Nantes, LINA)
4 rue Alfred Kastler. 44307 Nantes, France
abel.gomez-llana@inria.fr
[2] ISSI – DSIC, Universitat Politècnica de València.
Cno. de Vera, s/n. 46022 Valencia. Spain.
{pmarti,mpenades,jhcanos}@dsic.upv.es

Abstract. *Document Product Lines* (DPL) is a document engineering methodology that applies product-line engineering principles to the generation of documents in high variability contexts and with high reuse of components. Instead of standalone documents, DPL promotes the definition of families of documents where the members share some common content while differ in other parts. The key for the definition is the availability of a collection of content assets which can be parameterized and instantiated at document generation time.

In this demonstration, we show the features of the DPL framework (DPLFW), the tool that supports DPL. DPLFW implements the domain engineering and application engineering stages of typical product line engineering approaches, supports different asset repositories, and generates customized documents in different output formats. We use the case study of the generation of customized emergency plans in a University campus [http://youtu.be/ueKGfmfkyI0].

1 Motivation

The concept of document has changed in last decades from the classical printed artifact to a purposeful and self-contained collection of information in a technology-neutral way [1]. In more and more domains, documents are the central pieces of the business processes; moreover, in most cases the generation of customized documents has become the final milestone. Examples are customized emergency plans for organizations, customized learning objects based on students' profiles, or customized book catalogs made according to customers' preferences.

Customization can be made in three dimensions, namely content, structure and presentation. Specifically, content customization has been tackled from two different perspectives. On the one hand, proposals on Variable Data Printing (VDP) use variables within documents that are used as placeholders for content that are replaced with values to generate customized documents. These approaches are mostly XML-based [5, 8, 10]: document components are defined

in XML, and the customized document is generated using XSLT, XPath, and other related technologies. In these proposals, the variability model is implicit (that is, it is embedded in XML, XPath and XSLT expressions), forcing document engineers to have a high knowledge about the XML world. On the other hand, more recent proposals such as [4, 9] use a product line approach to model the variability explicitly. These proposals use feature models to identify the variability points from a domain-oriented perspective, hiding the XML complexity.

The *Document Product Lines* approach (DPL) [2, 7] is an example of the latter. DPL was created with a twofold goal: first, to make variable content documents creation affordable to non-expert users by including a domain engineering process previous to document generation; and second, to enforce content reuse at domain level following principles of Software Product Line Engineering (SPLE). We implemented a tool, DPLfw [3], to provide the methodological and technological background to creating variable content documents by the DPL approach. DPLfw implements a true product line engineering process where the content variability is represented using document feature models and different variants of the document may be generated by defining different document configurations. DPLfw was developed following the MDE and Model Driven Architecture (MDA) paradigms, which allowed us to take advantage of code generation techniques for the implementation of the tool prototype.

2 The *Document Product Lines* methodology

DPL is a method for the generation of variable content documents. As in SPLE, the DPL process is the concatenation of two main subprocesses, namely Domain Engineering and Application Engineering. Fig. 1 shows the most relevant tasks and artifacts in each subprocess using the BPMN notation.

The goal of the *Domain Engineering* is to define a family of documents and related artifacts. A family is a set of documents that share some mandatory content while differ in other optional content. To enforce reuse, the content of a

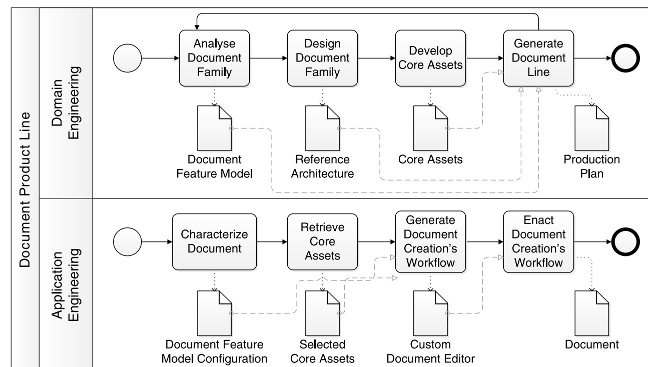


Fig. 1. DPL-based Document Generation Process

family of documents is structured via a document feature model. In the *Analyse Document Family* task, a domain engineer specifies the documents in terms of features and feature attributes. The result is a document feature model including mandatory, optional, alternative features and their corresponding attributes (if applicable). In the *Design Document Family* task, the generic document architecture is defined by identifying the document components (called *core assets* in the SPLE terminology and *InfoElements* in DPL) required according to the feature model built in the previous stage. These *InfoElements* may define a set of variables corresponding to a set of placeholders, i.e., parts of their content that may be instantiated at later stages of the document generation process. Specific instances of the architecture are created later in the process, after the variability points and document variable have been fixed for a specific document. DPL assumes the existence of a *Repository* where *InfoElements* are stored and organized for reuse. Metadata are attached to each *InfoElement* in order to support retrieval processes in the *Develop Core Asset* task to find existing components. Finally, in the *Generate Document Line* task, a production plan is obtained; it is a process that specifies how the components are integrated according to the different relationships defined between the document features.

In the *Application Engineering* subprocess, a member of the document family is generated. The process starts with the *Characterize Document* task by selecting in the configuration the specific document features and variable values to be included in the final document. Next, the core-assets associated to the selected document features are retrieved and put together to *Generate a Document Creation's Workflow* model that it is used to generate a set of *Custom Document Editors*. The editors provide guidance for *Enacting the Document's Creation Workflow* by giving both a task-oriented and user-centered view of the document based on editing task and permissions producing the final document.

3 The DPLfw Framework

Figure 2 describes how a (simplified) DPL process is carried out in DPLfw. The *Domain Engineering* stage is an iterative process. For the sake of simplicity no specific order is enforced to execute its tasks as far as there is a fully populated

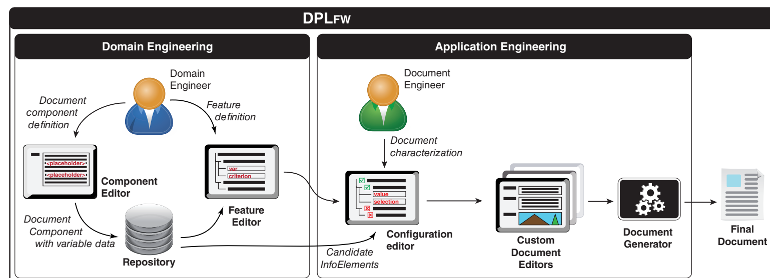


Fig. 2. DPLfw overview

document feature model describing the domain at the end of the stage. The *Feature Editor* is used by the *Domain Engineer* to characterize the variability of the domain as a document feature model. The *Feature Editor* interfaces with the *Repository*, which contains the *InfoElements* that will later be reused. All these elements support the *Analyse Document Family* task (cf. Figure 1).

It is noteworthy that, for the sake of simplicity, the *Reference Architecture* matches the structure of the feature model, and thus, the *Design Document Family* task does not require user-interaction. The *Component Editor* supports the *Develop Core Assets* task and is used to create new *InfoElements* and add them to the *Repository*. The *Generate Document Line*, which will describe how to retrieve and integrate the different components to obtain the final product, is also hidden from the user-point of view.

Regarding the *Application Engineering* subprocess, the *Configuration Editor* supports the *Characterize Document* task through the selection of variability points. Once a *document feature model configuration* is defined, the *Enact Document Creation's Workflow* task starts and the *Custom Document Editors* are generated by composing the *InfoElements*. These editors are used to fill in any remaining variable data. Finally, the *Document Generator* (a DITA-based [6] document generator engine) integrates the different components to obtain a fully instantiated document generated in a specific format (printed, hypermedia, etc.).

4 Demonstration: The *UPV Campus* Emergency Plans

Emergency plans development is the field we selected for our case study. An emergency plan is a document that contains all the knowledge required to respond to any incident in an organization. Emergency plans development is a domain that brings together a set of requirements that makes it an interesting real case study. In our demonstration we will show how a family of emergency plans is modeled, and how a customized emergency plan is generated. The *UPV Campus* is a real case study, and its document feature model represents the family of emergency plans of the university campus; where each building of the campus has its own emergency plan obtained as a configuration of the family.

Prior to the adoption of the DPL methodology in the UPV, new emergency plans were manually created using a text editor (MS Word). Applying DPL to the development of emergency plans requires that a set of document fragments (i.e. *InfoElements*) need to be created to be combined in the final document and a Document Feature Model describing the family of plans. However, sometimes an *InfoElement* needs some small parts to be changed from one emergency plan to another (e.g. the building name or the specific maps or warning systems, etc.). This scenario allows us to show how both approaches for document customization (variability-based customization and Variable Data Printing) have been combined in DPL and DPLFW.

5 Conclusions

DPLfw supports the generation of variable content documents in a product-line style. Such an approach has several advantages. On one hand, a domain-oriented variability specification helps to hide the complexity intrinsic to document description languages such as XML. On the other hand, the definition of generic content components increases the reusability of content significantly. DPLfw supports the generation of customized documents with high levels of reuse. Its foundation, DPL, aims at raising the level of abstraction in comparison with previous approaches, helping domain engineers and document engineers to develop families of documents without knowledge of the underlying document representation techniques such as XML, XPath, DITA or DocBook.

We have illustrated the use of DPLfw with an example taken from our collaboration with the emergency planning team at the *Universidad Politècnica de València*. The DPLfw documentation and prototype are publicly available for download in [3].

References

1. Glushko, R., McGrath, T.: Document Engineering: Analyzing and Designing Documents for Business Informatics & Web Services. MIT Press (2005)
2. Gómez, A., Penadés, M.C., Canós, J.H., Borges, M.R., Llavador, M.: A framework for variable content document generation with multiple actors. Information and Software Technology 56(9), 1101 – 1121 (2014), special Sections from "Asia-Pacific Software Engineering Conference (APSEC), 2012" and "Software Product Line conference (SPLC), 2012"
3. ISSI Research Group: DPLfw (2014), `http://dpl.dsic.upv.es/`, (spanish only)
4. Karol, S., Heinzerling, M., Heidenreich, F., Assmann, U.: Using feature models for creating families of documents. In: Proceedings of the 10th ACM symposium on Document engineering. pp. 259–262. ACM, New York, USA (2010)
5. Lumley, J., Gimson, R., Rees, O.: A framework for structure, layout & function in documents. In: Proceedings of the 2005 ACM symposium on Document engineering. pp. 32–41. ACM, New York, USA (2005)
6. OASIS: Darwin Information Typing Architecture (DITA) Version 1.2 (Dec 2010), `http://docs.oasis-open.org/dita/v1.2/spec/DITA1.2-spec.html`
7. Penadés, M.C., Canós, J.H., Borges, M.R., Llavador, M.: Document product lines: variability-driven document generation. In: Proceedings of the 10th ACM symposium on Document engineering. pp. 203–206. ACM, New York, USA (2010)
8. Piccoli, R.F.B., Chamun, R., Cogo, N.C., de Oliveira, J.a.B.S., Manssour, I.H.: A novel physics-based interaction model for free document layout. In: Proceedings of the 11th ACM symposium on Document engineering. pp. 153–162. ACM, New York, USA (2011)
9. Rabiser, R., Heider, W., Elsner, C., Lehofer, M., Grünbacher, P., Schwanninger, C.: A flexible approach for generating product-specific documents in product lines. In: Bosch, J., Lee, J. (eds.) SPLC. Lecture Notes in Computer Science, vol. 6287, pp. 47–61. Springer (2010)
10. Sellman, R.: VDP templates with theme-driven layer variants. In: Proceedings of the 2007 ACM symposium on Document engineering. pp. 53–55. ACM, New York, USA (2007)

Combining Textual and Web-based Modeling

Martin Haeusler, Matthias Farwick, and Thomas Trojer

University of Innsbruck, Technikerstr. 21a, 6020 Innsbruck, Austria
firstname.lastname@uibk.ac.at,

Abstract. Documenting large scale IT-architectures is a laborious task that is executed by many different stakeholder types. We argue that a major obstacle that keeps stakeholders from keeping models up-to-date is inadequate user interfaces for specific stakeholders. In this demo paper we present a novel modeling tool that provides adequate stakeholder views and describe the implementational challenges. The tool motivates users to contribute documentation by allowing textual modeling for technical users and web-based modeling via forms for business users at the same time. The tool demonstration video is available at: https://www.youtube.com/watch?v=PaP2Sppiv7g&feature=youtu.be

1 Introduction

In the context of Enterprise Architecture Management (EAM) and systems operation management, specialized tools are often used to model the dependencies between the IT-infrastructure, deployed applications and the business functions they support [5]. These models are used to analyze the current architecture, assess risks and plan changes to the architecture.

