Software Language Modernization

Patrick Neubauer

Dissertation Proposal
Business Informatics Group, Vienna University of Technology, Austria
neubauer@big.tuwien.ac.at

Abstract. The software engineering community has developed a plethora of software languages. On the one hand, instances of machine-oriented languages are considered to be fully machine readable but often less comprehensible for humans. On the other hand, human-oriented languages have been explicitly built with comprehensibility in mind but often lack the advantages of machine-oriented ones. Many of today's software languages fit into the category of machine-oriented languages, lacking comprehensibility and consequently presenting difficulties in development, re-use, and maintenance. Tackling these limitations by evolving machine-oriented languages into human-oriented languages represents a non-trivial undertaking but opens up possibilities of advancing former language concepts with latter representations. Therefore, the aim of this dissertation is to evolve machine-oriented languages into human-oriented languages by facilitating Model-Driven Engineering (MDE) techniques, such as model transformations, that allow to map between several concrete appearances of a language based on an abstract language definition. In order to evaluate the impact of modernizing software languages, a user study on effects of the proposed modernization process will be performed as well as a concrete metric suite for evaluating languages and their evolutions will be investigated.

1 Problem Description

Software Language Engineering (SLE) [26, 48] constitutes an emerging subfield of Software Engineering focusing mainly on language definition, parser building, and compiler construction [46]. Traditionally, there was a strong focus in language design to provide machine-readable and executable formats. In the last decades, there is a shift to pay more attention also to concrete syntax, i.e., notation, of a language due to several reasons, for example:

- more and more programs need to be maintained by humans;
- powerful tools [12,18] have emerged to define textual concrete syntax, graphical concrete syntax, and even a mixture of both without starting from scratch when defining the languages, cf. language workbench [16], metamodeling environments [44];
- the rise of languages tailored to the need of non-programming domain experts.

However, while the distinction between the notion of software languages, from which some of them are considered machine-oriented and others human-oriented, is known [15], it is not clearly described. It is suggested that, for a software language to become successful, both sides of a coin — from which one represents the machine side and the other the human side — have to be designed very carefully [48]. To be more precise, the machine's side is represented by abstract syntax and the human's side is represented by a concrete syntax emphasizing on perception, cognition, and usability in terms of the human brain as a processing unit of symbols [32].

With the introduction of the fully machine-processible Extensible Markup Language (XML) [9] in 1998, the World Wide Web Consortium (W3C) established a tremendous leap towards easing the design of software languages, leveraging the idea of having a generic editor, parser, and validation methodology. In 2004, W3C recommended XML Schema Definition (XSD) [42] as a new standard to formally describe elements in an XML document and hence also provide the capability for verifying the conformity of an XML document to an XSD specification. During the last decade a vast amount of markup languages have been implemented using XML Schemas. Rich Site Summary (RSS), Simple Object Access protocol (SOAP), Scalable Vector Graphics (SVG), Extensible Hypertext Markup Language (XHTML), and Business Process Execution Language (BPEL) represent prominent examples of markup languages.

With the rise of MDE [40], in which models are treated as first class citizens, several new methodologies emerged. For example, using Model-Driven Language Engineering (MDLE) [27], i.e., the application of MDE techniques to build software languages, Domain-Specific Modeling Languages (DSMLs) [17, 30] can be tailored to accurately match domain syntax and semantics while general-purpose notations seldom express design intent or domain concepts [40]. DSMLs can be equipped with elements that directly relate to familiar domains flattening the learning curves and therefore also increase the scope of subjects being able to work with such representations. Xtext [12] represents an exemplary MDLE tool to build, not only textual DSMLs from metamodels, but also advanced editor support by, for example, introducing syntax coloring, content assist, validation, and quick fix [13].

XML-based markup languages and DSMLs are residing in two different technological spaces [29] following distinctive goals. DSMLs are known for their success factors in terms of maintainability and re-use [19]. While DSMLs offer substantial gains in comprehensibility and maintainability when compared with markup languages, XML-based languages are fully machine readable and very flexible in contrast to DSMLs which should shield the user from mistakes. Furthermore, XML-based languages, also referred to as machine-oriented models [15], are specified in a generic concrete textual syntax, i.e, an angle bracket-based syntax, that can not be adapted to different user or domain requirements and appears verbose and complex in terms of human comprehension [2].

Moreover, Model-Driven Software Modernization (MDSM) [21], represents another emerging MDE discipline, that is loaded with approaches that focus on

the modernization of legacy software. In more detail, to evolve existing software artifacts, they are transformed into models through the use of model transformations typically yielding more abstract and platform-independent representations. Such representations are then being used in the process of migrating legacy software to new platforms.

By marrying MDLE and MDSM, techniques found in both fields could be exploited, i.e., the modernization of software languages by facilitating MDE, for example, to built a hybrid of markup languages and modeling languages intending to solve individual limitations regarding the orientation towards machines and humans. Equally, as with General-Purpose Programming Langauges (GPLs) and DSMLs, the choice of which one to use for software engineering tasks is not a binary one, i.e., a software engineering project can use both kinds of languages for the construction of software artifacts. MDLE, providing an abstract concrete syntax representation for a language, represents a promising starting point in building a hybrid between machine-oriented and human-oriented software languages such as a mapping to the original XML structures.

Therefore, the objective of the proposed dissertation is to address the challenges imposed by the modernization of software languages in terms of the evolution of machine-oriented languages to human-oriented languages. This entails not only the transformation of languages in a hybrid approach but also the automated transformation of sentences defined in such languages. Therefore, benefits of both technical spaces are kept intact. The purpose of evolving from machine-oriented languages to human-oriented languages is to approach individual barriers that the former languages impose on characteristics such as human perception, cognition, and usability and the latter languages have in terms of, e.g., executability.

Individual challenges tackled in the proposed dissertation include but are not limited to:

i In order to establish a common ground to evaluate a software languages' orientation towards machines and humans, high-level properties that describe the objectives of both machine-oriented software languages and languages that are human-oriented have to be identified and formalized. Furthermore, such properties have to be measurable in terms of case studies, user studies, and/or other experiments. However, not only properties are required, but also appropriate methods of analysis have to be identified, constructed, or a combination of both to allow conducting studies evaluating the formalized software language properties.

ii For the purpose of modernizing XML-based markup language expressed as XSDs, capabilities and barriers of existing MDLE and MDSM concepts, techniques, and tools that overlap with the intended approach, have to be identified and bridged, respectively. In more detail, a technique has to be found that bridges the XML Schema definition with a DSML in terms of a hybrid approach that allows to produce sentences on both sides and can be applied

to any XML-based markup language.

iii Individual solutions have to be formalized in a generic framework that is able to modernize software languages and in particular XML-based markup languages into a hybrid solution intending to solve individual limitations of a software language, i.e., its orientation towards machines or humans. This not only includes appropriate language transformations, but also transformation of sentences defined in such languages. Furthermore, state-of-the-art MDE frameworks and tools have to be, not only, extended to support the discovered generic solutions, but also to support extended editor features, such as syntax coloring, content assist, validation, and quick fix, intending to increase comprehensibility and therefore maintainability.

2 Expected Result

Since the increasing success of models in software development and model transformations in MDE during software development activities, such as automated forward engineering, there is no reason not to benefit from the same infrastructure to automate other tasks. In other words, in an advanced phase of MDE in which models and transformations are still central to the process of creating software, they also start to become an integral part of the developed system itself. Here the idea of transformation manipulation arises in which transformations themselves are generated and handled by MDE techniques such as transformation especially HOTs and expoiting meta-modeling frameworks offering such tools to derive abstract syntax, concrete syntax, as well as the model-driven generation of tools. Thus, transformations are represented by transformation models that conform to a transformation metamodel. This theory can be applied recursively and create, reshape, enhance transformations, i.e., transformation models, in the same way as models to reduce the time spend for developing and executing transformations. M. Tisi et al. [43] defines HOTs as follows:

"A higher-order transformation is a model transformation such that its input and/or output models are themselves transformation models."

Fig. 1 depicts a draft of the overall approach in mind to facilitate the advantages of HOTs when bridging the technological spaces introduced by different meta languages. In particular, when considering the example introduced in Section 1, the goal is to bridge XML-based markup languages (cf. *Language 1* in Fig. 1) with modeling languages (cf. *Language 2* in Fig. 1) starting from a transformation at meta language level (cf. *Transformation ML* in Fig. 1). Then, by instantiating the transformation (cf. *Transformation Execution* in Fig. 1) at language level, individual languages, conforming to their individual meta languages (cf. *Meta Language 1* and *Meta Language 2* in Fig. 1), can be transformed. Finally, at the sentence level, individual sentences (cf. *Sentence 1* and *Sentence 2* in Fig. 1) can be transformed — again by instantiating the transformation (cf.

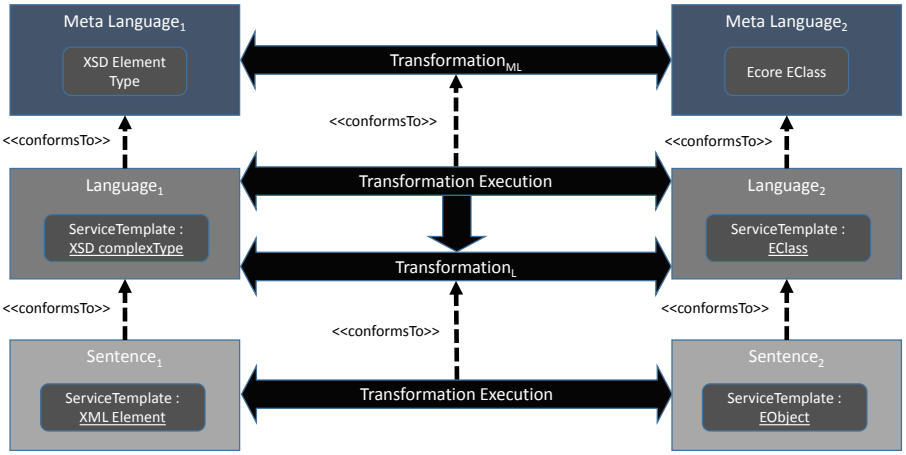


Fig. 1. Modernizing Software Languages at a Glance

Transformation L) which has been produced by the execution of *Transformation ML*.

The general idea to tackle the problem introduced in Section 1 is to approach the challenges imposed by modernizing software languages using MDE techniques. To address the challenges mentioned in Section 1, the following contributions are proposed:

Contribution 1: Property and Evaluation Metrics Suite. The first contribution targets challenge one by performing a literature review on individual machine-oriented and human-oriented characteristics of a software language and the evaluation thereof. This includes establishing a common ground to evaluate a software languages' orientation towards machines and humans by formalizing high-level properties that describe the objectives of both machine-orientation and human-orientation. Furthermore, an evaluation mechanism for software languages in terms of formalized properties is provided by a set of analysis methods. Both properties and analysis methods thereof are implemented in a metrics framework that can be executed in form of case studies, user studies, or other experiments, like machine-analysis of software language artifacts, or a combination of such.

Contribution 2: Individual Modernization Techniques and Feasibility Study. To approach the second challenge, capabilities and barriers of existing MDLE and MDSM approaches, tools, and frameworks that overlap with the intended approach are married and bridged by formalizing individual techniques, respectively. For example, a technique has to be developed to transform meta languages. Furthermore, for tackling the challenge of transforming sentences within

a language starting from the meta language level, existing techniques like HOT are evaluated. Fitting techniques and the extension thereof are then used to implement the transformation of language sentences starting from meta language level. Before individual techniques are merged, a case study will be performed that highlights the feasibility of individual techniques.

Contribution 3: Generic Framework and Evaluating Study. Finally, contribution three targets challenge three by implementing a generic framework facilitating existing MDE frameworks and tools, not only in terms of merging individual techniques, but also by further extending a language's comprehensibility provided by advanced editor features, such as syntax coloring, content assist, validation, and quick fix, and their implementation in a most generic way. Moreover, to show that arbitrary XML-based markup languages can be modernized into hybrid languages that are both machine-readable and human-comprehensible, a user study is performed evaluating the developed generic framework.

3 Methodological Approach

The methodological approach used in this thesis will be based on the design science research guideline from Hevner et al. [45]. The mentioned guidelines will be followed as follows:

1. Design as an Artifact. The thesis aims to provide a new approach from markup languages to modeling languages providing both a theoretical foundation and an implementation. The following artifacts are expected as outcome of this thesis:

- A case study intended to evaluate the feasibility of individual techniques built from marrying MDLE, MDSM, and/or other appropriate approaches in the sense of this thesis.
- A metrics framework formalizing properties and analysis methods to evaluate the orientation of a software language towards machines and humans.
- A generic framework implementation that not only merges the individual techniques but also implements advanced editor features in a most generic way if such are known to improve the comprehensibility of a software language.
- A user study to evaluate the impact of the generic framework implementation.

2. Problem relevance. Favre [15] highlights that an important property of technological spaces is that there is no "best technological space" but the choice depends on the problem at hand. Moreover, technological spaces are not islands and the notation of bridges between spaces is an important issue that is worth to be facilitated. For example, while there are machine-oriented models, like XML-based markup languages, on one side of a spectrum, there are human-oriented models, like UML or Human-Usable Textual Notation (HUTN)-based

DSMLs, on the other side of the spectrum. Through the construction of bridges between machine-oriented models and human-oriented models, such models gain comprehensibility and thus maintainability.

3. Design Evaluation. The artifacts developed in the course of this thesis will continuously be evaluated by case studies according to Runeson and Höst [37] to find out not only whether they are useful at all, but also to guide the direction of further development.

4. Research Contributions. The main research contribution of this thesis will be the experience in a new generic approach to bridge the technological space between XML-based markup languages and modeling languages as well as the theoretical foundations and implementation of it. For a more detailed description on intended contributions see Section 2.

5. Research Rigor. There exists much work in the area of bridging technological spaces [29] which will be taken into account when writing this thesis. Thus, initially existing literature and tools will be surveyed systematically [25] not only for their scientific contributions themselves, but also for their evaluation strategies where some of them might be applicable for this approach as well.

6. Design as a Search Process. All artifacts in this thesis will be constructed iteratively. First, all techniques and solutions will be evaluated manually on systematically developed scenarios to give an initial hint on the general feasibility. The approach then will be generalized capturing these simple scenarios. If this is possible, more complex scenarios will be evaluated and automated until either the automation becomes to difficult or not possible at all or the initially set target is reached. During this search, alternative paths might be recognized and taken.

7. Communication of Research. The research will be disseminated in through well known communication channels in the MDE, software language engineering as well as the general computer science community. There are several conferences and journals which might be interesting targets for this kind of work:

- International Conference on Model Driven Engineering Languages and Systems (MODELS)
- European Conference on Model Foundations and Applications (ECMFA)
- International Conference on Automated Software Engineering (ASE)
- International Conference on Software Language Engineering (SLE)
- International Conference on Advanced Information Systems Engineering (CAISE)
- Journal of Software and Systems Modeling (SOSYM)
- IEEE Transactions on Software Engineering (TSE)
- Transactions on Software Engineering and Methodology (TOSEM)
- Journal of Systems and Software (JSS)
- Information and Software Technology (IST)

4 State of the art

There are already some approaches to transition from machine-oriented to human-oriented languages and in particular from markup languages to modeling languages — an overview is given in [39]. To summarize, there are several approaches that either apply forward engineering from human-oriented languages to machine-oriented languages [6,10,20,33–35,41] or reverse engineering from the former space to the latter space [1, 3, 8, 11, 31, 38, 39]. Note that human-oriented languages in the most of these approaches are represented by Unified Modeling Language (UML) [36] models and the machine-oriented languages either by XML Schemas or DTDs.