In our previous empirical research [3] we showed that a major problem in EA documentation is that the EA models become quickly outdated. Therefore we focused on increasing automation in EA documentation and enhancing semi-automated data collection processes [3]. However, we realized that automation is not always applicable, especially in the case of application modeling where human abstraction is needed to hide distracting detail.

Therefore, we argue that in order to motivate stakeholders to continuously contribute their knowledge to the model, the data input methods need to be adapted to the preferences of the different user types. In our previous work [4] we presented *Txture*, an EA documentation tool that uses a textual domain-specific language (DSL) as a collaborative model input method. In *Txture* visualizations of the model can then be created online via a corresponding web-application. Experience that we gained with the tool in practice showed that this textual input method is fast and intuitive for technically skilled users and increases their motivation to contribute to the documentation, in particular when they are already working in a textual environment. However, for less technically skilled users, no alternative input method was available.

In this demo paper we therefore present a proof–of–concept prototype as part of the *Txture* development process that combines textual and form-based

modeling in the web. It thereby allows distinct user groups to work on the same model in their preferred input style. Both the textual and the web-based clients are kept in-sync.

There exists some related work in this area. For example the projectional modeling approach of the Meta Programming System[1] or the recent work of Atkinson et al. [1] that combines graphical and textual modeling. The two major differences to our approach are that the presented systems (a) work on desktop clients and (b) are applying a projectional editing approach in which the text can only be edited with a specific tool and not any text editor. Other approaches, like the work of Engelen et al. [2] only allow one-way synchronization between the clients. For example, the authors propose a textual language for describing *UML Activity* instances which are embedded in an *XMI* file. This file is processed by a compiler which transforms the textual descriptions into real Activities. The opposite direction – from object representation to textual syntax – is left as an open issue. In our demonstration video that accompanies this paper, we explain our use–case in more detail and demonstrate both transformation directions. Since the combination of the two modeling paradigms is a particularly complex task, we detail some of the key technical challenges in this demo paper that we faced when implementing this novel approach to model management and EA documentation.

2 Combining Textual and Web-based Modeling

The prototype consists of three main components. First, an Eclipse-based plugin that allows textual modeling according to a pre-defined syntax and is based on the *Xtext* framework[2]. Second, a web-application that allows to enter model data in a form-based manner. Third, a model repository that centrally stores the model for all clients. The difficulty of integrating the two types of modeling lies in the two different model organization styles. Model element organization on the textual side is governed by folders and files that contain structured text which is parsed to build the model in memory. On the other side a multi-user web application is employed that stores the model centrally in a model repository. As shown in Figure 1 we tackle this problem with our tools and both, the changes to the text model and the ones applied to the web-client synchronized via the model repository.

2.1 Challenges in Synchronizing Textual and Web-based Models

This section gives an overview of the most significant problems and our solutions. Figure 1 shows the simplified communication between attached clients in a distributed modeling context. Every number in the figure corresponds to a paragraph number which explains the synchronization challenges that occur.

[1] JetBrains MPS Homepage: `http://jetbrains.com/mps`
[2] Xtext Homepage: `http://www.xtext.org`

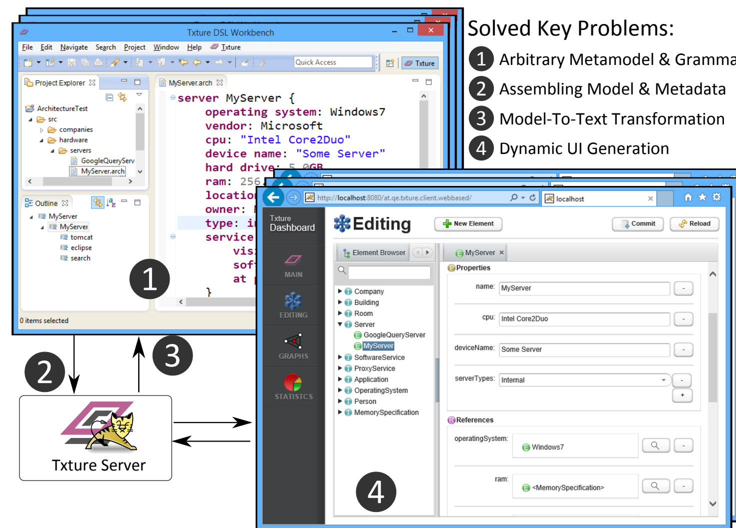


Fig. 1. Synchronizing Model Data across multiple Clients

Problem 1: Arbitrarity of Grammars & Metamodels
The textual client uses *Xtext*, a parser-based framework for DSLs. In order to suit the requirements of a given use case, our tool needs to be capable to deal with any Xtext grammar and metamodel.
Solution: We employ a reflective approach to metamodel access by utilizing the capabilities of *EMF*[3], avoiding a direct compile-time dependency to any given metamodel. In order to deal with arbitrary grammars, we offer interfaces for *Language Extensions*; i.e. *Eclipse Plugins* that contain Xtext grammars and Model–to–Text generators.

Problem 2: Assembling the Model & Gathering Metadata
Since the textual client relies on files containing concrete DSL syntax for storing the model, they must be parsed and merged into a single model. Still, we later require the source file location of any merged model element. This is important to later re-assemble model data into the correct positions in the text files and folders.
Solution: Xtext itself is based on EMF *Ecore* for representing the *Abstract Syntax Trees* (AST) that result from each parser run. We assemble the AST of each DSL file into a common model, resolving all encountered cross-references between model elements. The result is the model that we need to commit to the server. Xtext also offers access to element-based metadata (e.g. the element file location) which we need to send to the server as well. Please refer to Section 3 for details.

[3] Eclipse Modeling Framework: `http://www.eclipse.org/modeling/emf/`

Problem 3: Model–to–Text Transformation & File Locations
During a checkout process, the textual client receives a serialized version of the model. Each element in the model must be converted to concrete DSL syntax and furthermore must be placed in the correct file at the correct position.
Solution: The Model–to–Text Transformation depends directly on the grammar of the model at hand and therefore cannot be processed directly by the generic part of the application. Consequently, we defer this task to a specialized Eclipse Plugin which contains the Model–to–Text generator for the particular grammar. After generating the concrete syntax for a model element, the tool decides upon the file location of each resulting piece of text by taking the element file location meta-information into account. If none is given (e.g. if the element was newly created by the web client), it is put into a special folder for *pending elements*. The user can then manually reorganize these elements into appropriate locations.

Problem 4: Dynamic UI Generation
The *Vaadin*[4]-based web clients need to process instances that adhere to a metamodel which is unknown at compile time. Similar to the textual client, the web client needs to treat the metamodel as arbitrary. This raises the question of how to assemble a proper user interface in this scenario.
Solution: We employ a dynamic GUI generation approach which infers a UI widget for every element and property in the metamodel. For example, the inference mechanism will produce a Textfield widget for every attribute of type *String* with multiplicity one. For an attribute with an enumerated type, a Combo Box will be generated that contains the literals of the enumeration as choices. In a second step, the inferred widget receives its value directly from a given model element, effectively creating a databinding between GUI and model. When the model changes are committed, this binding is used in the inverse direction to apply changes made by the user directly to the model.

3 Problem Discussion: Element-based Metadata

As explained by Atkinson et al. [1], maintaining element-based metadata in an environment based on direct editing and parser technology (such as our text-based client) is a difficult problem. Metadata, such as unique identifiers and file locations, is usually not contained in the textual syntax for usability reasons. Therefore, it must be processed and maintained by the system in the background. Every time a file is changed by the user, a new AST is built by the parser. In our scenario, that AST (after minor modifications) is effectively the same as the resulting model. However, we only have element-based metadata for our current AST, not for the new one produced by the parser. Furthermore, there are no unique identifiers in the new AST that we could utilize to match two elements. For that reason, we have to rely on content-based matching. We use the *EMFCompare*[5] framework for this purpose, which identifies pairs of elements

[4] Vaadin Homepage: `https://vaadin.com/home`
[5] EMFCompare Homepage: `http://www.eclipse.org/emf/compare/`

(one from each parse tree) that refer to the same semantic object. Due to the undecidability of the model matching problem, resulting matches are only best-effort attempts. We have to make this trade-off of potentially losing element metadata in order to preserve the user experience of true textual editing, as opposed to indirect ("projectional") editing employed for example by Atkinson et al. It is important to note that the maintenance of element-based metadata across parse processes is not a strict requirement for the current tool implementation (the textual client is currently the sole producer and consumer of the metadata), but will be more important once other editors are added to the tool as well, which in turn may add more metadata to each element.

4 Conclusion & Outlook

In this paper we presented a modeling tool prototype that allows for the synchronization between textual and web-based modeling via a central model-repository. The motivation for this tool is the combination of these modeling paradigms to allow different stakeholder types to enter data in their desired format in the context of IT-architecture modeling. We highlighted the technical and usability challenges we faced in the implementation. We have shown that the different modeling paradigms like form-based and textual modeling pose a challenge in the implementation as well, in particular when it comes to element–based metadata in text–based environments that allow for direct editing.

The insights we gained from the here-described prototypical implementation are currently used to extend our EA tool *Txture*[6] with form-based modeling capabilities. We are also extending our approach of using multiple types of modeling editors. An editor e.g. using *Excel spreadsheets* is part of our current efforts.

References

1. Atkinson, C., Gerbig, R.: Harmonizing Textual and Graphical Visualizations of Domain Specific Models Categories and Subject Descriptors. In: Proceedings of the Second Workshop on Graphical Modeling Language Development. pp. 32–41. ACM, New York, NY, USA (2013)
2. Engelen, L., van den Brand, M.: Integrating textual and graphical modelling languages. Electronic Notes in Theoretical Computer Science 253(7), 105–120 (2010)
3. Farwick, M., Schweda, C.M., Breu, R., Hanschke, I.: A situational method for semi-automated Enterprise Architecture Documentation. Software & Systems Modeling (2014)
4. Farwick, M., Trojer, T., Breu, M., Ginther, S., Kleinlercher, J., Doblander, A.: A Case Study on Textual Enterprise Architecture Modeling. In: Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2013 17th IEEE International. pp. 305 – 309. IEEE, Vancouver, BC (2013)
5. Matthes, F., Buckl, S., Leitel, J., Schweda, C.M.: Enterprise Architecture Management Tool Survey 2008. Tech. rep., Technische Universität München,Chair for Informatics 19 (sebis) (2008)

[6] Txture Homepage: `http://www.txture.org`

Tool support for Collaborative Software Quality Management

Philipp Kalb and Ruth Breu

Institute of Computer Science
University of Innsbruck
Email: philipp.kalb, ruth.breu@uibk.ac.at

Abstract. Nowadays cloud services and complex cyber–physical systems gradually find their way into practice. As a result the need for end–to–end software quality management across platform and organizational boundaries has become paramount. One solution proposed by the software engineering community is the use of integrated model repositories for interchanging, interlinking and analyzing software engineering data and coordinating actions of manifold stakeholders working on this data. With MoVE, the Model Evolution Engine, we have developed a model repository supporting model–based data management in heterogeneous tool environments. The state machine based workflow concept allows a tight integration of data and automated and manual actions on the repository in a change–driven way. In this paper we will present the essential components of our MoVE Framework, starting with an introduction of the most important concepts, followed by the state based workflow language which will be contained in our demonstration [1].

1 Introduction

Modern software systems tend to consist of fragmented services across devices, platforms and organizational boundaries. To handle the rising complexity of such systems the consideration of end–to–end quality management is of major importance. For example the management of security in large–scale system like national health records requires coordinated efforts of heterogeneous stakeholders. Ranging from security engineers tackling technical issues such as designing secure software services to non–technical stakeholders such as compliance managers, surveying legal regulations, are also involved. As a consequence these systems demand for a consolidated treatment of data and processes in the realm of IT management, software engineering and systems operation [1].

Standards such as ITIL [2] and the software engineering community suggest the use of integrated model repositories for interchanging, interlinking and analyzing data [3,4,5]. While repositories have a long history in software engineering there exists still a huge gap in integrating different kinds of model–based data and semi–structured data. Additionally, the support of processes for end–to–end quality management, especially the interoperation of strictly structured

[1] `http://youtu.be/WKG__UnHL8U`

processes in IT management and agile processes in software engineering comes with further challenges. Flexible ticket based workflow management tools, such as IBMs Jazz platform [6] or Atlassians JIRA project management tool [7], have reached first adoption in practice in recent years. However, they do not address the aspect of data integration and are still weak in allying manual and automated tasks.

With MoVE, the Model Versioning and Evolution Engine, we have conceptualized and implemented a model repository referring not only to the data integration aspect but also the collaboration aspect. The MoVE–approach has a focus on continuous model integration for software engineering. MoVE provides methods to achieve traceability across tools, by applying concepts of meta–modelling and interlinkage. A key feature of MoVE is support for change–driven engineering, which is a novel methodology to cope with system evolution by supporting workflows triggered by changes of the systems data artifacts. The workflow language enables quality management to support change management as described in standards and guidelines such as ITIL [2] or ISO/IEC 20000 [8]. Hence, system evolution respecting data artifacts can be controlled to guaranty an integrated quality process during the complete software systems life cycle.

In Section 2 we will summarize the important concepts of the MoVE Framework. Section 3 describes the novel MoVE Workflow language, which is used to established a change–driven process.

2 Concepts and Architecture of the MoVE Framework

Figure 1 shows the overall architecture of the MoVE framework, consisting of the central MoVE Repository and multiple MoVE Clients connecting software engineering and IT management tools to the repository through MoVE Adapters.

From the conceptual point of view the basis of the MoVE Framework is the **Common Meta Model** (CMM). The CMM configures the data structures used in the MoVE Repository by specifying the (meta) model elements and their relationships. The CMM consists of a set of (partial) meta models such as the System Model, the Security Requirements Model, the Testing Model and the like. A full integration of all data structures of connected tools is not intended, the language should only contain the structures necessary for stakeholder collaboration. The CMM is designed using an UML modelling tool [2]. To enhance the UML models with MoVE specific features a UML profile (the MoVE Executeable Profile) defines a number of stereotypes. After its design the CMM is uploaded to the MoVE Repository with the help of a *Configuration–Service*. The Configuration Service uses the XMI representation of the CMM to configure the repository.

At the instance level Create, Read, Update, Delete and Query (**CRUDQ**) services the MoVE Repository provides to commit instances of the CMM. CRUDQ –Services are consumed by MoVE Adapters, which are plug–ins into client–side

[2] in the current implementation Magic Draw is used

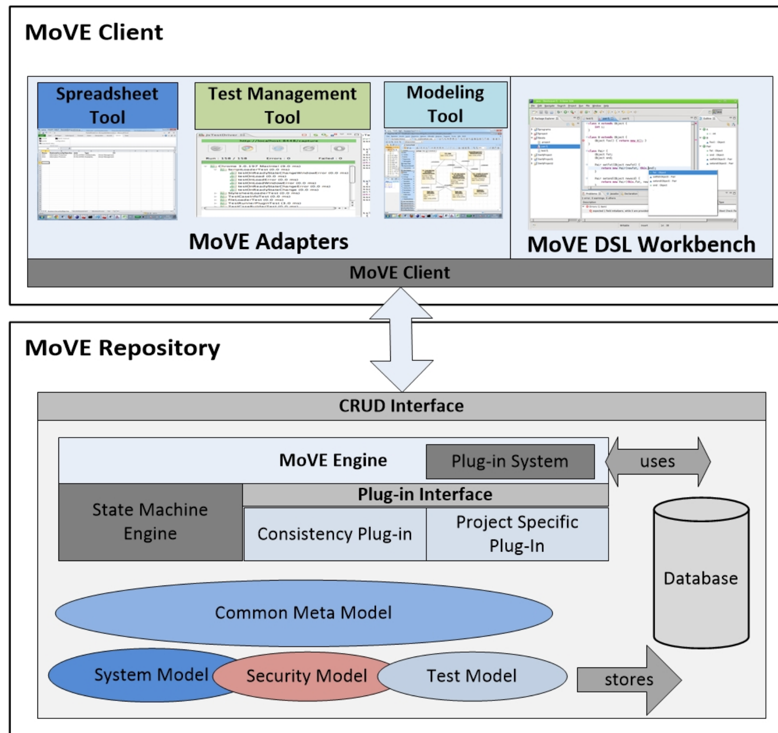


Fig. 1: Conceptual Architecture of the MoVE Framework

tools. These client–side tools are not limited to UML modelling tools. In the current environment we have developed e.g. a MoVE Adapter for Microsoft Excel to demonstrate the applicability of our concepts in a heterogeneous tool environment. The MoVE Adapter's main responsibility is to manage the mapping between the tool data representation and the representation in the MoVE Framework. CRUDQ–Services interact with the MoVE Engine, which is the main component of the MoVE server–side.

The major tasks of the MoVE Engine are to support versioning and persistency for all model elements stored in the MoVE Repository and to provide a **Plug–in Interface**. MoVE is event–driven in terms of generating an event for each change of a model or model element (using the CRUDQ–Services for changes). Each occurring change is analysed and then transformed into a change event. Server–side MoVE plug–ins listen to certain types of events and can trigger further actions. A **Plug–in System** allows users to register plug–ins for every (partial) model separately and therefore to decide which event should result in further actions. A crucial consumer of change events is the MoVE State Machine Engine, which allows to create state machine based workflows triggered by change events. Due to the importance of the MoVE workflow methodology it be described in Section 3 in more details.

3 The MoVE State Machine Workflow Language

The general idea of our state machine based workflow approach is that a model element can evolve during the operation of a system and typically undergoes a dedicated life cycle which is represented as a UML state machine.

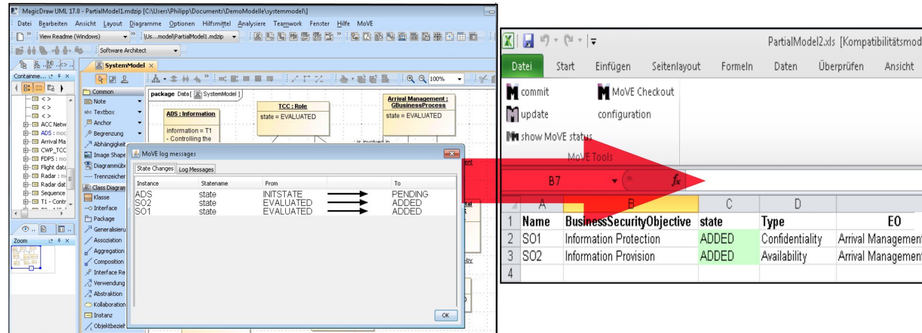


Fig. 2: System Model In Magic Draw with Updates for the Security Model in MS Excel

Each model element in the CMM can be attached with **states** and **state machines**. The states determine the quality gates in the quality lifecycle of the model element, like a *Security Requirement* being in the states *ADDED*, *COMPLETE* or *EVALUATED*. Transitions between states may be triggered in an *automatic* way by internal events stemming from other state machines, a timer or change events created by the MoVE Engine. Alternatively, *manual* transitions need user interaction which is implemented via systems such as mail [3]. Each transition can be guarded by conditions defined in OCL or the Hibernate Query Language (HQL).

Figure 2 shows two screenshots from our demonstration. The underlying CMM links a System Model with security requirements. The configured workflows control that on change of elements of the System Model, the linked security requirements have to be be re–evaluated. On the left side one can see a System Model designed with Magic Draw. On the right side, a spreedsheet–view in MS Excel contains the linked security requirements. Figure 2 shows the situation after an update of the System Model. The state of the linked security requirement changes from *EVALUATED* to *ADDED* and thus causes a re–evaluation of the security requirements.

In case a state has changed, it is possible to define a number of actions *onEntry* of the new state and *onExit* of the current state. These actions e.g. may involve external systems such as mail to notify stakeholders. Actions can be defined with two options: (i) The MoVE Executeable Profile contains several

[3] in our implementation we use Mylyn

predefined actions that can be composed in standard UML activity diagrams. Using this option it is possible to trigger predefined actions in a certain order but with limited expressiveness. (ii) Alternatively it is possible to use the fUML standard to create rich activity diagrams. FUML supports not only the actions defined in the MoVE Executeable Profile but a huge subset of UML activity diagrams. This enables users to create complex model changes on state changes.

4 Conclusion

In this demonstration paper we have sketched the MoVE framework which is a powerful model repository that integrates model storage capabilities.

The MoVE framework has been developed within the EU-IST project SecureChange [9] and has been employed as a central model repository for security policy interlinkage within the EU-IST project PoSecCo [10]. The has been evaluated in several case studies, both from applicability in industrial context, but also performance perspective. Within PoSecCo, the MoVE environment included six connected tools, using more than 4000 instances and about 15 state machines. The tool is available open source under Eclipse EPL license.

References

1. Breu, R., Agreiter, B., Farwick, M., Felderer, M., Hafner, M., Innerhofer-Oberperfler, F.: Living models - ten principles for change-driven software engineering. Int. J. Software and Informatics **5**(1-2) (2011) 267–290
2. APM Group Ltd: ITIL official website, accessed on february 19, 2014. http://www.itil-officialsite.com/home/home.aspx.
3. Sztipanovits, J.: Cyber physical systems - convergence of physical and information sciences. it - Information Technology **54**(6) (2012) 257–265
4. Atkinson, C., Stoll, D., Bostan, P.: Supporting view-based development through orthographic software modeling. In: ENASE. (2009) 71–86
5. Bruegge, B., Creighton, O., Helming, J., Kogel, M.: Unicase an ecosystem for unified software engineering research tools. In: Third IEEE International Conference on Global Software Engineering, ICGSE. (2008)
6. IBM: Jazz – rational team concert; project web side, accessed on february 20, 2014. https://jazz.net/products/rational-team-concert/.
7. Atlassian: Jira – project web side, accessed on march 20, 2014. https://www.atlassian.com/software/jira.
8. ISO/IEC: ISO/IEC20000. Information technology – Service management. ISO/IEC (2011)
9. SecureChange: EU project, accessed on june 30, 2014. http://www.securechange.eu/.
10. PoSecCo : EU project, accessed on june 30, 2014 http://www.posecco.eu/.

DSLFORGE: Textual Modeling on the Web

Amine Lajmi[1], Jabier Martinez[2,3], and Tewfik Ziadi[3]

[1] Software Architect, Paris, France, amine.lajmi@dslforge.com
[2] SnT, University of Luxembourg, Luxembourg, jabier.martinez@uni.lu
[3] LIP6, Université Pierre et Marie Curie, Paris, France, tewfik.ziadi@lip6.fr

Abstract. The use of Model-Driven Engineering in software development is increasingly growing in industrial applications as the technologies are becoming more mature. In particular, domain-specific languages bring to end-users simplicity of use and productivity by means of various artifacts generators. However, end-users still need to cope with heavy modeling infrastructures and complex deployment procedures, before being able to work on models. In this paper, we propose a centralized lightweight approach for performing textual modeling through web browsers. DSLFORGE is a generator of online text editors. Given a language grammar, the tool allows to generate lightweight web editors, supporting syntax highlighting, syntax validation, scoping, and code completion. DSLFORGE allows also automatic integration of existing code generators into the generated web editor providing a complete online modeling user experience.

Demo: `http://youtu.be/KN6cneWhhKY`

Keywords: textual modeling, online editor, model-driven engineering, domain-specific languages

1 Introduction

Domain-Specific Languages (DSL) [10] let end-users feel the advantages of using domain abstractions instead of general-purpose language constructs. Therefore, Model-Driven Engineering (MDE) is gaining more success in industrial applications as the available methods and tools are becoming more mature. Modeling technologies have made big steps towards better integration and simplicity of use and, as a consequence, domain-specific standards have met great success within multiple communities (e.g. Modelica [15], AUTOSAR [2], and SysML [17]).

However, end-users still need to cope with heavy development infrastructures and complex deployment procedures when it comes to using modeling tools. This is because most of the existing tools are based on general purpose Integrated Development Environments (IDEs) such as Eclipse or MPS, or standalone proprietary applications as MetaEdit+ [11]. Indeed, in most of the tools, whether bundled into standalone Rich Client Platforms (RCP), or packaged separately as individual features, both *Tooling* and *Runtime* need to run on top of heavy infrastructures. The deployment of modeling tools is still a tedious task for non-developers and goes sometimes against the adoption of DSLs in enterprises.