In a reverse engineering based approach of bridging the model technological space and the XML technical space, Wimmer et al. [39] introduces a semi-automatic approach for generating Meta Object Facility (MOF)-based metamodels from Document Type Definitions (DTDs) with the purpose of enhancing language understanding, enable model transformations or language extension and tool interoperability. In this approach, first, generic transformation rules for transforming DTD concepts into metamodel concepts are defined together with manual user input for validating them to achieve a higher-quality metamodel. Secondly, a user driven action targeting the semantic enrichment of the metamodel is taken. The approach is depicted in form of a case study in which the WebML web modeling language [7] is used as an example. The proposed thesis differs to this approach in several ways. For example, they use DTDs instead of XSDs and MOF instead of a DSMLs focused on human-comprehensibility including concrete syntax. Furthermore, the approach foresees user interaction during or after the model transformation process.

Eysenholdt et al. [14] presents a report on the migration of a large modeling environment from XML/UML to Xtext/GMF. In their current modeling environment they identified that XML is inefficient due to verbose syntax and lack of good tool support and that the loading of UML modules and models is very inefficient. Therefore, they performed a modernization of their modeling environment by starting from XSDs from which Ecore metamodels are created and then manually modified before creating concrete graphical and textual syntaxes on top of it. Moreover, in their new modeling environment, they manually customized several Xtext features, such as, syntax highlighting, scoping, content assist and validation of cross references. When compared to the approach proposed by this dissertation, the intention is to perform required metamodel adaptions in the most automated fashion as possible. Therefore, changes made to the XSD file can be taken into account by re-executing the automated approach and hence avoiding repeated manual adaptions. Additionally, Xtext features as those mentioned above, are intended to be implemented in a generic technique such that they can be applied to any XML-based markup language.

Moreover, there are two different kinds of approaches that aim to define the concrete syntax of DSMLs. On one hand, grammar-based approaches [24, 28,

47] generate metamodels out of existing grammar definitions, also refered as grammarware. On the other hand, metamodel-based approaches [24] generate grammar out of existing metamodels, also refered as modelware.

In a grammar-based approach, J. Cánovas et al. [22] present the Gra2Mol transformation language able to perform grammar-to-model transformations appearing in model-driven evolution scenarios. Their transformation language is similar to the ATLAS Transformation Language (ATL) [23], however with the difference that the source element of a transformation rule is a grammar element instead of a source metamodel element as in ATL. In another grammar-based approach, F. Jouault et al. [24] defines an extension, called Textual Concrete Syntax (TCS), to the ATLAS Model Management Architecture (AMMA) [5] framework for the purpose of specifying textual concrete syntaxes by providing means to associate syntactic elements, e.g., language specific keywords, to metamodel elements. As opposed to the proposed dissertation the user of these approaches has to define its own transformation rules between either individual grammar elements or syntactic elements and metamodel elements instead of the transformation to be automatically inferred from the XML-based markup language using HOTs.

Bernstein et al. [4] introduces a tool that translates schemas from a source metamodel to a target metamodel. The prototype generates relational schemas from an object-oriented design and produces an executable instance-level mapping from the source model to the target model. Microsoft Visual Studio 2005 has been used as the integrated environment for the prototype in order to add support for object-to-relational mapping scenarios common in business applications. In this approach the target schema is updated incrementally instead of being re-created upon modifications. In the proposed approach, not only a specific XML Schema instance is evolved, but the XML Schema definition itself, such that all XML Schema based languages can be transformed.

According to the best knowledge of the author of this proposal, no existing work exists that marries concepts from the MDLE and MDSM methodologies considering the full spectrum of language engineering including abstract syntax, concrete syntax, and pragmatics. Hence, the fusion into Model-Driven Language Modernization (MDLM) opens up new possibilities to leverage individual advantages of machine-oriented and human-oriented languages, such as machine-readability and human-comprehensibility, respectively.

5 Summary

While the software engineering community has developed a large set of software languages, many of them are considered as machine-oriented languages. However, machine-oriented languages lack comprehensibility and present obstacles in development, re-use, and maintenance. A dissertation was proposed that aims at addressing challenges of modernizing software languages in terms

of evolving machine-oriented languages to human-oriented languages. Although this is considered a non-trivial undertaking it stimulates possibilities to advance machine-oriented language concepts with human-oriented representations. For example, through the facilitation of MDE techniques, such as model transformations, DSMLs focusing on comprehensibility, re-use, and maintenance can be tailored to fit purposes of human-orientation. Furthermore, to keep benefits of both individual technical spaces intact, the transformation automation of source language instances to target language instance and vice-versa, is studied. Increasing the comprehensibility of machine-oriented languages is crucial for evolving them towards better human conformity. Having the latter goal in mind, during the course of this dissertation, several techniques and artifacts aiming to move closer to human-conformity will be produced and evaluated by applying them in terms of case studies and user studies.

References

1. Migrating from XML DTD to XML Schema using UML., 2000. http://www.rational.com/media/whitepapers/TP189draft.pdf.
2. Greg J Badros. JavaML: A Markup Language for Java Source Code. *Computer Networks*, 33(1):159–177, 2000.
3. Martin Bernauer, Gerti Kappel, and Gerhard Kramler. Representing XML schema in UML - an UML profile for XML schema. Technical report, Vienna University of Technology, 2003.
4. Philip A Bernstein, Sergey Melnik, and Peter Mork. Interactive schema translation with instance-level mappings. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 1283–1286. VLDB Endowment, 2005.
5. Jean Bézivin, Frédéric Jouault, and David Touzet. An introduction to the ATLAS Model Management Architecture. *Rapport de recherche*, (05.01), 2005.
6. Linda Bird, Andrew Goodchild, and Terry Halpin. Object Role Modelling and XML-Schema. In *Conceptual Modeling—ER 2000*, pages 309–322. Springer, 2000.
7. Aldo Bongio, Piero Fraternali, Marco Brambilla, Sara Comai, and Maristella Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 2003.
8. Grady Booch, Magnus Christerson, Matthew Fuchs, and Jari Koistinen. UML for XML schema mapping specification. *Rational White Paper*, 1999.
9. Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML), 1998.
10. Rainer Conrad, Dieter Scheffner, and J Christoph Freytag. XML Conceptual Modeling using UML. In *Conceptual Modeling—ER 2000*, pages 558–571. Springer, 2000.
11. Ronaldo dos Santos Mello and Carlos Alberto Heuser. A Rule-Based Conversion of a DTD to a Conceptual Schema. In *Conceptual Modeling—ER 2001*, pages 133–148. Springer, 2001.
12. Sven Efftinge and Markus Völter. oAW Xtext: A Framework for Textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.
13. Moritz Eysholdt and Heiko Behrens. Xtext: Implement your Language Faster than the Quick and Dirty Way. In *Companion Proceedings of the ACM International Conference Object Oriented Programming Systems Languages and Applications*, pages 307–309. ACM, 2010.

14. Moritz Eysholdt and Johannes Rupprecht. Migrating a Large Modeling Environment from XML/UML to Xtext/GMF. In *Companion Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 97–104. ACM, 2010.
15. Jean-Marie Favre. Foundations of model (driven) (reverse) engineering : Models - episode I: stories of the fidus papyrus and of the solarus. In *Language Engineering for Model-Driven Software Development, 29. February - 5. March 2004*, 2004.
16. Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.
17. Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.
18. Richard C Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Pearson Education, 2009.
19. Felienne Hermans, Martin Pinzger, and Arie Van Deursen. Domain-specific languages in practice: A user study on the success factors. In *Model Driven Engineering Languages and Systems*, pages 423–437. Springer, 2009.
20. Michael Hucka. SCHUCS: A UML-Based Approach for Describing Data Representations Intended for XML Encoding. *Sys. Biol. Workbench Develop. Group*, 2000.
21. Javier Luis Cánovas Izquierdo and Jesús García Molina. A domain specific language for extracting models in software modernization. In *Model Driven Architecture - Foundations and Applications*, pages 82–97, 2009.
22. Javier Luis Cánovas Izquierdo and Jesús García Molina. Extracting models from source code in software modernization. *Software and System Modeling*, 13(2):713–734, 2014.
23. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.
24. Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 249–254. ACM, 2006.
25. Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic Literature Reviews in Software Engineering - A Systematic Literature Review. *Information and Software Technology*, 51(1):7–15, January 2009.
26. Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages using Metamodels*. Pearson Education, 2008.
27. Thomas Kühne. What is a model? In *Language Engineering for Model-Driven Software Development, 29. February - 5. March 2004*, 2004.
28. Andreas Kunert. Semi-Automatic Generation of Metamodels and Models from Grammars and Programs. *Electronic Notes in Theoretical Computer Science*, 211:111–119, 2008.
29. Ivan Kurtev, Mehmet Aksit, and Jean Bézivin. Technical Spaces: An Initial Appraisal. In *Proc. of CoopIS*, 2002.
30. Janne Luoma, Steven Kelly, and Juha-Pekka Tolvanen. Defining Domain-Specific Modeling Languages: Collected Experiences. In *Proceedings of the 4th Workshop on Domain-Specific Modeling*, 2004.
31. Murali Mani, Dongwon Lee, and Richard R Muntz. Semantic Data Modeling using XML Schemas. In *Conceptual Modeling—ER 2001*, pages 149–163. Springer, 2001.
32. Daniel L Moody. The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.

33. Will Provost. UML for W3C XML schema design. In *XML. com*, volume 7, 2002.
34. Nicholas Routledge, Linda Bird, and Andrew Goodchild. UML and XML schema. In *Australian Computer Science Communications*, volume 24, pages 157–166. Australian Computer Society, Inc., 2002.
35. Nicholas Routledge, Andrew Goodchild, and Linda Bird. *XML Schema Profile Definition*. PhD thesis, 2002.
36. James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.
37. Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.
38. Flora Dilys Salim, Rosanne Price, Shonali Krishnaswamy, and Maria Indrawan. UML documentation support for XML schema. In *Australian Conference on Software Engineering*, pages 211–220. IEEE, 2004.
39. Andrea Schauerhuber, Manuel Wimmer, Elisabeth Kapsammer, Wieland Schwinger, and Werner Retschitzegger. Bridging WebML to model-driven engineering: from document type definitions to meta object facility. *IET Software*, 1(3):81–97, 2007.
40. Douglas C Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31, 2006.
41. David Skogan. UML as a schema language for XML based data interchanging. In *Proceedings of the 2nd International Conference on The Unified Modeling Language (UML'99)*, 1999.
42. Thompson, Beech, Maloney, Mendelsohn. XML Schema Part 1: Structures Second Edition, 2004.
43. Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In *Model Driven Architecture-Foundations and Applications*, pages 18–33. Springer, 2009.
44. Bernhard Volz and Stefan Jablonski. Towards an open meta modeling environment. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*. ACM, 2010.
45. R Hevner von Alan, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.
46. William M Waite and Gerhard Goos. *Compiler Construction*. Springer, 1984.
47. Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware. In *Satellite Events at the MoDELS 2005 Conference*, pages 159–168. Springer, 2006.
48. Andreas Winter, Jean-Marie Favre, Dragan Gašević, and Ralf Lämmel. Guest editors' introduction to the special section on software language engineering. *IEEE Transactions on Software Engineering*, 35(6):0737–741, 2009.

Software Language Modernization

Patrick Neubauer

Dissertation Proposal
Business Informatics Group, Vienna University of Technology, Austria
neubauer@big.tuwien.ac.at

Abstract. The software engineering community has developed a plethora of software languages. On the one hand, instances of machine-oriented languages are considered to be fully machine readable but often less comprehensible for humans. On the other hand, human-oriented languages have been explicitly built with comprehensibility in mind but lack often the advantages of machine-oriented ones. Many of today's software languages fit into the category of machine-oriented languages, lacking comprehensibility and consequently presenting difficulties in development, re-use, and maintenance. Tackling these limitations by evolving machine-oriented languages into human-oriented languages represents a non-trivial undertaking but opens up possibilities of advancing former language concepts with latter representations. Therefore, the aim of this dissertation is to evolve machine-oriented languages into human-oriented languages by facilitating Model-Driven Engineering (MDE) techniques, such as model transformations, that allow to map between several concrete appearances of a language based on an abstract language definition. In order to evaluate the impact of modernizing software languages, a user study on effects of the proposed modernization process will be performed as well as a concrete metric suite for evaluating languages and their evolutions will be investigated.

1 Problem Description

Software Language Engineering (SLE) [26, 48] constitutes an emerging subfield of Software Engineering focusing mainly on language definition, parser building, and compiler construction [46]. Traditionally, there was a strong focus in language design to provide machine-readable and executable formats. In the last decades, there is a shift to pay more attention also to concrete syntax, i.e., notation, of a language due to several reasons, for example:

- more and more programs need to be maintained by humans;
- powerful tools [12,18] have emerged to define textual concrete syntax, graphical concrete syntax, and even a mixture of both without starting from scratch when defining the languages, cf. language workbench [16], metamodeling environments [44];
- the rise of languages tailored to the need of non-programming domain experts.

However, while the distinction between the notion of software languages, from which some of them are considered machine-oriented and others human-oriented, is known [15], it is not clearly described. It is suggested that, for a software language to become successful, both sides of a coin — from which one represents the machine side and the other the human side — have to be designed very carefully [48]. To be more precise, the machine's side is represented by abstract syntax and the human's side is represented by a concrete syntax emphasizing on perception, cognition, and usability in terms of the human brain as a processing unit of symbols [32].

With the introduction of the fully machine-processible Extensible Markup Language (XML) [9] in 1998, the World Wide Web Consortium (W3C) established a tremendous leap towards easing the design of software languages, leveraging the idea of having a generic editor, parser, and validation methodology. In 2004, W3C recommended XML Schema Definition (XSD) [42] as a new standard to formally describe elements in an XML document and hence also provide the capability for verifying the conformity of an XML document to an XSD specification. During the last decade a vast amount of markup languages have been implemented using XML Schemas. Rich Site Summary (RSS), Simple Object Access protocol (SOAP), Scalable Vector Graphics (SVG), Extensible Hypertext Markup Language (XHTML), and Business Process Execution Language (BPEL) represent prominent examples of markup languages.

With the rise of MDE [40], in which models are treated as first class citizens, several new methodologies emerged. For example, using Model-Driven Language Engineering (MDLE) [27], i.e., the application of MDE techniques to build software languages, Domain-Specific Modeling Languages (DSMLs) [17, 30] can be tailored to accurately match domain syntax and semantics while general-purpose notations seldom express design intent or domain concepts [40]. DSMLs can be equipped with elements that directly relate to familiar domains flattening the learning curves and therefore also increase the scope of subjects being able to work with such representations. Xtext [12] represents an exemplary MDLE tool to build, not only textual DSMLs from metamodels, but also advanced editor support by, for example, introducing syntax coloring, content assist, validation, and quick fix [13].