Indeed, there is a lack of practical solutions, lightweight and easy-to-deploy. Moreover, modeling resources, as any kind of software resources today, experience an increasing demand to be accessed using different devices such as tablets and smartphones and from any place in the world at any time.

In this paper, we present the DSLFORGE tool, as support of a lightweight centralized approach that allows end-users to edit and process textual models through web browsers. The paper is organized as follows: Section 2 describes current web-based approaches and Section 3 presents the DSLFORGE tool, its methodology of use, two functional examples and its internal architecture.

2 Modeling in the web

There are still technological challenges to achieve modeling on the web. Nevertheless, some initiatives targeting *graphical* languages are worth to notice. GenMyModel [4] allows end-users to create UML diagrams and launch artifacts generation. AToMPM [16] is an online graphical modeling environment, which uses a subset of UML for the definition of modeling languages and renders model elements in SVG. GEFGWT [6] allows to build graphical editors based on GEF and could be the basis for porting GMF on the web. Also, the Eclipse-based Remote Application Platform (RAP) [7] makes easier to port Rich Client Applications (RCP) [9] to the web, since most of the SWT [13] libraries are transparently handled by the framework. RAP-based EMF editors have been provided in tools like EMF on Rails [8] or MUVITORKIT [12]. They are based on the tree viewer and a properties page offering limited user experience when a textual representation is more appropriate.

Few tools provide flexible means to define *textual* languages and they come with in-house formalisms. For example, in Concrete Editor [3], models are represented by DOM nodes with specific CSS classes. An Xtext-based online editor has been also prototyped [5]. This prototype, which is based on Orion [14], was proposed as an initial exploration on the feasibility of online textual editors. It allows editing EMF resources online but the main part of resource processing is done on the server. Within the server, each service (e.g. syntax highlighting, content assist, hover, etc.) is held by a dedicated servlet asynchronously, and the entire document is sent to the server and back again to the client at each user interaction, leading to serious performance issues. Moreover, neither methodology nor tool has been provided to show how one could bring a DSL to the web. Processing EMF-based DSL resources intensively on the client side is not reasonable also, as one has to port the entire Eclipse workbench to JavaScript. The frontier of what should be done on the client side against what should be done on the server side is an open question. DSLFORGE resource processing is distributed between the client and the server. We take advantage of two open-source technologies which are RAP and ACE [1]. The integration with RAP allows the easy integration with standard widgets such as file system navigators, file uploaders, forms, etc.

3 DSLFORGE

DSLFORGE is a framework for the generation of web textual DSL editors. The generated editors are packaged into workbench web applications which let users create, edit, and launch transformations from models online, making it possible to work simultaneously with partners or colleagues on the same resources. These online editors are also easily customizable and extensible. DSLFORGE is proposed to two main categories of users: (i) *DSL developers*, and (ii) *DSL users*. The former is given a technology to build and publish online editors on the web. The latter uses the DSL editor through lightweight applications (enterprise intranet, mobile devices, tablets, etc.).

3.1 Methodology

The framework is packaged into two features: *Tooling* and *Runtime*. DSL developers use the Tooling which contains all the needed components to generate and deploy the editor on application servers. The steps below are followed iteratively by DSL developers to get an operational online editor:

1. Design/enhance the DSL
2. If applicable, design/enhance transformations to generate some kind of artifacts from DSL instances
3. Automatically generate from the grammar the RCP and the RAP editor
4. If needed, enhance the web editor generated code. Third party plugins could provide extra functionalities.
5. Automatically package and deploy the editor.

DSL users use the deployed editor to edit and process models online, namely they can: create, edit and save models, trigger code generation from a model, and execute other actions or functionalities if the web editor was extended by third party plugins.

3.2 Examples

Generating a State Machine web editor: As a first example, we use the Martin Fowler's state machine example provided by Xtext. The example comes with a grammar allowing textual specification of state machines. Our objective is to provide a web editor that allows modeling state machines and generating java test classes from these state machines on the server using the browser. An extra functionality should be integrated for compiling and executing the test classes through the browser too.

Using DSLFORGE, as shown in Figure 1, we select the Xtext grammar to generate its online editor. At this moment we are able to select the existing transformations that will be available online for end-users. In this case, we select the existing state-machine to java transformation. The online editor is automatically packaged into a web application, together with a workspace navigator.

To execute the web application we launch the browser. Figure 2 shows how the editor handles syntax validation, content assist and how the web application is enhanced with a code generation action contributed to the Tools menu.

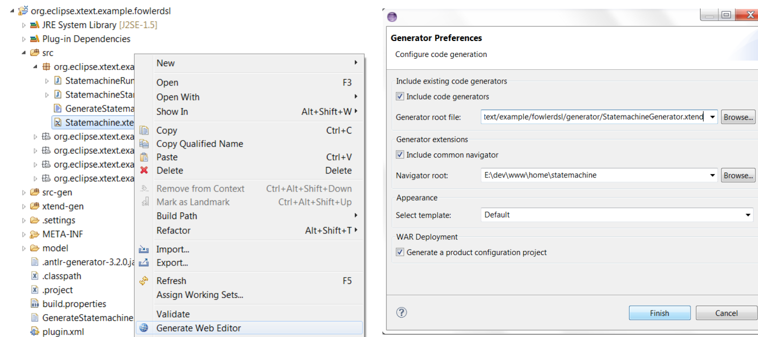


Fig. 1. DSL developers using DSLFORGE to automatically create the ready to use textual web editor

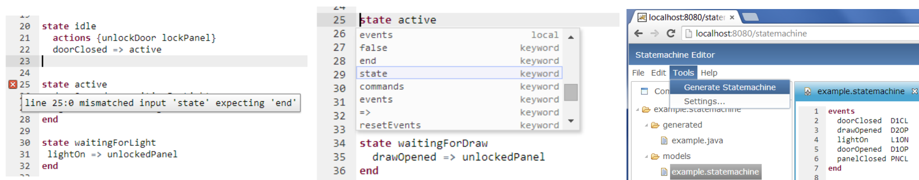


Fig. 2. DSL users using online syntax validation, content assist and model transformation launch for code generation

Semantic highlighting, scoping, history management (undo/redo), key bindings and folding are also supported by default. In addition, a custom action is integrated to the web application for compiling and executing Java classes on the server, to show an example of the online editor extensibility.

A web editor for the specification of conference websites: As a second example, we use DSLFORGE in the context of a conference DSL. This DSL allows end-users, who are not necessarily HTML/JavaScript experts, to generate and update on the server the conference or workshop website. This way, they can update the conference website wherever they are and without knowing the website technical details.

3.3 Architecture

The *Tooling* feature is shipped with Xtext, EMF, and ANTLR development features, and contains the editor generator and other contributed plugins which may be integrated with the editor into online workbenches (e.g. workspace navigator, authentication, file uploader). The *Runtime* contains the standard RAP target platform together with EMF target components, Xtext runtime, and additional plugins used during the execution of the editor. The editor can be integrated with any third party RAP-compatible plugin.

DSLFORGE follows the RAP standard architecture and lifecycle. Indeed, each generated editor widget has its counterpart in JavaScript. Resource processing is dispatched between the server and the client. The communication between the client and the server uses the JSON format. Multi-threading is used on the client side to handle the computationally expensive resource processing. Indeed, dedicated workers parse the user input and feed-back the UI with annotations. Shared workers are used to manage the global index which maintains available the cross references to all editors opened within the same user session. On the server side, parsing and validation is triggered when saving the resource. Asynchronous server pushes are used to notify the client about workspace change events and server-side resource validation. The generated web applications have been successfully deployed on two application servers (Tomcat and Jetty). Future work will include securing the communication between the client and the server and enhancing the management of resource access privileges.

References

1. Ace: Ace, Cloud 9 IDE (2014), `http://ace.c9.io`
2. AUTOSAR: AUTomotive Open System Architecture (2014), `http://autosar.org`
3. ConcreteEditor: Concrete Editor (2014), `http://concrete-editor.org`
4. Dirix, M., Muller, A., Aranega, V.: GenMyModel: An Online UML Case Tool. In: ECOOP (2013)
5. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: SPLASH/OOPSLA Companion. pp. 307–309 (2010)
6. GEFGWT: GEF in the web browser (2014), `http://www.gefgwt.org`
7. Lange, F.: Eclipse Rich Ajax Platform: Bringing Rich Client to the Web. Apress, Berkely, CA, USA, 1 edn. (2008)
8. López-Landa, R., Noguez, J., Guerra, E., de Lara, J.: EMF on Rails. In: Proc. ICSOFT. pp. 273–278 (2012)
9. McAffer, J., Lemieux, J.M.: Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications. Addison-Wesley Professional (2005)
10. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-specific Languages. ACM Comput. Surv. 37(4), 316–344 (Dec 2005)
11. MetaCase: MetaEdit+ (2014), `http://www.metacase.com/`
12. Modica, T., Biermann, E., Ermel, C.: An Eclipse Framework for Rapid Development of Rich-featured GEF Editors based on EMF Models. In: GI Jahrestagung (2009)
13. Northover, S., Wilson, M.: SWT: The Standard Widget Toolkit, Volume 1. Addison-Wesley Professional, first edn. (2004)
14. Orion: Orion Project (2014), `http://www.eclipse.org/orion`
15. Saldamli, L., Fritzson, P., Aronsson, P., Bunus, P., Engelson, V., Johansson, H., Karström, A.: The Open Source Modelica Project. In: Proc. Modelica Conf. Modelica Association (2002)
16. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Mierlo, S.V., Ergin, H.: AToMPM: A Web-based Modeling Environment. In: Demos/Posters/StudentResearch@MoDELS. pp. 21–25 (2013)
17. SysML: Systems Modeling Language (2014), `http://www.uml-sysml.org`

Umple: An Open-Source Tool for Easy-To-Use Modeling, Analysis, and Code Generation

Timothy C. Lethbridge

School of Electrical Engineering and Computer Science
University of Ottawa, Canada K1N 6N5
tcl@eecs.uottawa.ca

Abstract. We demonstrate the Umple technology, which allows software developers to blend abstract models, including class-, state- and composite structure diagrams textually into their Java, C++ or PhP code. Umple is targeted at developers who prefer textual programming but also want additional abstractions in order to simplify their software and improve its quality. Umple development has involved over 60 people, mostly at Canadian and US universities, and is used to develop itself. Several systems have been umplified – converted into Umple – thus raising their abstraction and reducing code volume. The accompanying video can be found at http://youtu.be/xD-zTpB_zyQ .

Keywords: Code generation, Textual Modeling, Umple, UML, State Machines

1 Introduction

Umple is a multi-faceted technology allowing users to integrate modeling into software development straightforwardly. It supports modeling using class diagrams, state machines and composite structure diagrams, and provides a textual syntax for these that can be blended into any C-family language such as Java or C++. The resulting system can consist completely of modeling abstractions, completely of base programming language code, or a blend of either. The Umple textual form is the 'master' code for the system. Umple therefore renders the distinction between model and code somewhat moot.

Umple can display and update model diagrams as text is edited, and allows changes to diagrams to automatically change the Umple text. This is accomplished in near-real-time using UmpleOnline [1]. The developer can hence work productively, whether they prefer text or diagrams.

Umple supports a rich feature set, all documented with examples in its user manual [2], and all generating fully-operational code in Java and C++. Features include:

- UML **associations** with capabilities such as referential integrity, sorting, and enforcement of multiplicity constraints [3].
- **State machines** with unlimited nesting, concurrent activities, and a choice of implementation semantics such as having a separate thread for queuing events [4].

- **Traits** to support inclusion of model or code fragments in different contexts, or to overcome lack of multiple inheritance.
- **Active objects** and **ports** for communicating among concurrent objects (including support of parts of Autosar [5]).
- **Constraints** for invariants, state transition guards, method preconditions and ports.
- Built-in **patterns** such as singleton and immutable, with idioms for other patterns such as delegation.
- **Aspect-oriented** code injection to allow tailoring of the generated code.
- **Templates** to allow construction of string output for language generation.
- **Trace-directives** to allow dynamic analysis at the model level [6].