XML-based markup languages and DSMLs are residing in two different technological spaces [29] following distinctive goals. DSMLs are known for their success factors in terms of maintainability and re-use [19]. While DSMLs offer substantial gains in comprehensibility and maintainability when compared with markup languages, XML-based languages are fully machine readable and very flexible in contrast to DSMLs which should shield the user from mistakes. Furthermore, XML-based languages, also referred to as machine-oriented models [15], are specified in a generic concrete textual syntax, i.e, an angle bracket-based syntax, that can not be adapted to different user or domain requirements and appears verbose and complex in terms of human comprehension [2].

Moreover, Model-Driven Software Modernization (MDSM) [21], represents another emerging MDE discipline, that is loaded with approaches that focus on

Towards Model-Driven Software Language Modernization*

Patrick Neubauer

Business Informatics Group,
TU Wien, Austria
neubauer@big.tuwien.ac.at
http://www.big.tuwien.ac.at

Abstract. The introduction of Extensible Markup Language (XML) represented a tremendous leap towards the design of Domain-Specific Languages (DSLs). Although XML-based languages are well adopted and flexible, their generic editors miss modern DSL editor functionality. Additionally, artifacts defined with such languages lack comprehensibility and, therefore, maintainability, because XML has primarily been designed as a machine-processable format with immutable concrete syntax. While there exist techniques to migrate XML-based languages to modeling languages, they are composed of manual steps demanding complex language engineering skills that are usually not part of a domain engineer's skill set. To tackle these shortcomings, we propose a bridge between XML-based languages and text-based modeling languages. This includes the automated and customizable generation of Xtext-based editors from XML schema definitions (XSDs) providing advanced editor functionality, individualized textual concrete syntax style, and round-trip transformations enabling the exchange of data between the two languages. For the evaluation of the approach, we plan to conduct case studies as well as user studies based on industrial-strength markup languages that will be transformed to textual modeling languages including editors that are intended to be at least as powerful as those manually built for XML-based languages.

Keywords: Domain Specific Language, Model-Driven Engineering, Language Engineering, Markup Language, Language Modernization

1 Problem

With the introduction of machine-processable XML [11] in 1998, the World Wide Web Consortium (W3C) accomplished a tremendous leap towards easing the design of software languages, leveraging the idea of having a generic editor, parser, and validation methodology. Although for prominent XML-based languages — which are themselves DSLs —, such as Business Process Model and Notation (BPMN), advanced editors have been handcrafted, for others dedicated editor

* This research work is supervised by Manuel Wimmer and categorized as *end of initial stage and beginning of maturity stage*. More details are depicted in Section 7.

support is missing. XML has been primarily designed as machine-processible with immutable concrete syntax. More specifically, users of XML-based languages are bound to tree-based syntax that is described as verbose and complex in terms of human comprehension [3]. One of the main consequences of such syntax is the limited capability to improve upon human-comprehension and therefore maintainability. Conquering this limitation requires breaking out of inflexible XML syntax by providing an approach to construct a fully-customizable concrete syntax, which is also referred to as visual syntax or visual notation [23].

Bridging Modelware and Grammarware [26] for markup languages and modeling languages will enable the reuse and ease the maintenance of a significant amount of data encoded in the XML legacy format. Moreover, bridging Grammarware and Modelware for markup languages and modeling languages will lower the entrance barrier for developers to capture models containing domain-specific information and their management in an automated manner through the use of Model-Driven Engineering (MDE) tools [16].

State-of-the-art Model-Driven Language Engineering (MDLE) [17] frameworks, such as Xtext [8], allow the development of Domain-Specific Modeling Languages (DSMLs) [32] and the customization of their textual syntax. However, manually re-creating existing XML-based languages is a complex, error-prone, and time consuming task requiring complex language-engineering skills [10, 20, 22]. Additionally, DSMLs that ought to replace markup languages leave open backward-compatibility with usually comprehensive XML applications.

Evolving from machine-oriented languages to human-oriented languages enables lowering barriers imposed by machine-oriented languages on characteristics such as human perception, cognition, and usability. Likewise, human-oriented languages tend to be less appropriate for executability and interoperability. This suggests the need for automated transformations of languages between different technical spaces [19] as well as the automated transformation of language instances, i.e., XML instances and DSML models. In other words, the bridge between Grammarware and Modelware for markup languages and modeling languages still needs to be established. Individual challenges tackled in the proposed dissertation include but are not limited to the following:

Modernization Barriers between Markup Languages and DSMLs. To pursue the modernization of markup languages, i.e., the evolution of markup languages to languages supporting state-of-the-art technologies, such as, advanced editing capabilities, we need to identify barriers of existing MDLE frameworks as well as develop solutions to automate the modernization procedure. In more detail, we need to find a technique that bridges XSDs with DSMLs in terms of a combined approach that allows to produce instances on both sides. Further, the technique is intended to be applicable to any XSD-based language.

Generic Modernization of Markup Languages. Furthermore, we need to formalize solutions for these barriers in a generic framework that is able to modernize software languages and, in particular, XSD-based languages intended to address individual limitations imposed by machine orientation and human orientation. Whereas this includes appropriate language transformations, it also

requires the transformation of instances defined in such languages. Moreover, state-of-the-art MDE frameworks and tools have to be extended to support individual solutions as well as extended editor features, such as syntax coloring, content assistance, validation, and quick fixes, intending to increase comprehensibility and therefore maintainability.

Language Comprehensibility Analysis. To establish a common ground to evaluate a software language's orientation towards machines and humans, high-level metrics that describe the objectives of both machine-oriented software languages and languages that are human-oriented have to be identified and formalized such as executability and time to understand, create, and manipulate language instances. Furthermore, such metrics have to be measurable in terms of case studies, user studies, and/or other experiments. Therefore, to conduct studies focusing on language comprehensibility and therefore maintainability of individual instances we need to identify and construct or reuse appropriate methods to analyze language comprehensibility metrics.

2 Related Work

With respect to our approach of modernizing markup languages, in particular XSD-based languages, with modeling languages, such as Xtext-based languages, there exists a set of related approaches which cover certain aspects regarding the transition between involved technical spaces: (*i*) bridges between XMLware and Modelware and (*ii*) bridges between Modelware and Grammarware. To the best of our knowledge, there exists only one approach [9] to bridge XMLware and Grammarware directly which focuses on XSD and Xtext. But of course, there are other efforts in different contexts for bridging XML schemas and BNF-like languages, e.g., in the context of grammar hunting [35].

XMLware and Modelware. There exist several approaches to either apply transformations from Modelware to XMLware [30, 6, 21, 29, 28] or transformations from XMLware to Modelware [5, 7, 31]. In most of these approaches, Modelware is represented by Unified Modeling Language (UML) [27] models (centered around UML class diagrams) and XMLware either by XSDs or Document Type Definitions (DTDs). We propose an approach that differs from these approaches in several ways. For example, while Wimmer et al. [30] employ DTDs to generate Meta Object Facility (MOF)-based metamodels, we use XSDs to generate Ecore-based metamodels. Furthermore, our approach does not stop after having created the abstract syntax of the language defined, for instance, with a metamodel but also produces a textual concrete syntax using Xtext.

Modelware and Grammarware. There are two different kinds of approaches that aim to bridge Grammarware and Modelware by switching between grammars and metamodels. On one hand, grammar-based approaches [34, 18, 1] generate metamodels out of existing grammar definitions. On the other hand, metamodel-based approaches generate grammars out of existing metamodels [13, 35, 24] or link metamodels with grammars [15]. Especially, EMFText [12] seems to be an interesting alternative to Xtext used in our approach, as there is also the possibility to automatically derive several configurable concrete textual syntaxes

for one metamodel. However, the syntax configuration in EMFText is limited to predefined options and cannot be extended. Furthermore, Cánovas et al. [14] present the Gra2Mol transformation language in which concrete syntax metamodel instances are transformed to abstract syntax metamodel instances.

XMLware and Grammarware. Eysenholdt et al. [9] present a report on the migration of a large modeling environment from XML/UML to Xtext/GMF. In their legacy modeling environment, they identified that XML is inefficient due to verbose syntax and lack of tool support, and that the loading of UML modules and models is very inefficient. Therefore, they performed a modernization of their modeling environment by starting from XML schemas from which Ecore metamodels are created and then manually modified before creating concrete syntaxes through hand-crafted customization. In contrast, the goal of our work is to perform necessary metamodel adaptions in an automated fashion. Therefore, we enable the automated modernization of any XSD-based language as well as avoiding repeated manual adaptations caused by changed XSD specifications.

Language Comprehensibility. Aranda et al. [2] present a framework to evaluate the comprehensibility of (graphical) model representations based on theoretical frameworks in cognitive science. They list several challenges imposed by empirical evaluation of comprehensibility. For example, information equivalence of two different representations cannot be guaranteed even if the underlying conceptual content of different human readers is equal. Qualitative data and a human's inherent ability to operationalize such information is described as notoriously difficult. However, it is possible to construct comprehensibility variables to capture affected comprehensibility—like time required to understand the representation—and affecting comprehensibility—such as previous expertise with the domain being modeled. Eventually, by employing such measures, we hypothesize that human-comprehensibility of concrete syntax can be measured.

3 Proposed Solution

The increasing success of models in software development and model transformations in MDE during software development activities, such as automated forward engineering, highlight reasons to benefit from the same infrastructure to automate other tasks. Hence, we propose a model-driven solution that automates bridging XMLware, Modelware, and Grammarware. Our goal is to provide a framework that automatically modernizes XSD-based languages to metamodel-based languages that are supported by rich language workbenches, flexible syntax, and model-based techniques such as code generation, transformation, and validation. We achieve this by (*i*) chaining together tools and transformations (into what is from now on referred to as *Default Transformation Chain*), (*ii*) introducing new transformations that overcome existing gaps between XMLware, Modelware, and Grammarware such as mixed content and wildcards, data types and restrictions, and identifiers and identifier references, as well as (*iii*) introducing a *Concrete Syntax Configuration Language* enabling the flexible definition of textual concrete syntaxes.

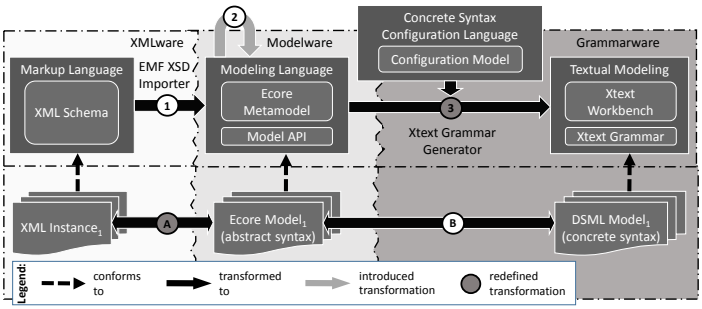


Fig. 1. Overview of the XMLText framework

Figure 1 depicts a conceptual overview of our XML to Xtext (XMLText) framework [25]. Like in the *Default Transformation Chain*, the first step is to transform a given *XML Schema* to an *Ecore Metamodel* by employing the *EMF XSD Importer* ①. To overcome limitations of this transformation, we complement it with novel transformations that adapt the generated *Ecore Metamodel* ②. For example, in order to tackle the issue of not supported feature maps occurring for mixed content and wildcards, we adapt the *Ecore Metamodel* by replacing feature maps with generic concrete constructs. Next, the adapted metamodel is used as input for generating the *Xtext Grammar* with the *Xtext Grammar Generator* provided by the Xtext framework ③. Also this step has to be extended with novel transformations that overcome limitations of the existing grammar generator, e.g., to enable storing actual values for attributes by importing, and referencing a library of data types. Moreover, we enable the automated customization of the target DSML's textual concrete syntax by providing a *Configuration Model*, i.e., a customizable template to specify the concrete syntax striving to enhance human-comprehensibility and therefore maintainability. For the adaptions of the *Ecore Metamodel* introduced by the XMLText framework, it is necessary to customize existing transformations (cf. Ⓐ in Figure 1) to act upon them on instance level. Therefore, we customize (*i*) the deserializer that reads *XML Instances* and creates in-memory *Ecore Model* representations conforming to the adapted *Ecore Metamodel* and (*ii*) the serializer that stores *Ecore Models* as *XML Instances*. As a result of keeping the *Xtext Grammar* coupled to the *Ecore Metamodel*, we are able to reuse the existing transformation Ⓑ between instances of the *Xtext Grammar* and the *Ecore Metamodel*.

With the introduction of transformation ② and the adaption of the transformations ③ and Ⓐ, our XMLText framework overcomes limitations of existing bridges between *XMLware* and *Grammarware* and thus allows an improved automated modernization of XML-based languages to metamodel-based and textual DSMLs. Listing 1.2 shows the result of applying the XMLText framework on an exemplary XML-based language instance, i.e., specifying an Apache web server cloud node, used in Listing 1.1.

```
1 <nodeTemplate id="ApacheWebServer" type="ApacheWebServerType" name="
      ↪Apache␣Web␣Server">
2   <properties id="ApacheWebServerProperties">
```

```
    <numCpus>1</numCpus>
    <memory>1024</memory>
  </properties>
</nodeTemplate>
```

Listing 1.1. Exemplary XML-based language instance

```
TNodeTemplate ApacheWebServer {
  name: "Apache Web Server"
  type: ApacheWebServerType
  Properties ApacheWebServerProperties {
      NumCpus: "1"
      Memory: "1024"
  }
}
```

Listing 1.2. Exemplary DSML model

4 Preliminary Work

Preliminary work in MDE and specifically in terms of interoperability between languages ultimately led to this research topic. In earlier work [4] we realized that many modeling languages targeting equal or similar purposes, e.g., modeling of cloud applications, are built from scratch causing extensive mismatches and difficulties in terms of interoperability among each other [33]. Hence, in [25] we established an initial framework that exploits existing seams between the technical spaces XMLware, Modelware, and Grammarware as well as closes several gaps between them. The resulting approach is able to generate Xtext-based editors from XSDs providing extended editor functionality, customization of textual concrete syntax, and round-trip transformations enabling the exchange of data between the involved technical spaces. The feasibility of the approach has been evaluated by a case study on TOSCA—an XML-based standard for defining Cloud deployments. The results show that the approach enables bridging XMLware with Modelware and Grammarware in several ways going beyond existing approaches by integrating useful and independent parts as well as improvements that allow the automated generation of editors that are at least equivalent to editors manually built for XML-based languages. However, while the results indicate that most of the known gaps have been bridged, some, e.g., XML namespaces, still need to be overcome. Further, XSD-based languages that employ a different subset of XML Schema language constructs may uncover previously unknown gaps that require further investigation.

5 Expected Contributions

The general idea to tackle the problem introduced in Section 1 is to address the challenges involved in modernizing software languages using MDE techniques. Hence, the following contributions are proposed:

Individual Language Modernization Techniques. This contribution intends to overcome individual barriers of existing MDLE frameworks when employing them in the formalization of techniques for the purpose of modernizing markup languages with DSMLs. For example, a technique has to be developed to transform XSD constructs that are currently not natively supported by Ecore,

such as data types, restrictions, and wildcards. Moreover, a language is developed that enables the configuration of the generated language in terms of configuration models. Furthermore, for tackling the challenge of transforming instances between XML-based languages and modernized DSML languages, we will develop model transformations that start from the meta language level and enable transformations on the instance level. Next, fitting techniques and extensions thereof are then used to implement the transformation of language instances starting from the meta language level. Finally, the soundness of individual modernization techniques will be validated using appropriate experiments.

Generic Framework and Initial Case Study. This contribution is designated to overcome challenges dictated by the generic modernization for markup languages through (*i*) the implementation of a generic framework facilitating existing MDE frameworks and tools, (*ii*) merging language modernization techniques, and (*iii*) extending a language's comprehensibility by incorporating the concrete syntax configuration language as well as providing advanced editor features, such as syntax coloring, content assisting, validation, and quick fixes, and their implementation in a generic and automated way. Moreover, to show that a markup language can be modernized into a machine-readable and human-comprehensible language, an initial case study, based on an XSD-based language, will be performed evaluating the feasibility of the developed generic framework.

Comprehensibility and Maintainability Analysis Framework. This contribution seeks to overcome the challenges imposed by evaluating language comprehensibility and therefore maintainability. First, a literature review on comprehensibility analysis, such as empirical comprehensibility [2], will be performed. This includes establishing a common ground to analyze a software language's orientation towards machines and humans and the formalization of high-level metrics that describe the objectives of both machine orientation and human orientation. Second, an evaluation mechanism for software languages in terms of formalized metrics is provided by a set of analysis methods. Next, both metrics and analysis methods thereof are implemented in an analysis framework that can be executed in the form of case studies, user studies, or other experiments, like machine analysis of software artifacts, or a combination of such.

6 Plan for Evaluation and Validation

The evaluation and validation of the proposed work are threefold. First, our solution to the problem caused by the current state of the art and its inability to transform instances between the technical spaces Grammarware and Modelware for markup language and modeling languages has been and will be evaluated by conducting case studies on the established bridge. In more detail, the initial case study evaluated the feasibility of our software language modernization framework XMLText in modernizing a markup language with a DSML, i.e., a modeling language, as well as validate the conformance of instances to their respective language. Language semantics are evaluated by performing round-trip transformations, i.e., comparing the source instance with the instance resulting from the round-trip transformation on the same source instance in terms of equality, as well as supplying instances to existing interpreters and compar-

ing the resulting behavior. Moreover, the study also included a comparison of the DSML produced by our framework as well as a hand-crafted DSML of the same language in terms of their completeness to their source XSD-based language. Furthermore, we plan to extend our initial case study by conducting a case study on a set of markup languages ensuring that all language concepts occurring in XSD-based markup languages are covered.

Second, we will evaluate the usability of our framework, in particular, the usability for domain engineers that usually do not have language engineering skills. In more detail, this study will involve professionals as well as students, a categorization of their language engineering skills, an evaluation of their experience with modernizing markup languages with modeling languages by using our framework, as well as the modernized language they produced.

Third, our solution to the problem of fixed concrete syntax of XML hindering human-comprehensibility and hence maintainability will be evaluated by conducting a user study. This study will involve engineers without advanced knowledge in both XMLware and Modelware and evaluate several aspects that are considered important in human comprehensibility metrics. For example, time spent to understand the representation and to create and manipulate instances by employing generic XML editors and their modernized counterparts, i.e., DSML model editors will be evaluated.

Relevant conferences for publishing the results include ECMFA, MODELS, SLE, SPLC, ASE, and PLDI. Moreover, relevant journals include TOPLAS, SoSyM, JSS, and IJPOP.

7 Current Status

Figure 2 shows the plan for conducting the presented research as well as its current status.

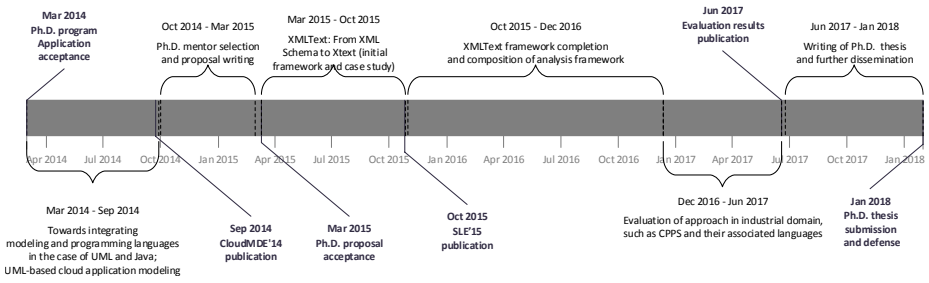


Fig. 2. Project phases and events time-line

We started by working on UML-based cloud application modeling [4]. The results have been published in September 2014. During the next phase a mentor has been selected as well as the proposal submitted and accepted. Next, we started working on our software language modernization framework XMLText [25] which resulted in an initial framework and case study based on the cloud topology and orchestration language TOSCA (cf. Section 4) published in

October 2015 at the International Conference on Software Language Engineering (SLE). Currently, we are working on completing the XMLText framework by bridging not yet resolved gaps, develop the concrete syntax configuration language, as well as establishing evaluation metrics and analysis methods required in the following phase. In more detail, the following phase will involve an evaluation of the overall approach in the industrial domain Cyber-Physical Production Systems (CPPS) and its associated languages in terms of a case study as well as a user study (cf. Section 6). The intention of selecting the domain of CPPS is caused by the fact that our faculty runs a laboratory[1] working in this domain and therefore provides us access to an extensive range of industrial partners and a real-world evaluation environment. The latter phase will be concluded with a publication of evaluation results in mid-2017, followed by the composition of a doctoral thesis as well as further dissemination of the established approach and framework.

References

1. Alanen, M., Porres, I.: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Tech. rep., Turku Centre for Computer Science (2003)
2. Aranda, J., Ernst, N., Horkoff, J., Easterbrook, S.: A framework for empirical evaluation of model comprehensibility. In: International Workshop on Modeling in Software Engineering (MISE). pp. 7–7 (2007)
3. Badros, G.J.: JavaML: A Markup Language for Java Source Code. Computer Networks 33(1), 159–177 (2000)
4. Bergmayr, A., Troya, J., Neubauer, P., Wimmer, M., Kappel, G.: UML-based cloud application modeling with libraries, profiles, and templates. In: Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE). pp. 56–65 (2014)
5. Bird, L., Goodchild, A., Halpin, T.: Object Role Modelling and XML-Schema. In: Proc. of ER, pp. 309–322. Springer (2000)
6. Booch, G., Christerson, M., Fuchs, M., Koistinen, J.: UML for XML schema mapping specification. Rational White Paper (1999)
7. Conrad, R., Scheffner, D., Freytag, J.C.: XML Conceptual Modeling using UML. In: Proc. of ER, pp. 558–571. Springer (2000)
8. Eysholdt, M., Behrens, H.: Xtext: Implement your Language Faster than the Quick and Dirty Way. In: Companion Proc. of OOPSLA. pp. 307–309. ACM (2010)
9. Eysholdt, M., et al.: Migrating a Large Modeling Environment from XML/UML to Xtext/GMF. In: Companion Proc. of OOPSLA. pp. 97–104. ACM (2010)
10. Fowler, M.: Domain-specific languages. Pearson Education (2010)
11. Harold, E.R., Means, W.S., Udemadu, K.: XML in a Nutshell, vol. 8. O'reilly Sebastopol, CA (2004)
12. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Model-Based Language Engineering with EMFText. In: GTTSE. pp. 322–345 (2011)
13. Heidenreich, F., Johannes, J., Karol, S., et al.: Derivation and Refinement of Textual Syntax for Models. In: Proc. of ECMDA-FA. pp. 114–129 (2009)
14. Izquierdo, J.L.C., Molina, J.G.: Extracting models from source code in software modernization. Software and System Modeling 13(2), 713–734 (2014)

[1] A list of currently ongoing Christian Doppler laboratories at the TU Wien can be found online at `http://goo.gl/HA95Rp`

15. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proc. of GPCE. pp. 249–254. ACM (2006)
16. Kolovos, D.S., Rose, L.M., Williams, J.R., Matragkas, N.D., Paige, R.F.: A Lightweight Approach for Managing XML Documents with MDE Languages. In: Proceedings of Modelling Foundations and Applications - 8th European Conference (ECMFA). pp. 118–132 (2012)
17. Kühne, T.: What is a model? In: Language Engineering for Model-Driven Software Development (2004)
18. Kunert, A.: Semi-Automatic Generation of Metamodels and Models from Grammars and Programs. Electronic Notes in Theoretical Computer Science 211, 111–119 (2008)
19. Kurtev, I., Aksit, M., Bézivin, J.: Technical Spaces: An Initial Appraisal. In: Proc. of CoopIS (2002)
20. Luoma, J., Kelly, S., Tolvanen, J.P.: Defining Domain-Specific Modeling Languages: Collected Experiences. In: Proc. DSM Workshop (2004)
21. Mani, M., Lee, D., Muntz, R.R.: Semantic Data Modeling using XML Schemas. In: Conceptual Modeling (ER), pp. 149–163. Springer (2001)
22. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM computing surveys (CSUR) 37(4), 316–344 (2005)
23. Moody, D.L.: The physics of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Transactions on Software Engineering 35(6), 756–779 (2009)
24. Muller, P.A., Hassenforder, M.: HUTN as a bridge between modelware and grammarware-an experience report. In: WISME Workshop (2005)
25. Neubauer, P., Bergmayr, A., Mayerhofer, T., Troya, J., Wimmer, M.: XMLText: From XML Schema to Xtext. In: Proceedings of the International Conference on Software Language Engineering. pp. 71–76. ACM, New York, NY, USA (2015)
26. Paige, R.F., Kolovos, D.S., Polack, F.A.C.: Metamodelling for grammarware researchers. In: Proc. of SLE. pp. 64–82 (2012)
27. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual. Pearson Higher Education (2004)
28. Salim, F.D., Price, R., Krishnaswamy, S., Indrawan, M.: UML documentation support for XML schema. In: Australian Conference on Software Engineering. pp. 211–220. IEEE (2004)
29. dos Santos Mello, R., Heuser, C.A.: A Rule-Based Conversion of a DTD to a Conceptual Schema. In: Proc. of ER, pp. 133–148. Springer (2001)
30. Schauerhuber, A., Wimmer, M., Kapsammer, E., Schwinger, W., Retschitzegger, W.: Bridging WebML to model-driven engineering: from document type definitions to meta object facility. IET Software 1(3), 81–97 (2007)
31. Skogan, D.: UML as a schema language for XML based data interchanging. In: Proc. of UML (1999)
32. Tolvanen, J., Kelly, S.: Defining domain-specific modeling languages to automate product derivation: Collected experiences. In: Proc. of SPLC. pp. 198–209 (2005)
33. Vallecillo, A., et al.: MDWEnet: A practical approach to achieving interoperability of model-driven web engineering methods. In: Proceedings of the 3rd International Workshop on Model-Driven Web Engineering (MDWE) (2007)
34. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Proc. of Satellite Events at MoDELS. pp. 159–168. Springer (2006)
35. Zaytsev, V.: Grammar Zoo: A corpus of experimental grammarware. Sci. Comput. Program. 98, 28–51 (2015)

Towards Model-Driven Software Language Modernization*

Patrick Neubauer

Business Informatics Group,
TU Wien, Austria
neubauer@big.tuwien.ac.at
http://www.big.tuwien.ac.at

Abstract. The introduction of Extensible Markup Language (XML) represented a tremendous leap towards the design of Domain-Specific Languages (DSLs). Although XML-based languages are well adopted and flexible, their generic editors miss modern DSL editor functionality. Additionally, artifacts defined with such languages lack comprehensibility and, therefore, maintainability, because XML has primarily been designed as a machine-processable format with immutable concrete syntax. While there exist techniques to migrate XML-based languages to modeling languages, they are composed of manual steps demanding complex language engineering skills that are usually not part of a domain engineer's skill set. To tackle these shortcomings, we propose a bridge between XML-based languages and text-based modeling languages. This includes the automated and customizable generation of Xtext-based editors from XML schema definitions (XSDs) providing advanced editor functionality, individualized textual concrete syntax style, and round-trip transformations enabling the exchange of data between the two languages. For the evaluation of the approach, we plan to conduct case studies as well as user studies based on industrial-strength markup languages that will be transformed to textual modeling languages including editors that are intended to be at least as powerful as those manually built for XML-based languages.

Keywords: Domain Specific Language, Model-Driven Engineering, Language Engineering, Markup Language, Language Modernization

1 Problem

With the introduction of machine-processable XML [11] in 1998, the World Wide Web Consortium (W3C) accomplished a tremendous leap towards easing the design of software languages, leveraging the idea of having a generic editor, parser, and validation methodology. Although for prominent XML-based languages — which are themselves DSLs —, such as Business Process Model and Notation (BPMN), advanced editors have been handcrafted, for others dedicated editor

* This research work is supervised by Manuel Wimmer and categorized as *end of initial stage and beginning of maturity stage*. More details are depicted in Section 7.

support is missing. XML has been primarily designed as machine-processible with immutable concrete syntax. More specifically, users of XML-based languages are bound to tree-based syntax that is described as verbose and complex in terms of human comprehension [3]. One of the main consequences of such syntax is the limited capability to improve upon human-comprehension and therefore maintainability. Conquering this limitation requires breaking out of inflexible XML syntax by providing an approach to construct a fully-customizable concrete syntax, which is also referred to as visual syntax or visual notation [23].

Bridging Modelware and Grammarware [26] for markup languages and modeling languages will enable the reuse and ease the maintenance of a significant amount of data encoded in the XML legacy format. Moreover, bridging Grammarware and Modelware for markup languages and modeling languages will lower the entrance barrier for developers to capture models containing domain-specific information and their management in an automated manner through the use of Model-Driven Engineering (MDE) tools [16].

State-of-the-art Model-Driven Language Engineering (MDLE) [17] frameworks, such as Xtext [8], allow the development of Domain-Specific Modeling Languages (DSMLs) [32] and the customization of their textual syntax. However, manually re-creating existing XML-based languages is a complex, error-prone, and time consuming task requiring complex language-engineering skills [10, 20, 22]. Additionally, DSMLs that ought to replace markup languages leave open backward-compatibility with usually comprehensive XML applications.

Evolving from machine-oriented languages to human-oriented languages enables lowering barriers imposed by machine-oriented languages on characteristics such as human perception, cognition, and usability. Likewise, human-oriented languages tend to be less appropriate for executability and interoperability. This suggests the need for automated transformations of languages between different technical spaces [19] as well as the automated transformation of language instances, i.e., XML instances and DSML models. In other words, the bridge between Grammarware and Modelware for markup languages and modeling languages still needs to be established. Individual challenges tackled in the proposed dissertation include but are not limited to the following:

Modernization Barriers between Markup Languages and DSMLs. To pursue the modernization of markup languages, i.e., the evolution of markup languages to languages supporting state-of-the-art technologies, such as, advanced editing capabilities, we need to identify barriers of existing MDLE frameworks as well as develop solutions to automate the modernization procedure. In more detail, we need to find a technique that bridges XSDs with DSMLs in terms of a combined approach that allows to produce instances on both sides. Further, the technique is intended to be applicable to any XSD-based language.

Generic Modernization of Markup Languages. Furthermore, we need to formalize solutions for these barriers in a generic framework that is able to modernize software languages and, in particular, XSD-based languages intended to address individual limitations imposed by machine orientation and human orientation. Whereas this includes appropriate language transformations, it also

requires the transformation of instances defined in such languages. Moreover, state-of-the-art MDE frameworks and tools have to be extended to support individual solutions as well as extended editor features, such as syntax coloring, content assistance, validation, and quick fixes, intending to increase comprehensibility and therefore maintainability.