Umple supports mixins to allow the system to be structured in several ways. These include separating model abstractions from methods of classes, or dividing up the system in a feature-oriented manner. With mixins, multiple definitions of a given model element (e.g. a class) found separately in the Umple source files, are merged.

Umple can generate C++. Java, PhP, Ruby, SQL, metrics, documentation and various model-interchange formats such as ECore XMI, USE , TextUML and YUML. Particular focus is being placed on its ability to generate real-time systems.

Umple is under active development. Upcoming features include formal method generation, incorporation of Use Cases, requirements , and product-line capabilities. Umple has been designed to be extensible; new code generators and modeling concepts can be added easily – a process that has been going on for the last 7 years.

2 Envisioned Users

Umple is intended for general-purpose development, so anybody currently developing in one of Umple's primary supported languages can use it to enhance productivity. Anyone who wants to model using UML class diagrams, state diagrams or composite structure diagrams can also use it purely for that purpose, even if they don't intend to generate code. However, Umple is particularly targeted at the following groups:

- **Open source developers and small in-house developers**: For these communities, code is king. They may use a little UML on whiteboards, but they don't generate code due to awkward or expensive tools, or poor quality of the code generated by many tools.
- **University professors and students**: Umple is designed to be as easy to use as possible to facilitate teaching and learning, as discussed in the next section.
- **Developers who want the flexibility and the minimum of dependency**: There are several ways of structuring an Umple system, and it can be managed with many tools: Umple supports command-line, Eclipse-based and web-based development. Umple generated code doesn't require linking with third-party libraries. Although Eclipse's EMF is powerful, we avoided it to preclude dependency on Eclipse.
- **Developers who want generated code that is readable (and inspectable), but need to avoid modifying it**: In Umple, any needed user code can be injected into

the master Umple files; nonetheless, generated code can be easily read as described in Section 3.2.

- **Real-time developers**: There are several UML profiles such as Marte and Autosar for real-time use, but these are hard to master. Umple's C++ code generation (supporting various platforms) and syntax for active objects, ports and composite structure are designed to simplify basic real-time system generation.

Other open source modeling tools are available. ArgoUML [7] was once a contender but has never had full-fledged code generation, and its development has trickled to a very slow pace. Papyrus [8] is an actively-developed open-source modeling suite (According to Ohloh –Black Duck Open Hub [9] its velocity and size is about twice that of Umple), but it is tied tightly to the Eclipse ecosystem, and is more complex than what we desire for our targeted users.

3 The MDE and modeling challenges that Umple addresses

The key challenge Umple addresses is to make modeling simple and adoptable, and hence accessible to most developers. Recent papers have commented on the lack of use of modeling in practice [10], and the obstacles to adoption of modeling [11]. Umple specifically targets these obstacles as described in the following subsections:

3.1 Textual modeling that blends into code and avoids round-tripping

Although many aspects of a system can be better understood using a diagram, textual formats have advantages: They allow rapid input and editing, they allow easier version-difference analysis, and the majority of targeted users are most comfortable with textual forms. We have therefore sought ways to make all modeling constructs textual, and to ensure they are syntactically compatible with our target programming languages. Umple is not the only textual modeling tool, but it is the only tool to allow transparent blending of models with multiple programming languages.

3.2 High quality code generation

Most tools we have studied either do not generate code at all, or else do it in a half-hearted way. It is common that UML associations only generate stub methods [12].

Much Umple research has focused on ensuring generated code code is of top quality and can be used for real systems out of the box. All aspects of Umple-generated code work synergistically with other aspects, and with hand-written code.

Although it is Umple philosophy to never edit generated code, Umple generates readable code. Comments in Umple source pass through to generated code, and traceability links are injected; this enables certification and raises confidence in code correctness. There are thousands of test cases verifying all aspects of the code generation.

3.3 A highly-usable user interface

In a recent paper, we explained how for the education community, Umple's design was guided by the need to achieve usability, incrementality in learning how to model, and various other traits [13]. UmpleOnline instantly starts on the web, and generates code with one click, and diagrams with zero clicks. Umple's command-line tool works just like any other compiler that people have been familiar with for decades, and Umple's Eclipse plugin works just like any other language plugin for Eclipse.

4 Methodology for using Umple

Umple gives the user the freedom to choose their methodology. Virtually any existing approach is possible.

Umple can be used in any of the following modes, or in a hybrid of these:

Model-first: The developer can start by creating the model (either graphically or textually). Developers can then inject any necessary additional program code, such as main programs or methods for algorithms, directly into the Umple text. It is possible in Umple to specify alternative versions of code in different languages. One model can hence be used to create a C++ and a java version of the same system.

Code-first: An existing system written in a pure programming language such as Java or C++ can be 'umplified' [14]. This can be done incrementally in a series of refactorings, gradually adding Umple syntactic constructs to replace existing code. We have so far performed this on systems such as JHotDraw [15] and Weka [16].

5 Research and Development of Umple

Umple has been under development since 2007, and has been the subject of several theses and many published papers that are referenced throughout this paper.

The effectiveness of Umple has been evaluated in several contexts. For example in an experiment [17], Badreddin et al. show that developers can model with Umple's textual form just as readily as they can use the standard UML diagram form for the same model. We plan to conduct more such experiments soon.

One of the key tests of Umple is that it is developed in itself. The Umple compiler code is written in over 120 Umple files, describing over 460 classes. The project is managed using model-driven and test-driven development, as well as continuous integration. The status of the build server [18], and the most recent test run can be found online [19].

Development velocity has been increasing over the years. Most contributions have been by professors and students at ten Canadian and three US universities.

6 Conclusion

Umple is an open-source modeling suite designed to make modeling practical and accessible to a wide variety of software developers and application types.

The accompanying video (http://youtu.be/xD-zTpB_zyQ [20]) gives a walkthrough of the use of UmpleOnline for editing models, generating code and analyzing models. It also gives a quick look at the extensive user manual [2] and the architectural diagram generated by Umple of Umple itself [21]. At the Models conference the demonstration will expand on many of these aspects.

References

1. UmpleOnline, http://try.umple.org
2. Umple user manual, http://manual.umple.org
3. Badreddin, O, Forward, A., and Lethbridge, T.C. (2013), "Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity", SERA 2013, Springer SCI 496, pp. 129-149, DOI: 10.1007/978-3-319-00948-3_9
4. Badreddin, O., Lethbridge, T.C., Forward, A., Elasaar, M. Aljamaan, H, Garzon, M. (2014), "Enhanced Code Generation from UML Composite State Machines", MODELSWARD 2014, Portugal
5. Autosar, http://www.autosar.org
6. Aljamaan, H., Lethbridge, T.C., Badreddin, O., Guest, G., and Forward, A. (2014), "Specifying Trace Directives for UML Attributes and State Machines", Modelsward 2014
7. ArgoUML, http://argouml.tigris.org
8. Papyrus, http://www.eclipse.org/papyrus/
9. The Umple Open Source Project on Black Duck Open Hub, http://www.openhub.net/p/Umple
10. M. Petre, "UML in Practice", ICSE 2013, pp 722-731
11. Forward, A., Badreddin, O., and Lethbridge T.C. (2010), "Perceptions of Software Modeling: A Survey of Software Practitioners", 5th C2M:EEMDD Workshop, Paris, June 2010, http://www.esi.es/modelplex/c2m/papers.php.
12. Forward, A. (2010): The Convergence of Modeling and Programming: Facilitating the Representation of Attributes and Associations in the Umple Model-Oriented Programming Language, PhD Thesis, UOttawa, http://www.site.uottawa.ca/~tcl/gradtheses/aforwardphd/
13. Lethbridge, T.C. (2014), "Teaching Modeling Using Umple: Principles for the Development of an Effective Tool", CSEE&T 2014, IEEE Computer Society, Austria, pp 23-28
14. Lethbridge, T.C., Forward, A. and Badreddin, O. (2010), "Umplification: Refactoring to Incrementally Add Abstraction to a Program", Working Conference on Reverse Engineering, Boston, October 2010, pp. 220-224
15. JHotDraw, http://sourceforge.net/projects/jhotdraw/
16. Weka, http, //sourceforge.net/projects/weka/
17. Badreddin, O., Forward, A., and Lethbridge, T. "Model Oriented Programming: An Empirical Study of Comprehension", Cascon, ACM (2012)
18. Umple Continuous Integration Server, http://cc.umple.org
19. Umple Quality Assurance Report, http://qa.umple.org
20. YouTube, Umple Demo – Summer 2014, http://youtu.be/xD-zTpB_zyQ
21. Umple Metamodel, http://metamodel.umple.org

Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools

Michaela Rindt, Timo Kehrer, Udo Kelter

Software Engineering Group
University of Siegen
{mrindt,kehrer,kelter}@informatik.uni-siegen.de

Abstract. Many tools for Model-Driven Engineering (MDE) which are based on the widespread Eclipse Modelling Framework (EMF) [4] are developed for single tasks like e.g., generating, editing, refactoring, merging, patching or viewing of models. Thus, models are oftentimes exchanged in a series of tools. In such a tool chain, a grafical model editor or viewer usually sets the degree of well-formedness of a model in order to visualize it. Well-formedness rules are typically defined in the meta-models, yet not all tools take them into account. As a result, a model can become unprocessable for other tools. This leads to the requirement, that all tools should be based on a common definition of minimum consistency.
An obvious solution for this challenge is to use a common library of consistency-preserving edit operations (CPEOs) for models. However, typical meta-models lead to a large number of CPEOs. Manually specifying and implementing such a high number of CPEOs is hardly feasible and prone to error. This paper presents a new meta-tool which generates a complete set of CPEOs for a given meta-model. We have successfully integrated the generated CPEOs in several developer tools. The video http://youtu.be/w31AcMOd83Y demonstrates our meta-tool in the context of one of our developer tools.

1 Introduction

Model-Driven Engineering (MDE) must be supported by tools which can edit or refactor (e.g., [2]), generate (e.g., [8]), patch or merge models (e.g., [6]). These tools are typically based upon the Eclipse Modeling Framework (EMF) [4], in which a model is represented as an Abstract Syntax Graph (ASG). Frameworks such as EMF provide basic API methods to edit the ASG of a model, e.g. creating, deleting or updating single objects or attributes. However, editing ASGs with such low-level operations can violate consistency constraints on the ASG defined in a meta-model. The resulting inconsistent ASGs cannot be processed and graphically visualized by most MDE tools. In order to solve this problem, all model editing tools should use a common library of *consistency-preserving edit operations (CPEOs)*. These CPEOs must be tailored to the relevant meta-model and its constraints. Unfortunately, complete sets of CPEOs can be quite large for comprehensive meta-models such as the UML [11] meta-model. Obviously, the

manual implementation of a large number of CPEOs, e.g., as code or executable transformations, is not only tedious, but also very error-prone.

The main contribution of this paper is a meta-tool called SERGe (SiDiff Edit Rule Generator) which generates a complete set of executable CPEOs for a given meta-model. The generated sets of CPEOs can be integrated by tool developers into an MDE environment as illustrated in Figure 1. In this example, a model generator integrates the functionality of SERGe to initially generate a set of CPEOs. Afterwards, the model generator algorithm can execute these CPEOs to generate models. Moreover, the generated CPEOs comprise a common library which can be reused by further tools, e.g., a model refactor tool.

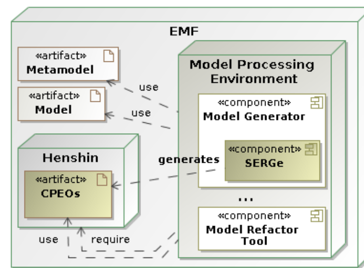


Fig. 1. Deployment Diagram showing SERGe and CPEO integration

The generation process for a set of CPEOs is fully automated and meta-model independent. SERGe is based on EMF. The generated CPEOs use EMF Henshin [5] as the transformation language and require the Henshin interpreter as the execution platform. Henshin transformation rules are in-place transformations and can contain model patterns to be found and preserved, deleted or created and also to be forbidden or required. Some consistency criteria are already enforced by EMF, e.g., type conformance, guaranteeing at most one container for each model element or a consistent handling of opposite references. With CPEOs generated by SERGe we can extend this list by (a) the preservation of multiplicity constraints and (b) the prevention of containment hierarchy cycles. The generated CPEO sets are complete in the sense that any change between two consistent models can be expressed using these CPEOs. These types of consistency constraints are sufficient to be able to graphically display models. We are not aware of an existing model editor which is usable in combination with other EMF based MDE tools and which enforces stronger consistency constraints. There can be more advanced constraints (i.e., OCL Constraints) inside a meta-model. However, they are typically not enforced by model editors and thus are not covered with SERGe so far.