Language Comprehensibility Analysis. To establish a common ground to evaluate a software language's orientation towards machines and humans, high-level metrics that describe the objectives of both machine-oriented software languages and languages that are human-oriented have to be identified and formalized such as executability and time to understand, create, and manipulate language instances. Furthermore, such metrics have to be measurable in terms of case studies, user studies, and/or other experiments. Therefore, to conduct studies focusing on language comprehensibility and therefore maintainability of individual instances we need to identify and construct or reuse appropriate methods to analyze language comprehensibility metrics.

2 Related Work

With respect to our approach of modernizing markup languages, in particular XSD-based languages, with modeling languages, such as Xtext-based languages, there exists a set of related approaches which cover certain aspects regarding the transition between involved technical spaces: (*i*) bridges between XMLware and Modelware and (*ii*) bridges between Modelware and Grammarware. To the best of our knowledge, there exists only one approach [9] to bridge XMLware and Grammarware directly which focuses on XSD and Xtext. But of course, there are other efforts in different contexts for bridging XML schemas and BNF-like languages, e.g., in the context of grammar hunting [35].

XMLware and Modelware. There exist several approaches to either apply transformations from Modelware to XMLware [30, 6, 21, 29, 28] or transformations from XMLware to Modelware [5, 7, 31]. In most of these approaches, Modelware is represented by Unified Modeling Language (UML) [27] models (centered around UML class diagrams) and XMLware either by XSDs or Document Type Definitions (DTDs). We propose an approach that differs from these approaches in several ways. For example, while Wimmer et al. [30] employ DTDs to generate Meta Object Facility (MOF)-based metamodels, we use XSDs to generate Ecore-based metamodels. Furthermore, our approach does not stop after having created the abstract syntax of the language defined, for instance, with a metamodel but also produces a textual concrete syntax using Xtext.

Modelware and Grammarware. There are two different kinds of approaches that aim to bridge Grammarware and Modelware by switching between grammars and metamodels. On one hand, grammar-based approaches [34, 18, 1] generate metamodels out of existing grammar definitions. On the other hand, metamodel-based approaches generate grammars out of existing metamodels [13, 35, 24] or link metamodels with grammars [15]. Especially, EMFText [12] seems to be an interesting alternative to Xtext used in our approach, as there is also the possibility to automatically derive several configurable concrete textual syntaxes

for one metamodel. However, the syntax configuration in EMFText is limited to predefined options and cannot be extended. Furthermore, Cánovas et al. [14] present the Gra2Mol transformation language in which concrete syntax metamodel instances are transformed to abstract syntax metamodel instances.

XMLware and Grammarware. Eysenholdt et al. [9] present a report on the migration of a large modeling environment from XML/UML to Xtext/GMF. In their legacy modeling environment, they identified that XML is inefficient due to verbose syntax and lack of tool support, and that the loading of UML modules and models is very inefficient. Therefore, they performed a modernization of their modeling environment by starting from XML schemas from which Ecore metamodels are created and then manually modified before creating concrete syntaxes through hand-crafted customization. In contrast, the goal of our work is to perform necessary metamodel adaptions in an automated fashion. Therefore, we enable the automated modernization of any XSD-based language as well as avoiding repeated manual adaptations caused by changed XSD specifications.

Language Comprehensibility. Aranda et al. [2] present a framework to evaluate the comprehensibility of (graphical) model representations based on theoretical frameworks in cognitive science. They list several challenges imposed by empirical evaluation of comprehensibility. For example, information equivalence of two different representations cannot be guaranteed even if the underlying conceptual content of different human readers is equal. Qualitative data and a human's inherent ability to operationalize such information is described as notoriously difficult. However, it is possible to construct comprehensibility variables to capture affected comprehensibility—like time required to understand the representation—and affecting comprehensibility—such as previous expertise with the domain being modeled. Eventually, by employing such measures, we hypothesize that human-comprehensibility of concrete syntax can be measured.

3 Proposed Solution

The increasing success of models in software development and model transformations in MDE during software development activities, such as automated forward engineering, highlight reasons to benefit from the same infrastructure to automate other tasks. Hence, we propose a model-driven solution that automates bridging XMLware, Modelware, and Grammarware. Our goal is to provide a framework that automatically modernizes XSD-based languages to metamodel-based languages that are supported by rich language workbenches, flexible syntax, and model-based techniques such as code generation, transformation, and validation. We achieve this by (*i*) chaining together tools and transformations (into what is from now on referred to as *Default Transformation Chain*), (*ii*) introducing new transformations that overcome existing gaps between XMLware, Modelware, and Grammarware such as mixed content and wildcards, data types and restrictions, and identifiers and identifier references, as well as (*iii*) introducing a *Concrete Syntax Configuration Language* enabling the flexible definition of textual concrete syntaxes.

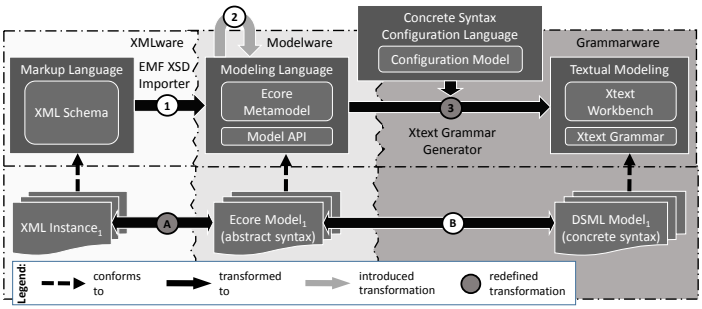


Fig. 1. Overview of the XMLText framework

Figure 1 depicts a conceptual overview of our XML to Xtext (XMLText) framework [25]. Like in the *Default Transformation Chain*, the first step is to transform a given *XML Schema* to an *Ecore Metamodel* by employing the *EMF XSD Importer* ①. To overcome limitations of this transformation, we complement it with novel transformations that adapt the generated *Ecore Metamodel* ②. For example, in order to tackle the issue of not supported feature maps occurring for mixed content and wildcards, we adapt the *Ecore Metamodel* by replacing feature maps with generic concrete constructs. Next, the adapted metamodel is used as input for generating the *Xtext Grammar* with the *Xtext Grammar Generator* provided by the Xtext framework ③. Also this step has to be extended with novel transformations that overcome limitations of the existing grammar generator, e.g., to enable storing actual values for attributes by importing, and referencing a library of data types. Moreover, we enable the automated customization of the target DSML's textual concrete syntax by providing a *Configuration Model*, i.e., a customizable template to specify the concrete syntax striving to enhance human-comprehensibility and therefore maintainability. For the adaptions of the *Ecore Metamodel* introduced by the XMLText framework, it is necessary to customize existing transformations (cf. Ⓐ in Figure 1) to act upon them on instance level. Therefore, we customize (*i*) the deserializer that reads *XML Instances* and creates in-memory *Ecore Model* representations conforming to the adapted *Ecore Metamodel* and (*ii*) the serializer that stores *Ecore Models* as *XML Instances*. As a result of keeping the *Xtext Grammar* coupled to the *Ecore Metamodel*, we are able to reuse the existing transformation Ⓑ between instances of the *Xtext Grammar* and the *Ecore Metamodel*.

With the introduction of transformation ② and the adaption of the transformations ③ and Ⓐ, our XMLText framework overcomes limitations of existing bridges between *XMLware* and *Grammarware* and thus allows an improved automated modernization of XML-based languages to metamodel-based and textual DSMLs. Listing 1.2 shows the result of applying the XMLText framework on an exemplary XML-based language instance, i.e., specifying an Apache web server cloud node, used in Listing 1.1.

```
1 <nodeTemplate id="ApacheWebServer" type="ApacheWebServerType" name="
      ↪Apache␣Web␣Server">
2   <properties id="ApacheWebServerProperties">
```

```
    <numCpus>1</numCpus>
    <memory>1024</memory>
  </properties>
</nodeTemplate>
```

Listing 1.1. Exemplary XML-based language instance

```
TNodeTemplate ApacheWebServer {
  name: "Apache Web Server"
  type: ApacheWebServerType
  Properties ApacheWebServerProperties {
      NumCpus: "1"
      Memory: "1024"
  }
}
```

Listing 1.2. Exemplary DSML model

4 Preliminary Work

Preliminary work in MDE and specifically in terms of interoperability between languages ultimately led to this research topic. In earlier work [4] we realized that many modeling languages targeting equal or similar purposes, e.g., modeling of cloud applications, are built from scratch causing extensive mismatches and difficulties in terms of interoperability among each other [33]. Hence, in [25] we established an initial framework that exploits existing seams between the technical spaces XMLware, Modelware, and Grammarware as well as closes several gaps between them. The resulting approach is able to generate Xtext-based editors from XSDs providing extended editor functionality, customization of textual concrete syntax, and round-trip transformations enabling the exchange of data between the involved technical spaces. The feasibility of the approach has been evaluated by a case study on TOSCA—an XML-based standard for defining Cloud deployments. The results show that the approach enables bridging XMLware with Modelware and Grammarware in several ways going beyond existing approaches by integrating useful and independent parts as well as improvements that allow the automated generation of editors that are at least equivalent to editors manually built for XML-based languages. However, while the results indicate that most of the known gaps have been bridged, some, e.g., XML namespaces, still need to be overcome. Further, XSD-based languages that employ a different subset of XML Schema language constructs may uncover previously unknown gaps that require further investigation.

5 Expected Contributions

The general idea to tackle the problem introduced in Section 1 is to address the challenges involved in modernizing software languages using MDE techniques. Hence, the following contributions are proposed:

Individual Language Modernization Techniques. This contribution intends to overcome individual barriers of existing MDLE frameworks when employing them in the formalization of techniques for the purpose of modernizing markup languages with DSMLs. For example, a technique has to be developed to transform XSD constructs that are currently not natively supported by Ecore,

such as data types, restrictions, and wildcards. Moreover, a language is developed that enables the configuration of the generated language in terms of configuration models. Furthermore, for tackling the challenge of transforming instances between XML-based languages and modernized DSML languages, we will develop model transformations that start from the meta language level and enable transformations on the instance level. Next, fitting techniques and extensions thereof are then used to implement the transformation of language instances starting from the meta language level. Finally, the soundness of individual modernization techniques will be validated using appropriate experiments.

Generic Framework and Initial Case Study. This contribution is designated to overcome challenges dictated by the generic modernization for markup languages through (*i*) the implementation of a generic framework facilitating existing MDE frameworks and tools, (*ii*) merging language modernization techniques, and (*iii*) extending a language's comprehensibility by incorporating the concrete syntax configuration language as well as providing advanced editor features, such as syntax coloring, content assisting, validation, and quick fixes, and their implementation in a generic and automated way. Moreover, to show that a markup language can be modernized into a machine-readable and human-comprehensible language, an initial case study, based on an XSD-based language, will be performed evaluating the feasibility of the developed generic framework.

Comprehensibility and Maintainability Analysis Framework. This contribution seeks to overcome the challenges imposed by evaluating language comprehensibility and therefore maintainability. First, a literature review on comprehensibility analysis, such as empirical comprehensibility [2], will be performed. This includes establishing a common ground to analyze a software language's orientation towards machines and humans and the formalization of high-level metrics that describe the objectives of both machine orientation and human orientation. Second, an evaluation mechanism for software languages in terms of formalized metrics is provided by a set of analysis methods. Next, both metrics and analysis methods thereof are implemented in an analysis framework that can be executed in the form of case studies, user studies, or other experiments, like machine analysis of software artifacts, or a combination of such.

6 Plan for Evaluation and Validation

The evaluation and validation of the proposed work are threefold. First, our solution to the problem caused by the current state of the art and its inability to transform instances between the technical spaces Grammarware and Modelware for markup language and modeling languages has been and will be evaluated by conducting case studies on the established bridge. In more detail, the initial case study evaluated the feasibility of our software language modernization framework XMLText in modernizing a markup language with a DSML, i.e., a modeling language, as well as validate the conformance of instances to their respective language. Language semantics are evaluated by performing round-trip transformations, i.e., comparing the source instance with the instance resulting from the round-trip transformation on the same source instance in terms of equality, as well as supplying instances to existing interpreters and compar-

ing the resulting behavior. Moreover, the study also included a comparison of the DSML produced by our framework as well as a hand-crafted DSML of the same language in terms of their completeness to their source XSD-based language. Furthermore, we plan to extend our initial case study by conducting a case study on a set of markup languages ensuring that all language concepts occurring in XSD-based markup languages are covered.

Second, we will evaluate the usability of our framework, in particular, the usability for domain engineers that usually do not have language engineering skills. In more detail, this study will involve professionals as well as students, a categorization of their language engineering skills, an evaluation of their experience with modernizing markup languages with modeling languages by using our framework, as well as the modernized language they produced.

Third, our solution to the problem of fixed concrete syntax of XML hindering human-comprehensibility and hence maintainability will be evaluated by conducting a user study. This study will involve engineers without advanced knowledge in both XMLware and Modelware and evaluate several aspects that are considered important in human comprehensibility metrics. For example, time spent to understand the representation and to create and manipulate instances by employing generic XML editors and their modernized counterparts, i.e., DSML model editors will be evaluated.

Relevant conferences for publishing the results include ECMFA, MODELS, SLE, SPLC, ASE, and PLDI. Moreover, relevant journals include TOPLAS, SoSyM, JSS, and IJPOP.

7 Current Status

Figure 2 shows the plan for conducting the presented research as well as its current status.

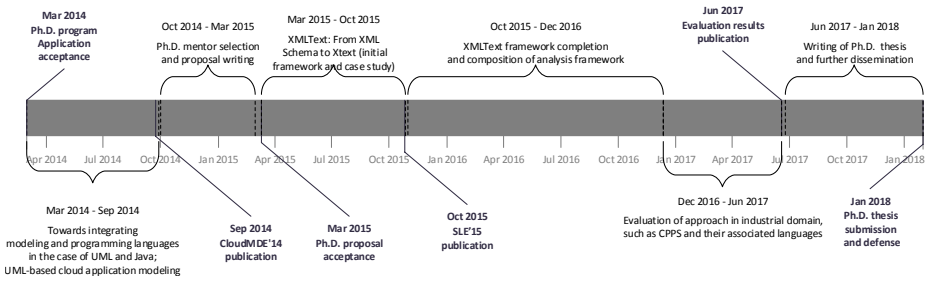


Fig. 2. Project phases and events time-line

We started by working on UML-based cloud application modeling [4]. The results have been published in September 2014. During the next phase a mentor has been selected as well as the proposal submitted and accepted. Next, we started working on our software language modernization framework XMLText [25] which resulted in an initial framework and case study based on the cloud topology and orchestration language TOSCA (cf. Section 4) published in

October 2015 at the International Conference on Software Language Engineering (SLE). Currently, we are working on completing the XMLText framework by bridging not yet resolved gaps, develop the concrete syntax configuration language, as well as establishing evaluation metrics and analysis methods required in the following phase. In more detail, the following phase will involve an evaluation of the overall approach in the industrial domain Cyber-Physical Production Systems (CPPS) and its associated languages in terms of a case study as well as a user study (cf. Section 6). The intention of selecting the domain of CPPS is caused by the fact that our faculty runs a laboratory[1] working in this domain and therefore provides us access to an extensive range of industrial partners and a real-world evaluation environment. The latter phase will be concluded with a publication of evaluation results in mid-2017, followed by the composition of a doctoral thesis as well as further dissemination of the established approach and framework.

References

1. Alanen, M., Porres, I.: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Tech. rep., Turku Centre for Computer Science (2003)
2. Aranda, J., Ernst, N., Horkoff, J., Easterbrook, S.: A framework for empirical evaluation of model comprehensibility. In: International Workshop on Modeling in Software Engineering (MISE). pp. 7–7 (2007)
3. Badros, G.J.: JavaML: A Markup Language for Java Source Code. Computer Networks 33(1), 159–177 (2000)
4. Bergmayr, A., Troya, J., Neubauer, P., Wimmer, M., Kappel, G.: UML-based cloud application modeling with libraries, profiles, and templates. In: Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE). pp. 56–65 (2014)
5. Bird, L., Goodchild, A., Halpin, T.: Object Role Modelling and XML-Schema. In: Proc. of ER, pp. 309–322. Springer (2000)
6. Booch, G., Christerson, M., Fuchs, M., Koistinen, J.: UML for XML schema mapping specification. Rational White Paper (1999)
7. Conrad, R., Scheffner, D., Freytag, J.C.: XML Conceptual Modeling using UML. In: Proc. of ER, pp. 558–571. Springer (2000)
8. Eysholdt, M., Behrens, H.: Xtext: Implement your Language Faster than the Quick and Dirty Way. In: Companion Proc. of OOPSLA. pp. 307–309. ACM (2010)
9. Eysholdt, M., et al.: Migrating a Large Modeling Environment from XML/UML to Xtext/GMF. In: Companion Proc. of OOPSLA. pp. 97–104. ACM (2010)
10. Fowler, M.: Domain-specific languages. Pearson Education (2010)
11. Harold, E.R., Means, W.S., Udemadu, K.: XML in a Nutshell, vol. 8. O'reilly Sebastopol, CA (2004)
12. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Model-Based Language Engineering with EMFText. In: GTTSE. pp. 322–345 (2011)
13. Heidenreich, F., Johannes, J., Karol, S., et al.: Derivation and Refinement of Textual Syntax for Models. In: Proc. of ECMDA-FA. pp. 114–129 (2009)
14. Izquierdo, J.L.C., Molina, J.G.: Extracting models from source code in software modernization. Software and System Modeling 13(2), 713–734 (2014)

[1] A list of currently ongoing Christian Doppler laboratories at the TU Wien can be found online at `http://goo.gl/HA95Rp`

15. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proc. of GPCE. pp. 249–254. ACM (2006)
16. Kolovos, D.S., Rose, L.M., Williams, J.R., Matragkas, N.D., Paige, R.F.: A Lightweight Approach for Managing XML Documents with MDE Languages. In: Proceedings of Modelling Foundations and Applications - 8th European Conference (ECMFA). pp. 118–132 (2012)
17. Kühne, T.: What is a model? In: Language Engineering for Model-Driven Software Development (2004)
18. Kunert, A.: Semi-Automatic Generation of Metamodels and Models from Grammars and Programs. Electronic Notes in Theoretical Computer Science 211, 111–119 (2008)
19. Kurtev, I., Aksit, M., Bézivin, J.: Technical Spaces: An Initial Appraisal. In: Proc. of CoopIS (2002)
20. Luoma, J., Kelly, S., Tolvanen, J.P.: Defining Domain-Specific Modeling Languages: Collected Experiences. In: Proc. DSM Workshop (2004)
21. Mani, M., Lee, D., Muntz, R.R.: Semantic Data Modeling using XML Schemas. In: Conceptual Modeling (ER), pp. 149–163. Springer (2001)
22. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM computing surveys (CSUR) 37(4), 316–344 (2005)
23. Moody, D.L.: The physics of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Transactions on Software Engineering 35(6), 756–779 (2009)
24. Muller, P.A., Hassenforder, M.: HUTN as a bridge between modelware and grammarware-an experience report. In: WISME Workshop (2005)
25. Neubauer, P., Bergmayr, A., Mayerhofer, T., Troya, J., Wimmer, M.: XMLText: From XML Schema to Xtext. In: Proceedings of the International Conference on Software Language Engineering. pp. 71–76. ACM, New York, NY, USA (2015)
26. Paige, R.F., Kolovos, D.S., Polack, F.A.C.: Metamodelling for grammarware researchers. In: Proc. of SLE. pp. 64–82 (2012)
27. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual. Pearson Higher Education (2004)
28. Salim, F.D., Price, R., Krishnaswamy, S., Indrawan, M.: UML documentation support for XML schema. In: Australian Conference on Software Engineering. pp. 211–220. IEEE (2004)
29. dos Santos Mello, R., Heuser, C.A.: A Rule-Based Conversion of a DTD to a Conceptual Schema. In: Proc. of ER, pp. 133–148. Springer (2001)
30. Schauerhuber, A., Wimmer, M., Kapsammer, E., Schwinger, W., Retschitzegger, W.: Bridging WebML to model-driven engineering: from document type definitions to meta object facility. IET Software 1(3), 81–97 (2007)
31. Skogan, D.: UML as a schema language for XML based data interchanging. In: Proc. of UML (1999)
32. Tolvanen, J., Kelly, S.: Defining domain-specific modeling languages to automate product derivation: Collected experiences. In: Proc. of SPLC. pp. 198–209 (2005)
33. Vallecillo, A., et al.: MDWEnet: A practical approach to achieving interoperability of model-driven web engineering methods. In: Proceedings of the 3rd International Workshop on Model-Driven Web Engineering (MDWE) (2007)
34. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Proc. of Satellite Events at MoDELS. pp. 159–168. Springer (2006)
35. Zaytsev, V.: Grammar Zoo: A corpus of experimental grammarware. Sci. Comput. Program. 98, 28–51 (2015)

XMLText: From XML Schema to Xtext

Patrick Neubauer, Alexander Bergmayr, Tanja Mayerhofer,
Javier Troya, and Manuel Wimmer

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

Abstract

A multitude of Domain-Specific Languages (DSLs) have been implemented with XML Schemas. While such DSLs are well adopted and flexible, they miss modern DSL editor functionality. Moreover, since XML is primarily designed as a machine-processible format, artifacts defined with XML-based DSLs lack comprehensibility and, therefore, maintainability. In order to tackle these shortcomings, we propose a bridge between the XML Schema Definition (XSD) language and text-based metamodeling languages. This bridge exploits existing seams between the technical spaces XMLware, modelware, and grammarware as well as closes identified gaps. The resulting approach is able to generate Xtext-based editors from XSDs providing powerful editor functionality, customization options for the textual concrete syntax style, and round-trip transformations enabling the exchange of data between the involved technical spaces.

We evaluate our approach by a case study on TOSCA, which is an XML-based standard for defining Cloud deployments. The results show that our approach enables bridging XMLware with modelware and grammarware in several ways going beyond existing approaches and allows the automated generation of editors that are at least equivalent to editors manually built for XML-based languages.

Categories and Subject Descriptors D.2.6 [*Programming Environments*]: Programmer workbench; I.7.2 [*Document Preparation*]: Markup languages

Keywords DSL, Language Engineering, Markup Language, Language Modernization, XSD, Xtext

General Terms Algorithms, Languages

SLE'15, October 26–27, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3686-4/15/10...$15.00
http://dx.doi.org/10.1145/2814251.2814267

1. Introduction

XML has been primarily designed as a machine-processible format following the fixed angle-bracket syntax. While for prominent XML-based languages, such as OASIS's TOSCA [8], advanced editors have been handcrafted, for others, like Artificial Intelligence Markup Language (AIML) [19], no dedicated editor is available. In the latter case, language users are bound to the angle-bracket syntax that is verbose and complex in terms of human-comprehension and therefore impedes maintainability [3].

Tackling these major limitations requires breaking out of inflexible XML syntax by providing support to construct a fully-customizable concrete syntax and language workbench. While state-of-the-art Model-Driven Language Engineering (MDLE) frameworks, such as Xtext [9], allow the development of Domain-Specific Modeling Languages (DSMLs) as well as accompanying customized concrete syntax and rich language workbenches, manually recreating existing XML-based languages with such frameworks is a complex, error-prone, and time-consuming task requiring language-engineering skills. Additionally, modeling languages that ought to replace XML-based languages leave behind backward-compatibility issues with the usually comprehensive set of applications built for the XML-based language.

To overcome these issues, our approach—the Model-Driven Language Modernization (MDLM) approach which is instantiated through the XML to Xtext (XMLText) framework[1]—facilitates the modernization of XSD-based languages with modelware and grammarware [14] by (i) transforming existing XSD-based languages to metamodels, (ii) adapting those metamodels to facilitate the production of effective language grammars, (iii) generating both customized language grammars and workbenches from the adapted metamodels, and (iv) enabling round-trip transformations between the original XSD-based language and the modernized DSML by generic serializers and parsers.

[1] Access to XMLText is provided on the paper's website at `http://xmltext.big.tuwien.ac.at`.

By supporting round-trip transformations, our framework inherently merges benefits of both XMLware and grammarware, namely machine-processibility and re-use of extensive XMLware applications of the former and high-customizability, enabling to target human-comprehensibility and therefore maintainability, of the latter. XMLText is evaluated based on a reproduction study on the XML-based language TOSCA, in particular, the framework's ability to produce complete DSMLs from XML Schemas.

The remainder of this paper is organized as follows. Section 2 provides an overview of gaps between XMLware, modelware, and grammarware as exposed by our case study. Section 3 introduces the MDLM approach as well as the XMLText framework. Section 4 evaluates the findings based on a reproductive study concerning an industrial strength language. Finally, Section 5 discusses related work before Section 6 concludes with a perspective on future work.

2. Gaps between XMLware, Modelware, and Grammarware

MDLE frameworks like Xtext accelerate the development of DSMLs and DSML environments to a great extent. They cover all aspects of a textual language infrastructure, including the default generation of a lexer, parser, as well as an editor featuring rich editing capabilities, such as, syntax highlighting, error indication, and content assisting. At the same time, they provide language engineers with the power to completely customize the look-and-feel, i.e., the textual concrete syntax, of DSMLs and therewith to construct DSMLs tailored to optimize human-comprehensibility—a customization not possible in XML due to its fixed concrete syntax.

For the purpose of modernizing XSD-based languages by transforming them to metamodel-based DSLs, we seek for a fully automated approach that produces from a given XSD a language grammar that fulfills our needs for human-comprehensibility. As we describe in the next paragraphs, we build upon existing tools that are integrated in EMF, namely the *EMF XSD Importer* and the *Xtext Grammar Generator*. First, the *EMF XSD Importer* is employed to produce an *Ecore Metamodel* from an existing *XML Schema*. Secondly, the *Ecore Metamodel* is used as input for the *Xtext Grammar Generator*, which transforms it to a corresponding *Xtext Grammar* as well as to a corresponding *Xtext Workbench*. However, our investigations have shown that chaining these tools together (into what is from now on referred to as *Default Transformation Chain*) leaves many gaps between the technical spaces of XMLware, modelware, and grammware open.

In the next paragraphs we dig into specific gaps discovered through a case study on the TOSCA cloud topology and orchestration language. Version 1.0 of the TOSCA XSD[2] contains 791 lines of code, 99 complex types, 11 simple types, 54 global types, 10 wildcards, 2 abstract types, and 2 global elements. Table 1 depicts an overview of *XML Schema* concepts currently processed by the *Default Transformation Chain* focusing on language concepts used in typical TOSCA instances. The column "Supported" denotes whether a particular *XML Schema* concept can be transformed to a DSML grammar rule able to represent the original *XML Schema* concept. In the following we summarize the concepts which are currently unsupported by the *Default Transformation Chain*.

XML Schema concepts	Definition	Supported	Notes
Element	xs:element	✓	Grammar rule is created
Attribute	xs:attribute	✓	Feature in grammar rule is created
Containment	(through nesting)	✓	Grammar rule is created and rule call stated
Mixed content	mixed="true"	✗	Ecore feature map is neglected in grammar generation
Wildcard	xs:any, xs:anyAttribute	✗	Ecore feature map is neglected in grammar generation
Restriction	xs:restriction	✗	Different interpretations
Data type	type="xs:string", type="..."	✗	Placeholder terminal and a to-do comment replace data types
Identifier and identifier reference	type="xs:ID", type="xs:IDREF"	✗	Placeholder terminal replaces identifier value

Table 1. Overview of *XML Schema* language concepts and their support by the *Default Transformation Chain*

Gap 1: Mixed Content and Wildcards. *XML Schema* allows to define mixed complex type elements, i.e., allowing character data to appear within the body of the element. Furthermore, the use of `xs:any` (cf. line 4 in Listing 1), i.e., a wildcard element, allows to specify any type of markup content in XML instances (cf. line 3-4 in Listing 2). The *EMF XSD Importer* translates such types to metaclasses containing feature maps that represent ambiguous language concepts whose handling is delegated to the underlying parser and serializer implementations. However, since the *Xtext Grammar Generator* neglects the occurrence of such implicitly modeled language concepts, they become unavailable at grammar level as well as on instance level.

```
1 <xs:element name="Properties" minOccurs="0">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:any namespace="##other" processContents="
            ↪lax"/>
5     </xs:sequence>
6   </xs:complexType>
7 </xs:element>
```

Listing 1. TOSCA XML Schema (excerpt)

Gap 2: Data Types and Restrictions. The W3C Recommendation on *XML Schema* data types [6] describes a set of built-in data types for different kinds of data, such as numbers, dates, strings, identifiers, and references. The *EMF XSD Importer* successfully transforms *XML Schema* data types to custom data types mapped to Java types at metamodel level. However, the *Xtext Grammar Generator* transforms any metamodel data type to a placeholder terminal symbol replacing the actual data type. Therefore,

[2] `http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/schemas/TOSCA-v1.0.xsd`.

the *Xtext Grammar* created by the *Default Transformation Chain* does not allow to construct instances able to store values for variables of any kind of data type. Moreover, this limitation also impacts *XML Schema* restrictions. For example, in case of a restricted XSD attribute of type `xs:string`, even if we correct the type definition of the resulting grammar attribute to the `STRING` terminal rule provided by Xtext, the attribute created in the Xtext grammar is interpreted differently: a String in *XML Schema* and Ecore is interpreted by excluding its surrounding quotes and in Xtext it is interpreted by including its surrounding quotes.

Gap 3: Identifiers and Identifier References. The *EMF XSD Importer* transforms attributes of type `xs:ID` (cf. `id` attributes in Listing 2) to Ecore attributes of type `java.lang.String` having set the `ID` property to `true`. Attributes of type `xs:IDREF` are also transformed to Ecore attributes instead of references. Hence, even if the gap related to data types is closed, the *Xtext Grammar Generator* still handles `xs:IDREF` equally to attributes – capable of holding primitive values not referring to other elements.

```
<nodeTemplate id="ApacheWebServer" type="
    ↪ApacheWebServerType" name="Apache␣Web␣Server">
  <properties id="ApacheWebServerProperties">
    <numCpus>1</numCpus>
    <memory>1024</memory>
  </properties>
</nodeTemplate>
```

Listing 2. TOSCA Moodle XML instance (excerpt)

Gap 4: Customizing Concrete Syntax. XML has been primarily designed as a machine-processible format composed of immutable concrete syntax. Therefore, users of XML-based languages are bound to angle-bracket syntax that is described as verbose and complex in terms of human-comprehension and therefore impedes maintainability [3].