SERGe provides a variety of optional configuration settings to tailor the generation process, e.g., whether to generate CPEOs for supertypes instead of for each subtype. The former will decrease the number of generated CPEOs heavily. One can also enable or disable the kinds of CPEOs (create, move, etc.) that should be generated. These are just a few configurations that are possible. SERGe has already been used extensively in different research projects, e.g., the SMG (SiDiff Model Generator) [8], SiLift [6, 10] for difference recognition between models and patching of models, and others [7]. Further possible use-cases can be tools for merging, refactoring or checking of models. More information and an example set of CPEOs can be found at the SERGe project website [9].

2 Consistency-preserving Edit Operations (CPEOs)

The easiest way to modify the Abstract Syntax Graph (ASG) of a model is to use *basic graph operations* e.g., creating or deleting single model objects. However, basic ASG operations do not consider well-formedness rules (e.g., multiplicity constraints) as defined by the meta-model. Hence, they can lead to inconsistent ASGs, which cannot be processed by other tools.

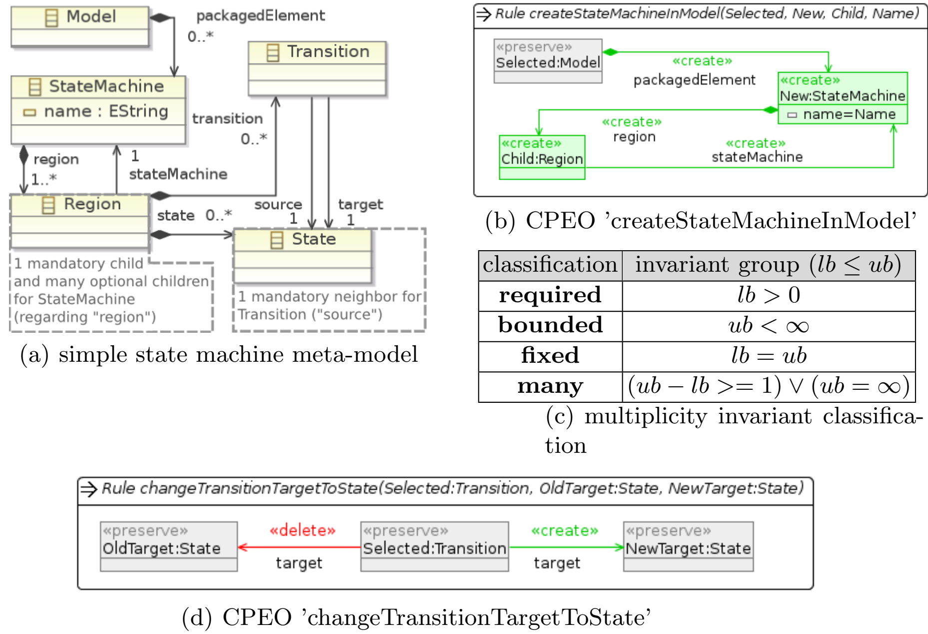
As an example, we use simplified state machines with a meta-model as shown in Figure 2(a). A `StateMachine` object must have at least one child object of type `Region`, s. the multiplicity constraint of [1..∗] of the *containment* reference `region`. A basic ASG operation which creates only a single `StateMachine` object violates this constraint. A CPEO on the other hand will create a `StateMachine` object together with a contained, mandatory child object of type `Region`.

A CPEO usually comprises several basic ASG operations, but at least those which are required to implement a consistency-preserving editing behavior. Figure 2(b) shows the CPEO mentioned above as an EMF Henshin [5] transformation rule named 'createStatemachineInModel'. Another example is provided in Figure 2(d). The example CPEO rule has a few input and output parameters: e.g., `Selected` is a placeholder for an input model object which defines the context for the transformation application. Figure 2(d) depicts the changing of an old targeted State object to a new target State in the context of a Transition. This operation contains two ASG operations, notably the deletion of an old reference `target` and the creation of a new reference `target`.

3 Generation of CPEOs

Prior to the generation phase, the meta-model is analyzed to identify the relationships between classifiers. This is done by considering incoming references of each classifier and the complete inheritance hierarchy. The source of a reference can either be a parent context or a neighbor context. This depends on the nature of a reference which is either *containment* in the first or *non-containment* in the latter case. Analogously, the target of a reference can either be identified as a 'child' or a 'neighbor'. Naturally, opposite references (e.g., `region` and `stateMachine` in the example) have to be considered together. Otherwise invalid CPEOs could be generated. An invalid operation would be the change of the reference target `stateMachine` without also changing its opposite, namely the containment reference region

Multiplicities of a reference (i.e., the upper bound (*ub*) and lower bound (*lb*)) are classified by one or more of the invariant groups shown in Figure 2(c). The meta-model analysis categorizes each relationship by considering each multiplicity invariant, which can be found on a reference. It determines if a target of a reference needs **mandatory** objects. This is the case if the reference multiplicity is classified as *required*. In a relationship between model elements, there can also exist **optional** objects. This is the case if the reference is attached with a *many* multiplicity classification. Naturally, these classifications can both apply



(a) simple state machine meta-model

(b) CPEO 'createStateMachineInModel'

classification	invariant group ($lb \leq ub$)
required	$lb > 0$
bounded	$ub < \infty$
fixed	$lb = ub$
many	$(ub - lb >= 1) \vee (ub = \infty)$

(c) multiplicity invariant classification

(d) CPEO 'changeTransitionTargetToState'

Fig. 2. Generation details

to one reference, e.g., for [1..∗] (see Figure 2(a)). This identification allows the generation algorithm to decide if a CPEO for the creation of an object may be generated or if this creation may only happen in the context of another CPEO.

During the generation phase every classifier in the given meta-model is visited. By means of the previously analyzed relationships and attributes of each considered classifier, SERGe determines which CPEO kinds will be generated for which classifier, reference or attribute. The starting point for each decision is the nature of a reference (i.e., being *containment* or *noncontainment*). [1] The following CPEO kinds can be generated depending on the occurring multiplicity invariants: creation/deletion of elements, `adding`/`removing` neighbors, `setting`/`unsetting`/`changing` of single neighbors or (default) attribute values or `moving` of children between different contexts. Mandatory children and neighbors of elements are integrated recursively inside a CPEO.

The generated CPEOs can also contain precondition checks to avoid falling below *required* multiplicities or exceeding *bounded* multiplicities. This is realized with Henshin *Positive Application Conditions (PAC)* and *Negative Application Conditions (NAC)*.

[1] We assume attributes can be handled equally to non-containment references.

4 Related Work

There are several approaches to generate executable edit operations for models beyond basic ASG edit operations. The closest approaches to ours are [1, 3].

Ehrig and Taenzer [3] address the problem to generate correct instances of a given meta-model. In such a context, only edit operations which create model elements are needed. Edit operations which delete or modify models are not provided. The problem that mandatory components cannot be simply deleted, but only be replaced, is not addressed here. The generated sets of edit operations are thus not complete in our sense. Moreover, the final result of the instance generation process must conform to the meta-model; here intermediate and inconsistent states can occur and need to be repaired afterwards. Our CPEOs on the other hand never produce inconsistent intermediate states when applied; i.e., CPEOs preserve the consistency by-construction.

Alanen and Porres [1] proposes to first convert a model into a string representation, then edit the model using a syntax-directed editor, and finally to convert it back to an ASG-based representation. Although basic consistency constraints can be preserved this way, this process is not very convenient, especially in the case of visual models.

To our best knowledge, the coverage of consistency constraints, configurability and completeness of the generated sets is not met by any other existing meta-tool to generate edit operations.

References

1. Alanen, M.; Porres, I.: A relation between context-free grammars and meta object facility metamodels; Technical Report 606, TUCS Turku Center for Computer Science; 2003
2. Arendt, T.; Taentzer, G.: A tool environment for quality assurance based on the Eclipse Modeling Framework; p.141-184 in: Automated Software Engineering 20(2); 2013
3. Ehrig, K.; Küster, J.M.; Taentzer, G.; Generating instance models from meta models; p.479-500 in: Software and Systems Modeling 8(4); 2009
4. EMF: Eclipse Modeling Framework; 2014; http://www.eclipse.org/emf;
5. EMF Henshin; 2014; http://www.eclipse.org/modeling/emft/henshin/
6. Kehrer, T.; Kelter, U.; Taentzer, G.: Consistency-Preserving Edit Scripts in Model Versioning; p.191-201 in: Proc. 28th IEEE/ACM Intl. Conf. Automated Software Engineering (ASE 2013); 2013
7. Kehrer, T.; Rindt, M.; Pietsch, P; Kelter, U.: Generating Edit Operations for Profiled UML Models; p.30-39 in: Proc. Models and Evolution (ME 2013); 2013
8. Pietsch, P.; Shariat Yazdi, H.; Kelter, U.: Generating Realistic Test Models for Model Processing Tools; p.620-623 in: Proc. 26th IEEE & ACM Inter. Conf. Automated Software Engineering (ASE 2011); ACM; 2011
9. SERGe; Project Page; 2014; http://pi.informatik.uni-siegen.de/Projekte/-SERGe.php
10. SiLift project website; http://pi.informatik.uni-siegen.de/Projekte/SiLift
11. Unified Modeling Language: Superstructure, Version 2.4.1; OMG, Doc. formal/2011-08-05; 2011

From Pen-and-Paper Sketches to Prototypes: The Advanced Interaction Design Environment

Harald Störrle

Dept. of Applied Mathematics and Computer Science
Technical University of Denmark (DTU), hsto@dtu.dk

Abstract. Pen and paper is still the best tool for sketching GUIs. However, sketches cannot be executed, at best we have facilitated or animated scenarios. The Advanced User Interaction Environment facilitates turning hand-drawn sketches into executable prototypes.

1 Introduction

Graphical user interfaces (GUIs) have two important, independent aspects: appearance and affordances (i.e., visuals vs. behavior). Existing techniques focus mostly on visual appearances, providing tools for a limited scope of visual fidelity (e.g., from GUI-builders at the high end via drawing tools such as Photoshop, Visio or PowerPoint, to sketching tools like Balsamic at the low end). The simplest possible tool for creating sketches of the visual appearance of a UI is, of course, pen and paper (PaP). It turns out, that PaP is hard to beat in terms of usability and cost/benefit ratio; thus it is our gold standard of drawing.

On the other hand, there is the behavioral aspect of GUIs. Most tools completely abstract from this aspect, restricting designers to simple mock-ups of individual scenarios made from hyperlinked pictures, or manually facilitated paper prototypes. A notable exception is Flowella (see `http://www.youtube.com/watch?v=xmuJwKYjiW0`). An early approach of combining rough sketches with interactive executability (to a degree) is the Silk/Denim line of work by Landay et al. [3,2], where users would input a sketch with a digital pen. Both Flowella and Silk/Denim are limited to very small UIs as the complexity of designs grows dramatically with the number of behavioral variants and details added.

AIDE aspires to overcome this limitation by a number of mechanisms, most notably using hierarchical state machines with concurrent substates, and syntactic layers.[1] Also, AIDE allows inputting UI designs by PaP, thus allowing to include diverse audiences in the creation process and reducing overhead and extraneous load caused by inadequate tooling. AIDE allows to complement rough sketches (i.e., PaP input) by more elaborate input in the form of XUL (XML User Interface Language, see `https://developer.mozilla.org/en/XUL`). PaP and XUL may be mixed freely, allowing scalable fidelity. AIDE provides the following advantages.

[1] See `https://www.youtube.com/watch?v=vbMblOWTRko&feature=youtube.be` or the AIDE homepage `www.compute.dtu.dk/~hsto/tools.html` for a demo.