3. XMLText

Our approach proposes bridging *XMLware*, *Modelware*, and *Grammarware*. Therefore, our goal is to provide a framework that automatically modernizes XSD-based languages to metamodel-based languages providing flexible syntax, rich language workbenches, and model-based techniques such as transformation, validation, and code generation. We achieve this goal by improving the transformations of the *Default Transformation Chain* as well as introducing new transformations that overcome the issues discussed in Section 2. Figure 1 depicts a conceptual overview of our XML to Xtext (XMLText) framework. Details are discussed in the following subsections.

Like in the *Default Transformation Chain*, the first step is to transform a given *XML Schema* to an *Ecore Metamodel* by employing the *EMF XSD Importer* (1). In order to tackle the issue of feature maps causing the production of empty grammar rules, we adapt the *Ecore Metamodel* by replacing feature maps with generic concrete constructs (cf. (2) in Figure 1). Next, the adapted metamodel is used as

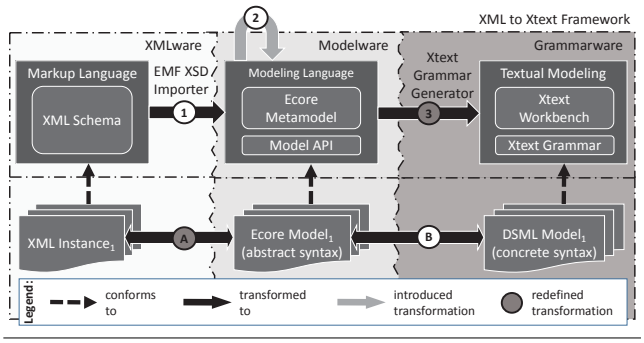


Figure 1. Overview of the XMLText framework

input for generating the *Xtext Grammar*. However, to store actual values for attributes we enhance the *Xtext Grammar Generator* (cf. (3) in Figure 1) by creating, importing, and referencing a library of data types. Moreover, we enable the automated customization of the textual concrete syntax of the target DSML by providing a configurable grammar rule template.

For the adaptions of the *Ecore Metamodel* introduced by the XMLText framework, it is necessary to customize existing transformations (cf. (A) in Figure 1) to act upon them on instance level. Therefore, we customize (*i*) the deserializer that reads *XML Instance*s and creates in-memory *Ecore Model* representations conforming to the adapted *Ecore Metamodel* and (*ii*) the serializer that stores *Ecore Model*s as *XML Instance*s. As a result of keeping the *Xtext Grammar* coupled to the *Ecore Metamodel*, we are able to reuse the existing transformation (B). With the introduction of transformation (2) and the adaption of the transformations (3) and (A), our XMLText framework overcomes limitations of existing bridges between *XMLware* and *Grammarware* and thus allows an improved automated modernization of XML-based languages to metamodel-based DSMLs. In the following, we detail these transformations. Listing 3 shows the result of applying the XMLText framework on the exemplary XML-based language instance used in Listing 2.

```
TNodeTemplate ApacheWebServer {
  name: "Apache␣Web␣Server"
  type: ApacheWebServerType
  Properties ApacheWebServerProperties {
      NumCpus: "1"
      Memory: "1024"
  }
}
```

Listing 3. TOSCA Moodle DSML model (excerpt)

3.1 Mixed Content and Wildcards

The definition of mixed content as well as wildcards causes the *EMF XSD Importer* to create an attribute of type `EFeatureMapEntry`. However, since the *Xtext Grammar Generator* neglects feature maps, such XSD concepts cannot be represented with the resulting *Xtext Grammar*. To successfully cope with the occurrence of feature maps, we replace them with generic concrete constructs for which grammar

rules are generated (cf. ② in Figure 1). As shown in Figure 2, `AnyGenericConstruct` is an abstract class extended by `AnyGenericElement` and `AnyGenericText`. Therefore, a wildcard XML tag is represented by `AnyGenericElement` and the text before or after an XML tag is represented by `AnyGenericText`, thus, allowing mixed content. While the former represents the notion of wildcards in terms of `xs:any`, the latter allows representing mixed content appearing either prior or after an XML tag. The successful application of this solution is depicted by lines 5-6 of Listing 3.

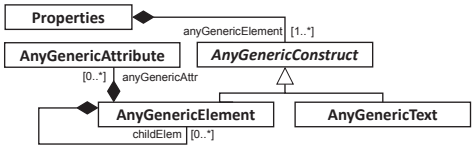


Figure 2. Explicit modeling structures replacing feature maps

3.2 Data Types

As mentioned earlier, the *Xtext Grammar Generator* does not create rules for metamodel data types. Therefore, both the specification of terminal rules as well as calls to these rules are missing. To overcome this limitation, we first constructed the *Xtext Data Type Library* defining terminal rules for built-in XSD data types and secondly adapted transformation ③ in Figure 1, such that these terminal rules are used in the final language grammar.

3.3 Identifiers and Identifier References

In order to tackle the gap associated with identifiers and identifier references, transformation ② replaces model attributes of type `xs:IDREF` with references to the metaclass `EObject`, which is the uppermost class in the Ecore hierarchy. While such a reference allows objects to reference any kind of object, an `xs:IDREF` attribute can only reference elements having an `xs:ID` attribute. Thus, transformation ③ introduces a necessary refinement. In particular, for an attribute of type `xs:IDREF`, we generate a grammar rule, allowing only to refer to objects with `xs:ID` attribute. Therefore, we define a new terminal rule (cf. example in Listing 4). Hereby, the subsequently generated editor provides support for referencing objects through content assist.

```
1 SourceElementType returns SourceElementType
2   referencingAttribute=[ecore::EObject|IDREF])?;
3
4 IDREF returns ecore::EString:
5   ID;
```

Listing 4. Xtext grammar for Identifiers and Identifier References

3.4 Customizing Concrete Syntax

To overcome the major limitation imposed by inflexible XML syntax, we provide an approach to construct a customizable concrete syntax enabling the specification of human-comprehensible and therefore increasingly maintainable instances. Usually, changing the concrete syntax of a DSML requires either to manually adapt the associated language grammar or the *Xtext Grammar Generator* transformations. The XMLText framework introduces a template mechanism that allows to specify customizable template files defining the concrete syntax of the target language. For example, as depicted by Listing 5, `VariableValueSpecification-TerminalSymbol` determines the terminal symbol used in the language grammar to specify a variable's value.

```
1 InterPackageReferenceTerminalSymbol = '.'
2 VariableValueSpecificationTerminalSymbol = ':'
3 PropertyMemberOpenTerminalSymbol = '{'
4 PropertyMemberCloseTerminalSymbol = '}'
```

Listing 5. Concrete syntax customization template file (excerpt)

4. Evaluation

In the evaluation we aimed at answering the following research question (*RQ*): Is the DSML generated by XMLText, i.e., $TOSCA_{XMLText}$, more complete as the DSML produced by the *Default Transformation Chain*, i.e., $TOSCA_{DTC}$ as well as available hand-crafted DSLs?

Evaluation Procedure. First we operate the *Default Transformation Chain* with the TOSCA XSD version 1.0 to generate $TOSCA_{DTC}$. Secondly, we employ the XMLText framework with the same TOSCA XSD to produce $TOSCA_{XMLText}$. Third, we gather language concepts and features appearing in the TOSCA XSD-conforming Moodle reference example [4], i.e., a complete definition of topology and orchestration details for an open source course management system, and correlate them with language concepts and features available in (i) $TOSCA_{DTC}$, (ii) *Cloudify DSL*—the only available textual DSL based on the TOSCA standard—, and (iii) $TOSCA_{XMLText}$.

Due to the fact that these languages have been implemented using different approaches, a common way of comparing them in terms of their completeness to a common language specification—the TOSCA standard [8] XSD version 1.0—is established. Therefore, our comparison takes into account language concepts and features occurring in the TOSCA XSD as well as in the individual language implementations. In particular, to analyze the language concepts of the *Cloudify DSL*, for which no language grammar is provided, we examine its language parser[3] on the existence of language concepts and features as defined in the TOSCA XSD. Furthermore, the unit of analysis in $TOSCA_{DTC}$ as well as $TOSCA_{XMLText}$ is represented by the DSML grammar.

Results. Table 2 depicts TOSCA language concepts and features employed in the Moodle example and their availability in $TOSCA_{DTC}$, *Cloudify DSL*, and $TOSCA_{XMLText}$. In total the Moodle example uses 19 different language con-

[3] The *Cloudify DSL* parser (version from April 1, 2015) examined during this evaluation can be retrieved online at `https://goo.gl/JzPL7U`.

cepts and 35 features defined in the TOSCA XSD. When looking for the availability of the combination of language concepts and features in the different languages we found that (i) *TOSCA$_{DTC}$* contains 17%, (ii) *Cloudify DSL* accommodates 37%, and (iii) *TOSCA$_{XMLText}$* encloses 98% of the TOSCA standard concepts and features found in the TOSCA XSD-conforming Moodle instance.

	Moodle Example	TOSCA$_{DTC}$	Cloudify DSL	TOSCA$_{XMLText}$
TOSCA Concepts	19	2 (~11%)	11 (~58%)	19 (100%)
TOSCA Features	35	7 (20%)	9 (~26%)	34 (~97%)
TOSCA Combined	54	9 (~17%)	20 (~37%)	53 (~98%)

Table 2. Availability of TOSCA standard concepts and features in different languages based on the Moodle example

In summary, we conclude that (i) the language grammar of *TOSCA$_{DTC}$* is missing essential concepts, such as, nodes and relationships, and is therefore not sufficient to represent the Moodle example. Furthermore, (ii) while the *Cloudify DSL* parser contains more language concepts and features as available in the *TOSCA$_{DTC}$*, it is still missing certain concepts, such as, requirements and capabilities. Moreover, for some missing concepts, such as `TDefinitions`, features are scattered throughout different language concepts in the *Cloudify DSL*. For example, their parser rule `models.Plan` contains policies and relationships that are originally located in `TDefinitions`. Therefore, the *Cloudify DSL* does not fully conform to the TOSCA standard and hence requires the user to map TOSCA XSD-conforming instances to *Cloudify DSL*-conforming instances. Finally, (iii) *TOSCA$_{XMLText}$* allows to represent almost entirely the same information as depicted in the Moodle example. In more detail, *TOSCA$_{XMLText}$* is missing the representation of the `xmlns` feature which is represented in the root element of the metamodel. Therefore, except for the occurrence of `xmlns`, the XMLText framework is able to perform round-trip transformations between TOSCA XSD-conforming XML instances and modernized TOSCA DSML-conforming models facilitating the re-use of existing XMLware applications as well as advanced capabilities of modern DSMLs.

Threats to Validity. We identified three threats of validity: (i) misinterpretation of language concepts and features due to their naming differences in the *Cloudify DSL* and the TOSCA standard, (ii) consideration of a subset of the TOSCA language represented by the TOSCA Moodle example, i.e., representing a subset of possible TOSCA language concepts and features, and (iii) the consideration of TOSCA as a representative for an XSD-based language, i.e., considering only a subset of all possible XML Schema language concepts and features. As a countermeasure to (i), we studied both the language concepts and features appearing in the *Cloudify DSL* language parser as well as in the *Cloudify DSL* language documentation. In order to act upon (ii), we identified that several TOSCA-based examples can act as a countermeasure. However, due to the lack of available TOSCA-based open source examples, we did not act upon it. Although, to encounter (iii) we selected TOSCA because it is a relatively complex language which poses several challenges when turning it into a modern textual DSML, we cannot claim any results outside of the TOSCA language.

5. Related Work

On a general level, we apply the ModelGen operator of Atzeni et al. [2]. This operator defines a general pattern which uses bridges on the meta-language level to derive transformations on the language level and instance level. This pattern also fits our architecture as presented in Figure 1. Traditionally, this pattern is proposed and used in the database field for schema-independent transformations, but it is of course also applicable in language engineering.

With respect to the complete transformation chain proposed in this paper, there exist a set of related approaches which cover certain aspects of this chain by focusing on the transitions between the involved technical spaces: (i) bridges between XMLware and modelware and (ii) bridges between modelware and grammarware. To the best of our knowledge, there exists only one approach [10] to bridge XMLware and grammarware directly which focus on XSD and Xtext. But of course, there are other efforts in different contexts for bridging XSD and BNF-like languages such as it is done in the context of grammar hunting [21].

XMLware and Modelware Several approaches for realizing either forward engineering from modelware to XMLware [5, 7] or reverse engineering from XMLware to modelware [16, 18] exist. In previous work [18], we presented an approach for generating MOF-based metamodels from DTDs. Our work presented in this paper differs from the previous approach in several ways, e.g., XSDs are used instead of DTDs and the full transition to textual modeling languages is done instead of stopping with the creation of the language's abstract syntax.

Modelware and Grammarware There exist grammar-driven approaches [1, 15, 20] in which metamodels are generated out of existing grammar definitions. In addition, metamodel-driven approaches generate grammars out of existing metamodels [11, 17] or link metamodels with grammars [13]. Especially, EMFText [12] seems to be an interesting alternative to Xtext used in this paper, as there is also the possibility to define several concrete textual syntaxes for one metamodel. As opposed to our work, the user of these approaches has to define its own transformation rules between either individual grammar rules or terminal rules and metamodel elements instead of relying on a generic and automated transformation of XSDs as proposed in this work.

XMLware and Grammarware Eysholdt and Rupprecht present a report [10] on the migration of a modeling environment from XML/UML to Xtext/GMF. Due to inefficiency of XML in terms of verbose syntax and lack of tool support

they perform the modernization of a legacy modeling environment. In detail, they start by creating Ecore metamodels from XSDs, then perform changes as well as customizations of Xtext features, and finally end up with a modernized language and workbench. Compared to our approach, they manually perform metamodel changes as well as customizations of Xtext features instead of building a generic and automated transformation chain.

6. Conclusion and Future Work

In this work we aimed at highlighting currently existing limitations in bridging XML-based languages with textual modeling languages and overcoming them by means of several improvements. This includes dealing with specific XSD concepts, namely mixed content, wildcards, restrictions, identifier and identifier references, and data types, as well as concrete syntax customization. The main principle that guided our solution was to represent each XSD concept explicitly in the modelware technical space. By this, important characteristics of XML such as being able to represent semi-structured data is now also better reflected in the corresponding textual modeling languages.

The proposed improvements have been bundled into the XMLText framework as well as evaluated regarding completeness. In particular, the evaluation has been carried out based on the OASIS TOSCA standard. The evaluation results indicate that the proposed XMLText framework significantly improves over existing solutions and generates a textual modeling language for TOSCA that is more complete than the currently available hand-crafted *Cloudify DSL*.

With respect to future work, first, to fully exploit the benefits of modernizing XSD-based languages with modeling languages, we strive to extend the current framework by addressing currently unresolved challenges as well as eventually arising challenges when conducting further case studies. In detail, we plan to conduct case studies based on different examples of the TOSCA language as well as other XSD-based languages covering different sets of XML Schema concepts. Secondly, we plan to quantify the actual impact of modernizing XSD-based languages by conducting user studies focusing on human-comprehension.