- **Inclusive Technology** We all learn to use pen and paper from early childhood, so it is safe to assume everyone has sufficient dexterity in sketching; artistic skills are not required. In contrast, operating a computer-based tool often demands substantial expertise and/or training, which end users and domain experts may lack. Thus, using simple pen-and-paper sketches as AIDE does allows us to include virtually everybody in the UI design process.
- **Continuous Workflow** Graphic designers appreciate sketching tools: their low viscosity makes them ideal for exploring the design space (cf. [1,7,8]). However, exploration has to turn into engineering eventually, at which point developers take over from designers and visionary sketches give way to formal models and code. Often, the overall development process is greatly affected by this discontinuity of people, cultures, and methods. AIDE supports a continuous workflow integrating initial sketching with subsequent elaboration.
- **Comprehensive Design** While it is relatively easy to specify the *appearance* of an interface by a drawing, specifying its *behavior* is much more difficult (cf. [6]). In fact, the only way to completely describe arbitrary complex interface behaviors is by programming them. Obviously, integrating drawings and code in a harmonious way is difficult, further disrupting the interface design workflow. Most people are *either* good at graphic design *or* at programming, but rarely at both. Storyboards only offer a partial solution, since they allow to specify a very limited degree of behavior only (basically only linear sequences, see [5, p. 105]). So, in order to create comprehensive interface designs, we need a way to integrate both aspects of an interface, appearance and behavior.
- **Scalable Abstraction** Even small UIs may offer a large number of affordances, all of which act together to create the overall user experience. Capturing them in a prototype is expensive, time consuming, and inhibiting change. Capturing them in a more abstract specification will lead to a bloated and/or fragmented design that is difficult to reintegrate, maintain, and communicate. Establishing and maintaining consistency is an arduous and complex task [4]. So, we need a way of managing the complexity of large interface designs in such a way, that neither the clarity of the initial vision, nor the details of the interactions get lost.

2 Example

Consider an example Fig. 1. It shows a small portion of an interface design from a teaching example, the Library Management System (LMS). This toy case study is about searching for media in a library, lending them, prolonging, them, and and so on. Fig. 1 shows an interface design for the process of issuing a new reader card. The numbers in red dots are not part of the notation but have been added to improve the presentation here.

First, a dialog for entering some data appears. It contains a text field, two groups of radio buttons with two choices each, and two buttons to proceed and abort. The user inputs a reader's name, selects a few options, and eventually

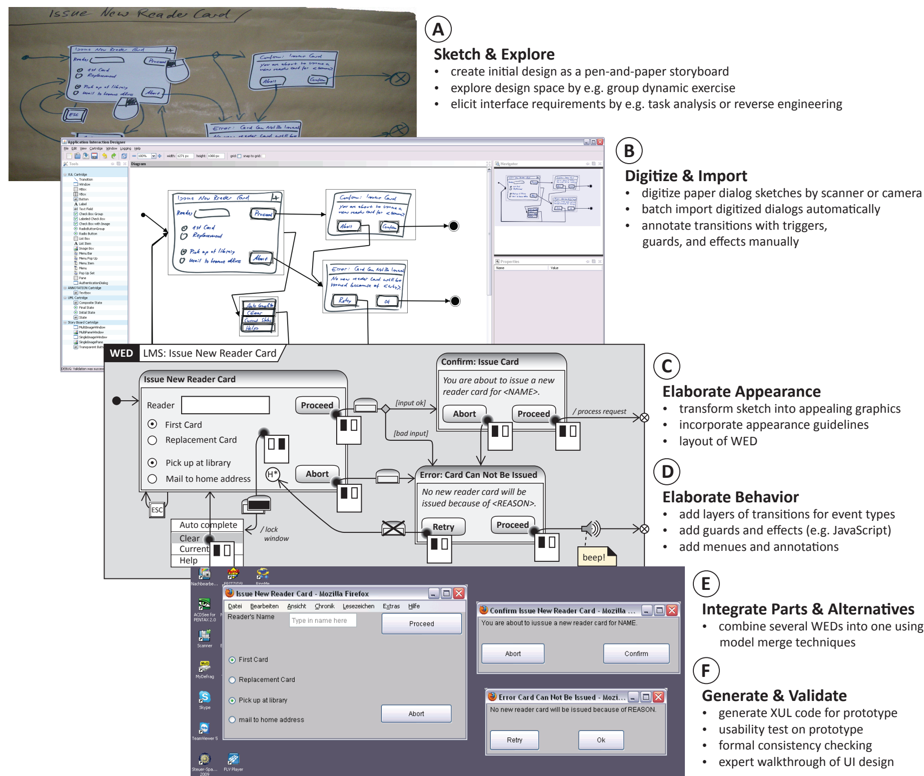


Fig. 1. Stages in the UI development life cycle using WEDs and AIDE: Starting from a traditional pen-and-paper wallpaper prototype, subsequent steps of elaboration yield a prototype with executable XUL code. Note that XUL-elements, clickable areas, and transitions have to be annotated manually in the current version of AIDE.

pressed “Proceed”. If the data validation is successful, the user must acknowledges or aborts. If the process is aborted or the validation was not successful, the user may either revise the inputs or terminate the whole process. Using the right mouse button on the window “Issue New Reader Card” will open a pop-up menu with four options.

States UI widgets like text boxes or buttons are represented as simple states. Groups of widgets and complete dialogs are modeled as composite states. As a default, only one widget or dialog can be active at any time, which in UML maps to sequential composite states. In order to achieve different behavior, concurrent composite may be used. Not all states need be visible in a design. For instance, grouping radio buttons together can be achieved by an invisible compound state. The same applies for layout elements such as vertical boxes.

Triggers Positioning a pointer over a WED element is interpreted as putting the focus on that element. Technically, the corresponding state configuration tree

is activated. Any user events issued subsequently will be interpreted by that tree, bottom up. For instance, positioning the mouse pointer over the “Abort” button and pushing the left mouse button (a) issues the event single left click in the state “IssueNewReaderCard.Abort” and triggers the transition emanating that state. Likewise, moving the pointer over “Issue New Reader Card” and pressing the Escape key (b) will reset the corresponding state. Any (user) actions the UI affords may be used as triggers.

Guards Transitions may carry a guard that enforces the respective condition to be true before the transition is taken. Guards may refer to environment variables that may be used to represent a hidden UI state such as a mode.

Effects Then, the effect of the transition (modeling a UI action) is executed and its target state is entered. Effects may be described by plain text (a), code snippets, invocation of library functions, or maybe visualized by an icon (b). Probably the most common effects are opening a new window (a), closing one (b), or opening a modal dialog (c).

Entering States When entering a composite state C for the first time, the substate to be entered is determined by the initial state. Reentering C will reset its state configuration unless a history state is added to C, which restores the state configuration in effect at the time of exiting C. Exit Points (and and Entry Points) help achieving a modular design (this is regular UML 2.2 syntax). Exiting a state (or state machine) automatically exits all sub states, i.e., corresponding windows are closed by reaching a final node.

Secondary Notation Annotations and comment boxes may be used freely; they are represented as UML PseudoStates.

There is no semantic difference between UML 2 state machines and a UI design in this form: every construct in a UI design may be mapped to a UML state machine construct. These mappings are typically very straightforward, but have to be added manually in the current version of AIDE.

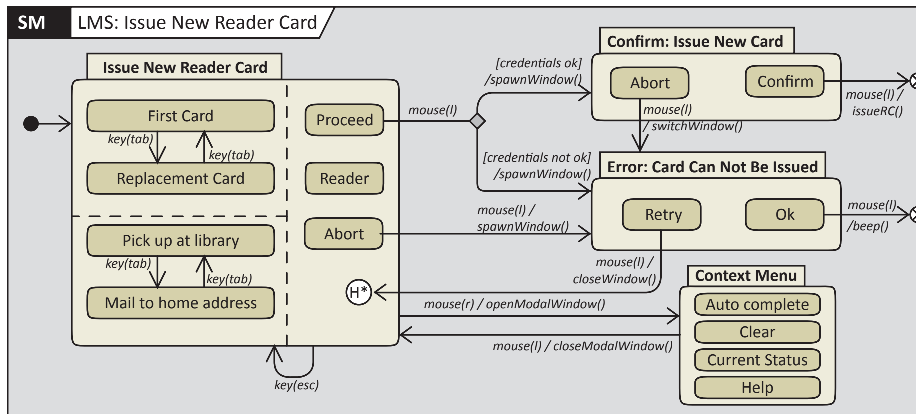


Fig. 2. This UML state machine is yielded by stripping all appearance cues and elaborating effects to procedure calls.

3 The AIDE Tool

The Advanced Interaction Design Environment (AIDE) is a highly modular platform independent direct interaction tool set for creating WEDs, refining and elaborating them in a methodical fashion, supporting distributed concurrent group work, and generating working prototypes from WEDs. AIDE has been under development since 2006, with 25 students at Innsbruck University, Munich University, and the Technical University of Denmark contributing a total of approximately 10,000 work hours to it. Step Ⓑ in Fig. 1 is actually a screenshot of the AIDE tool.

AIDE is created using pure Java, using Piccolo2D, jEdit, JGoodies Looks, the Tango Iconset, VLDocking, and JAXB for persistency. AIDE follows a strict separation of logic and presentation. Extensibility of AIDE is ensured by a cartridge mechanism that encapsulates the visual appearance of elements and functions associated with them. Cartridges may be dynamically loaded or unloaded. Apart from the basic cartridges of UML state machines and annotations, there are currently cartridges for the XML User Interface Language (XUL), and for importing hand-drawn storyboards. XUL is used e.g., for defining the UIs of Mozilla projects such as Firefox and Thunderbird. AIDE provides XUL export and integrated simulation. Finally, AIDE also offers unbounded Undo/Redo, user definable roll-back points, tear-off-menus to support very large screens, a locator map, sophisticated zoom and scrolling functions, and multiple views on elements.

References

1. Bill Buxton. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann, 2007.
2. James A. Landay and Brad A. Myers. Interactive Sketching for the Early Stages of User Interface Design. Technical Report CMU-HCII-94-104, Carnegie Mellon University, July 1994.
3. James A. Landay and Brad A. Myers. Sketching Interfaces: Toward More Human Interface Design. *IEEE Computer*, 34(3), March 2001.
4. Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Inf. Softw. Technol.*, 51(12):1631–1645, December 2009.
5. Scott McCloud. *Understanding Comics: The Invisible Art*. HarperPerennial, 1993.
6. Brad A. Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew J. Ko. How Designers Design and Program Interactive Behaviors. In Paolo Bottoni, Mary Beth Rosson, and Mark Minas, editors, *Proc. IEEE Symp. Visual Languages and humand Centric Computing (VL/HCC'08)*, pages 177–184. IEEE Press, 2008.
7. Mark W. Newman and James A. Landay. Sitemaps, Storyboards, and Specifications: a Sketch of Web Site Design Practice. In *Proc. 3rd Conf. Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS'00)*, pages 263–274. ACM, 2000.
8. Y.Y. Wong. Rough and Ready Prototypes: Lessons from Graphic Design. In *SIG Computer Human Interaction (SIGCHI'92)*, page 685, 1992.

Concern-Driven Software Development with jUCMNav and TouchRAM

Nishanth Thimmegowda[1], Omar Alam[1], Matthias Schöttle[1],
Wisam Al Abed[1], Thomas Di'Meco[2], Laura Martellotto[2],
Gunter Mussbacher[3], Jörg Kienzle[1]

[1]School of Computer Science, McGill University, Montreal, Canada
[2]Polytech Nice-Sophia, Sophia Antipolis, France
[3]Dep. of Electrical and Computer Engineering, McGill University, Montreal, Canada
{Nishanth.Thimmegowda,Omar.Alam,Matthias.Schoettle,Wisam.Alabed}
@mail.mcgill.ca, {Thomas.DiMeco,Laura.Martellotto}@gmail.com,
{Gunter.Mussbacher,Joerg.Kienzle}@mcgill.ca

Abstract A concern is a unit of reuse that groups together software artifacts describing properties and behaviour related to any domain of interest to a software engineer at different levels of abstraction. This demonstration illustrates how to specify, design, and reuse concerns with two integrated tools: *jUCMNav* for feature modelling, goal modelling, and scenario modelling, and *TouchRAM* for design modelling with class, sequence, and state diagrams, and for code generation. For a demo video see: `http://www.youtube.com/watch?v=KWZ7wLsRFFA`.

1 Introduction

In contrast to the focus of classic Model-Driven Engineering (MDE) on models, the main unit of abstraction, construction, and reasoning in Concern-Driven Software Development (CDD) is the *concern* [2]. CDD seeks to address the challenge of how to enable broad-scale, model-based reuse. A concern is a unit of reuse that groups together software artifacts (models and code, henceforth called simply models) describing properties and behaviour related to any domain of interest to a software engineer at different levels of abstraction.

A concern provides a three-part interface. The *variation interface* describes required design decisions and their impact on high-level system qualities, both explicitly expressed using feature models and goal models in the concern specification. The goal models used in CDD are called impact models. The *customization interface* allows the chosen variation to be adapted to a specific reuse context, while the *usage interface* defines how the functionality encapsulated by a concern may eventually be used.

Building a concern is a non-trivial, time consuming task, typically done by or in consultation with a domain expert (subsequently called the *concern designer*). On the other hand, reusing an existing concern is extremely simple, and essentially involves 3 steps for the *concern user*:

1. Selecting the feature(s) of the concern with the best impact on relevant goals and system qualities from the variation interface of the concern,
2. Adapting the general models of features of the concern that were selected to the specific application context based on the customization interface, and

3. Using the functionality provided by the selected concern features as defined in the usage interface within the application.

In general, MDE approaches rely heavily on tool support. Tool support is even more important in the context of CDD, in particular for the concern user:

- When selecting the set of features of a concern that best meets the requirements of the application under development, a concern user needs to be able to perform trade-off analysis between different variations/implementations of the needed functionality. To do that efficiently, a tool is needed that performs real-time impact analysis of feature selections.
- Once a selection is made, a tool is needed that composes the models that realize the selected features to yield new models of the concern corresponding to the desired configuration.
- When adapting the generated concern models to the application context, the concern user must map customization interface elements from the concern to application-specific model elements in the application. Tool support is helpful to ensure that the mapping is specified correctly.
- Once the concern model is customized, a tool can help to ensure that the functionality provided by the concern is correctly used.

This demo illustrates CDD in practice by demonstrating Concern-Driven Development with two integrated tools: jUCMNav and TouchRAM. Section 2 of this paper briefly describes how the two tools were modified to support concerns. Section 3 outlines concern development and concern reuse with the two tools.

2 Integration of jUCMNav and TouchRAM

jUCMNav is a requirements engineering tool created in 2005 for the elicitation, analysis, specification, and validation of requirements with the User Requirements Notation (URN). jUCMNav combines two complementary views: one for goals provided by the Goal-oriented Requirement Language (GRL) and one for scenarios provided by the Use Case Map (UCM) notation. Recently, jUCMNav was extended to support combined goal and feature modelling and their integrated analysis based on GRL semantics [5]. Over the last two years, jUCMNav was demoed at RE [3,4], the 2013 SDL Forum, and the 2013 iStar workshop.

TouchRAM is a multitouch-enabled tool for agile software design modelling aimed at developing scalable and reusable software design models using UML class, sequence, and state diagrams. It exploits model interfaces and aspect-oriented model weaving to enable the concern user to rapidly apply reusable design concerns within the design model of the software under development. The user interface features of the tool are specifically designed for ease of use, reuse, and agility. TouchRAM was introduced initially at SLE 2012 [1], and later demonstrated at Modularity:aosd 2013 and 2014 [7] and MODELS 2013 [6].

For CDD, jUCMNav and TouchRAM are complementary to each other. jUCMNav covers the requirements modelling side, providing support for feature and goal modelling necessary for the definition of a concern's variation interface. Furthermore, jUCMNav supports scenario modelling with the Use Case Map notation. TouchRAM on the other hand provides support for detailed design

modelling and code generation. The first step in integrating the two tools was to define the concepts of CDD in a metamodel – the CORE (Concern-Oriented REuse) metamodel. It defines:

- *Concern*, which groups together a set of models,
- *Concern Interface*, i.e., the variation interface, the customization interface, and the usage interface,
- *Concern Reuse*, i.e., a concept that is used to store the selected features and the customization whenever a concern is reused, and
- *Feature,* with associations to connect the models that realize the feature and the concern reuses that the feature specifies.

Next, the existing metamodels of jUCMNav and TouchRAM had to be made compliant with CORE. This involved declaring classes in the metamodel of jUCMNav and TouchRAM to subclass classes in CORE. In jUCMNav, the new subclasses of CORE concepts also subclassed existing URN classes. This allowed the same analyses performed on URN models to also be performed on CORE-compliant URN models. Since none of the classes and properties that already existed in the old metamodels had to be removed or modified, the tools are still backwards compatible, i.e., they can still read models created with previous versions of the tools. In addition, the two tools are now file compatible, i.e., it is possible to create a concern and define features for it in jUCMNav and then work with it in TouchRAM, and vice versa.

The current version of jUCMNav can be downloaded from `http://www.softwareengineering.ca/jucmnav`, the current version of TouchRAM from `http://cs.mcgill.ca/~joerg/SEL/TouchRAM.html`

3 Concern-Driven Development in Action

The demo at MODELS shows how to first build requirements and design models for the *Authentication* concern with jUCMNav and TouchRAM, and then how simple it is to reuse the *Authentication* concern within a banking application.

3.1 Developing a Concern

First, the concern is created in jUCMNav, and the features of the concern are specified. The left picture in Fig. 1 shows that *Authentication* has a mandatory *Authentication Means* feature that may either be *Password* or *Biometrics*. *Biometrics* must at least include *Retinal Scan* or *Voice Recognition*. An optional subfeature of *Password* is *Password Expiry*. If desired, unsuccessful authentication may lead to *Access Blocking* and long idle periods to *Auto Logoff*.

The right picture in Fig. 1 shows how goal models are used to specify the relative impact that each feature has on non-functional requirements and qualities (e.g., security). The model shows that *Retinal Scan* and *Voice Recognition* are the strongest *Security* mechanisms, even stronger than *Password* with *Password Expiry*, *Access Blocking*, and *Auto Logoff*. Once the impacts are specified, jUCMNav allows the concern designer to interact with the feature model and evaluate the impacts of different configurations (sets of feature selections) of the concern (visualized with a color scheme and evaluation values from 0 to 100).

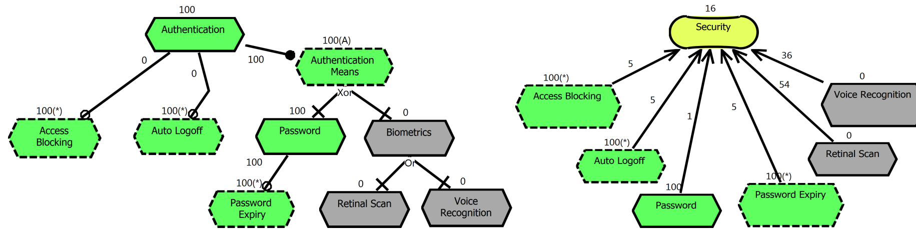


Figure 1. Feature and Impact Modelling and Analysis in *jUCMNav*

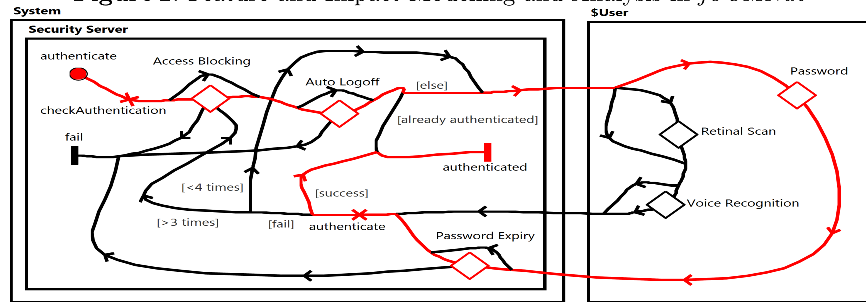


Figure 2. Scenario Modelling and Analysis in *jUCMNav*

In jUCMNav, it is also possible to specify scenarios that describe how a user would interact with the *Authentication* concern with the Use Case Maps notation as shown in Fig. 2. The modeller can associate features with path elements in the scenario, which makes it possible to automatically visualize the scenario traversal for a given feature configuration (by highlighting the scenario path in red).

Next, the concern designer uses TouchRAM to create detailed design models and associate them with each feature of the concern. Aspect-oriented techniques such as class merge and sequence diagram advising are used to modularize the structural and behavioural properties of each feature. For instance, if *Authentication* defines a class called `Credential`, the design of *Password* adds a `String` attribute for the password in the class, and the design of *Password Expiry* adds a `Date` attribute that stores when the password was last changed as well as additional behaviour to update this attribute whenever the password is changed.

3.2 Reusing a Concern

When a modeller creates a specific application for which *Authentication* is of relevance, the modeller in the role of the concern user opens the *Authentication* concern and selects the desired features from the feature model. While interacting with the feature model, the impacts resulting from the current selection are constantly updated. When a satisfactory selection has been made, TouchRAM composes all design models of the selected features together to produce a detailed design model for this specific configuration. The modeller is then

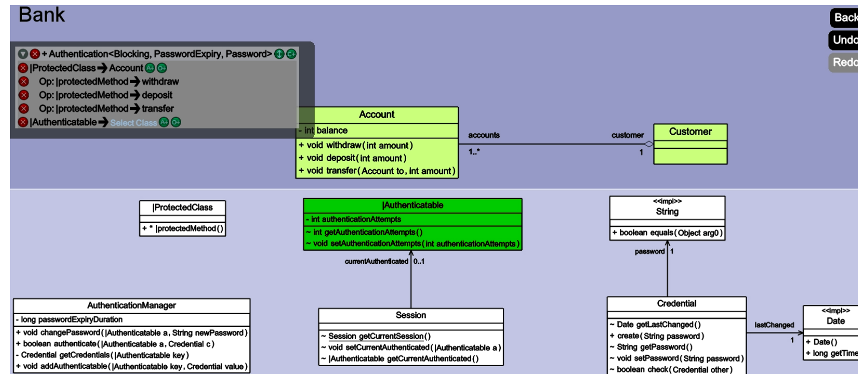


Figure 3. Reusing Authentication in *TouchRAM*

presented with a mapping view as shown in Fig. 3 that allows the modeller to customize the *Authentication* concern to her specific needs by establishing mappings between the model elements in the concern and the application model. In our case, the software is a simple Banking application, and the modeller wants to enforce authenticated access to accounts. Therefore, `Authenticatable` maps to the `Customer` class, `ProtectedClass` to `Account`, and `protectedMethod` to `withdraw`, `deposit`, and `transfer`.

Once the customization is completed, the designer of the bank application can instruct TouchRAM to compose the entire application model to yield the combined structure and behaviour of the system. From that, TouchRAM allows the developer to generate executable Java code.

In future work, we are planning to develop several, more complex concerns to empirically validate the integration of the jUCMNav and TouchRAM tools.

References

1. Al Abed, W., Bonnet, V., Schöttle, M., Alam, O., Kienzle, J.: TouchRAM: A multitouch-enabled tool for aspect-oriented software design. In: SLE 2012. pp. 275 – 285. No. 7745 in LNCS, Springer (October 2012)
2. Alam, O., Kienzle, J., Mussbacher, G.: Concern-Oriented Software Design. In: MODELS 2013. LNCS, vol. 8107, pp. 604–621. Springer (October 2013)
3. Amyot, D., Leblanc, S., Kealey, J., Kienzle, J.: Concern-Driven Development with jUCMNav. In: RE 2012, Chicago, USA. pp. 319 – 320. IEEE CS (September 2012)
4. Liu, Y., Su, Y., Yin, X., Mussbacher, G.: Combined Goal and Feature Model Reasoning with the User Requirements Notation and jUCMNav. In: RE 2014, Karlskrona, Sweden. IEEE CS (August 2014)
5. Liu, Y., Su, Y., Yin, X., Mussbacher:, G.: Combined Propagation-Based Reasoning with Goal and Feature Models. In: MoDRE 2014 (August 2014)
6. Schöttle, M., Alam, O., Ayed, A., Kienzle, J.: Concern-Oriented Software Design with TouchRAM. In: Demonstration Paper at MODELS 2013. CEUR Workshop Proceedings, vol. 1115, pp. 1 – 6 (october 2013), `http://ceur-ws.org/Vol-1115/demo10.pdf`
7. Schöttle, M., Alam, O., Garcia, F.P., Mussbacher, G., Kienzle, J.: TouchRAM: A Multitouch-enabled Software Design Tool Supporting Concern-oriented Reuse. In: Companion of Modularity:2014. pp. 25–28. ACM (2014)