Acknowledgments

This work is funded by the European Commission under ICT Policy Support Programme, grant no. 317859 and by the Christian Doppler Forschungsgesellschaft and the BMWFW, Austria.

References

[1] M. Alanen and I. Porres. A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, Turku Centre for Computer Science, 2003.

[2] P. Atzeni, P. Cappellari, and P. A. Bernstein. ModelGen: Model Independent Schema Translation. In *Proc. of ICDE*, pages 1111–1112, 2005.

[3] G. J. Badros. JavaML: A Markup Language for Java Source Code. *Computer Networks*, 33(1):159–177, 2000.

[4] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner. *OpenTOSCA – a runtime for TOSCA-based cloud applications*. Springer, 2013.

[5] L. Bird, A. Goodchild, and T. Halpin. Object Role Modelling and XML-Schema. In *Proc. of ER*, pages 309–322. Springer, 2000.

[6] P. V. Biron and A. Malhotra. XML schema part 2: Datatypes second edition. W3C recommendation, W3C, Oct. 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[7] R. Conrad, D. Scheffner, and J. C. Freytag. XML Conceptual Modeling using UML. In *Proc. of ER*, pages 558–571. Springer, 2000.

[8] Derek Palma, Thomas Spatzier. Topology and Orchestration Specification for Cloud Applications Version 1.0, 2013.

[9] M. Eysholdt and H. Behrens. Xtext: Implement your Language Faster than the Quick and Dirty Way. In *Companion Proc. of OOPSLA*, pages 307–309. ACM, 2010.

[10] M. Eysholdt and J. Rupprecht. Migrating a Large Modeling Environment from XML/UML to Xtext/GMF. In *Companion Proc. of OOPSLA*, pages 97–104. ACM, 2010.

[11] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and Refinement of Textual Syntax for Models. In *Proc. of ECMDA-FA*, pages 114–129. Springer.

[12] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Model-Based Language Engineering with EMFText. In *Proc. of GTTSE*, pages 322–345. Springer, 2011.

[13] F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proc. of GPCE*, pages 249–254. ACM, 2006.

[14] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.

[15] A. Kunert. Semi-Automatic Generation of Metamodels and Models from Grammars and Programs. *Electronic Notes in Theoretical Computer Science*, 211:111–119, 2008.

[16] M. Mani, D. Lee, and R. R. Muntz. Semantic Data Modeling using XML Schemas. In *Proc. of ER*, pages 149–163. Springer, 2001.

[17] P.-A. Muller and M. Hassenforder. HUTN as a bridge between modelware and grammarware-an experience report. In *Proc. of WISME Workshop*, 2005.

[18] A. Schauerhuber, M. Wimmer, E. Kapsammer, W. Schwinger, and W. Retschitzegger. Bridging WebML to model-driven engineering: from document type definitions to meta object facility. *IET Software*, 1(3):81–97, 2007.

[19] R. S. Wallace. *The anatomy of ALICE*. Springer, 2009.

[20] M. Wimmer and G. Kramler. Bridging Grammarware and Modelware. In *Proc. of Satellite Events at MoDELS*, pages 159–168. Springer, 2006.

[21] V. Zaytsev. Grammar Zoo: A corpus of experimental grammarware. *Sci. Comput. Program.*, 98:28–51, 2015.

Automated Generation of Consistency-Achieving Model Editors

Patrick Neubauer, Robert Bill, Tanja Mayerhofer, and Manuel Wimmer
Business Informatics Group, TU Wien, Austria
{neubauer, bill, mayerhofer, wimmer}@big.tuwien.ac.at

***Abstract*—The advances of domain-specific modeling languages (DSMLs) and their editors created with modern language workbenches, have convinced domain experts of applying them as important and powerful means in their daily endeavors. Despite the fact that such editors are proficient in retaining syntactical model correctness, they present major shortages in mastering the preservation of consistency in models with elaborated language-specific constraints which require language engineers to manually implement sophisticated editing capabilities. Consequently, there is a demand for automating procedures to support editor users in both comprehending as well as resolving consistency violations. In this paper, we present an approach to automate the generation of advanced editing support for DSMLs offering automated validation, content-assist, and quick fix capabilities beyond those created by state-of-the-art language workbenches that help domain experts in retaining and achieving the consistency of models. For validation, we show potential error causes for violated constraints, instead of only the context in which constraints are violated. The state-space explosion problem is mitigated by our approach resolving constraint violations by increasing the neighborhood scope in a three-stage process, seeking constraint repair solutions presented as quick fixes to the editor user. We illustrate and provide an initial evaluation of our approach based on an Xtext-based DSML for modeling service clusters.**

***Index Terms*—Domain Specific Modeling Languages, Model Driven Engineering, Search-based Software Engineering, Advanced Editor Support**

I. Introduction

Domain-specific modeling languages (DSMLs) [1] encode the expertise of domain experts [2], e.g., hospital process managers and mechatronics engineers, and are, hence, languages designed for a specific class of problems allowing domain experts to describe their problem on a higher level of abstraction than possible with general-purpose modeling languages (GPMLs) [3]. DSMLs are known to leverage domain expertise and improve usability, comprehensibility, and maintainability compared to alternatives. On the one hand the benefits of DSMLs have been recognized, but on the other hand the development of advanced DSML implementations still requires extensive manual effort and the combined knowledge of domain and language engineers resulting in slow industrial adoption. Therefore, to advance the state-of-the-art in maintenance and evolution of languages and systems created with them, our goal is to significantly reduce the cost associated with the creation and thus the adoption of advanced DSML implementations required by currently available approaches [4].

Model Driven Language Engineering (MDLE) frameworks (a.k.a. language workbenches) such as Xtext [5], ease the out-of-the-box creation of powerful DSML implementations, including associated editors [4]. Typically, an MDLE framework allows creating a language either by developing a metamodel representing the language's abstract syntax, which is subsequently used to generate a DSML implementation including language grammar, or by developing a language grammar, which is automatically used to generate an associated metamodel. Either way, generated DSML implementations require extensive manual implementation for enabling advanced validation, content-assist, and quick fix capabilities. For example, in case a metamodel containing formally specified consistency constraints, such as restricting the `speed` attribute of a `Server` element to be represented by an Integer greater than `0`, is used to derive a DSML editor, it does not take into account such restrictions for displaying an appropriate validation result. In more detail, it highlights the entire `Server` element as erroneous and does not provide any quick fix to solve the violated constraint. Further, to take formal constraint specifications into account, the language developer has to manually implement validation, content-assist, and quick fixes using a general-purpose programming language, such as Java, on top of the generated DSML editor.

On one side, advanced editing capabilities decrease the effort to create and modify instances of a language (i.e., models) and therefore increase the productivity of language use. On the other side, they significantly increase the development effort for DSML editors.

Moreover, models created with DSMLs as well as the DSMLs themselves evolve [6], [7]. Each time a metamodel or its associated formal constraint specifications are adapted, handwritten implementations for model validation, content-assist, and quick fixes need to be accustomed as well.

In this work, we present an approach to automatically generate advanced validation, content-assist, and quick fix capabilities for DSMLs based on formal constraint specifications residing in language metamodels. Hence, the approach exploits language metamodels equipped with formal constraint specifications to achieve DSML implementations that include the desired capabilities, thus reducing initial and running costs for creating and maintaining editors which allow to efficiently create and maintain models.

Formal constraint specifications may be added to the language metamodels in terms of Object Constraint Language

(OCL) [8] constraints and Ecore annotations added to metamodel elements. In our approach, we extract OCL constraints and Ecore annotations from metamodels and analyze them to create advanced DSML editors. For doing so, we had to face several challenges: (i) narrow the scope of infinite possible ways to repair a given set of inconsistencies to a practically applicable level, e.g., as done by Nentwich et al. in [9], and (ii) measure the impact of an identified inconsistency repair solution as it may introduce new inconsistencies and therefore may be counterproductive [10].

We hypothesize that, by applying our approach, a DSML implementation is created, which enables to (i) accelerate the creation and editing of DSML models, (ii) build DSML models with fewer errors, and hence (iii) improve the overall productivity and quality of DSML creation by language engineers and DSML usage by editor users.

Our vision for the future is to completely automate the evolution of existing languages, e.g., the evolution of markup languages, such as XML Schema languages, to advanced DSMLs providing instance-level backward compatibility [11]. Hence, in this paper, we focus on the particular aspect of increasing the quality and usability of DSML editors by extracting formal specifications from the metamodel that is part of the DSML as well as its editor.

A realization and evaluation of our approach has been made available in form of the IntellEdit framework[1] as well as its application on a DSML for modeling service clusters.

In the next sections we describe (i) the background by providing an overview of appearing concepts, (ii) related work, (iii) a motivating example, (iv) the generic approach as well as our concrete implementation, (v) the evaluation of our implementation of a language for modeling service clusters, (vi) a detailed comparison of our approach with existing language workbenches, and (vii) the conclusion including a presentation of future work.

II. Background

This section describes key concepts behind our approach. Gray et al. define domain-specific modeling as being tightly coupled to a language that is by definition linked to the domain over which it is valid [2]. Consequently a domain-specific (modeling) language (DS(M)L) [12] is a language tailored to a specific application.

DSMLs are based on MDE [13] technologies that formalize application structure, behavior, and requirements within particular domains using metamodels defining relationships among concepts in a domain as well as key semantics and constraints associated with such concepts [13]. Therefore, the development of DSMLs, which is considered challenging, requires developers with domain expertise as well as language engineering expertise from which few of them have both [12]. Moreover, DSMLs are intended to be used by developers to build applications expressing design intent declaratively rather than imperatively and hence require significant customization before they can be applied in practice [14].

To support the development of DSMLs, MDLE frameworks, such as Xtext [5], emerged, enabling language designers to ease language development by leveraging advances in editor technology of mainstream IDEs. Such MDLE frameworks automate, for instance, the creation of language specific parsers, serializers, and editors providing basic syntax highlighting, content-assist, folding, jump-to-declaration, and reverse reference lookup across multiple files. However, their validation, content-assist, and quick fix capabilities are limited, not only due to the unavailable information but also due to the capacity in which the available information is exploited. For example, validators typically highlight an entire class as invalid instead of the actual feature violating a constraint, which makes identification and resolving of errors more difficult and time-consuming.

On one hand and as found in our earlier work [15], the generation of language grammar from metamodels through MDE techniques and MDLE frameworks still suffers several limitations, such as the ability to store values for data type instances, and hence require extensive manual customization and extension by language engineers. On the other hand, if a metamodel is derived from a language grammar, that metamodel is not supplied with formal constraints. In particular, we do not know of any approach which allows translating constraints defined in e.g. attribute grammars to metamodel constraints. However, formal constraints have become key components in MDE for expressing different kinds of (meta)model queries and specification and manipulation requirements. Moreover, in the technological space of Grammarware, it is not feasible to construct and maintain complex formal constraints.

As a textual formal constraint language, the standardized Object Constraint Language (OCL) [8] is widely used to enhance modeling languages with unambiguous and precise constraints. OCL constraints are captured as invariants using a typed, declarative and side-effect free specification language supporting first-order predicate logic offering navigation and (meta)model querying facilities.

However, while OCL represents the most used textual formal specification language in the area of MDE, several shortcomings have been identified [16]. These include (i) poor support for user feedback, (ii) no support for warnings, (iii) no support for dependent constraints, (iv) limited flexibility in context definition, and (v) no support for repairing invariants.

III. Related Work

Egyed et al. and Reder et al. [17], [18], [19] present an approach to assist editor users in fixing inconsistencies in UML models by generating a set of concrete changes as well as their impact on consistency rules. In [18] and [19] they focus on the cause of inconsistencies by analyzing the consistency rules as well as their behavior during the validation, and present validation as well as repair in form of linearly growing trees. They evaluated the scalability of their approach by applying it to UML models and OCL rules. They were able to compute

[1] A ready-to-use virtual machine image and Eclipse instance of the IntellEdit framework as well as evaluation results can be retrieved online from http://intelledit.big.tuwien.ac.at.

validation and repair trees in the millisecond-range, which indicates that the application of the tree data structure may be highly appropriate for the problem at hand. Our approach differs in the way it searches for complete repairs but is similar with respect to basic repair rules such as the insertion of a reference.

In [17] they found that the performance of applying the brute force technique to generate repair solutions is too low and hence decided to add manual specifications in form of value generation functions and (back)pointer specifications. Compared to our approach, they follow the same motivation for automating the generation of repairs for constraints that otherwise require significant manual effort. However, our approach avoids the manual burden on the language engineer in the manual specification of value generation functions and (back)pointer references by adopting Search-based Software Engineering (SBSE) [20] techniques instead. SBSE entails the application of optimization techniques, originating from metaheuristic computation and operations research. Moreover, we hypothesize that the language engineer may not be able to feasibly state all possible value generation functions and (back)pointer specifications and hence unintentionally limits the generation of potentially favorable quick fix solutions. Furthermore, their approach does not support (i) the generation of fixes of inconsistencies whose resolution does not require the introduction of new model classes or attributes and (ii) the generation of resolutions that involve changes in multiple locations at a time.

Hegedüs et al. [21] present an approach to automate the generation of quick fixes for DSMLs by taking a set of constraints and model manipulation policies as input. Their repair solutions are realized as a sequence of operations, which are computed by employing state space exploration techniques, targeted to decrease the number of inconsistencies. Compared to our approach, they also look for local and global quick fix solutions. However, they do not take content-assist as well as the overall validity of the model into account. Moreover, they employ graph patterns to capture inconsistency rules and graph transformation rules for repair operations instead of textual formal constraints and SBSE techniques.

Silva et al. [22] describe a method for the generation of repair plans for an inconsistent model by configuring the explored search space to antagonize the underlying characteristic of the problem, i.e., the infinite amount of possible repair solutions, at hand. Compared to our approach, their approach restricts the search space exploration by (i) limiting the generation of repair solutions in general, (ii) the applicability to every model at hand, and (iii) the requirement to manually create inconsistency detection rules. Therefore the quality of generated results will suffer and the burden on the language engineer, in terms of language implementation and maintenance, is increased due to the manual construction of inconsistency detection rules. In our approach, we asynchronously look for solutions during editor execution and thus can potentially find solutions to resolve inconsistencies with a higher quality compared to results found in a limited amount of time.

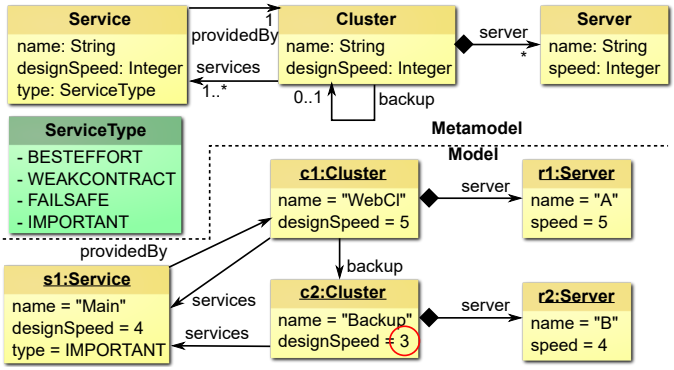


Fig. 1. Metamodel and model of the motivating example.

Ali et al. [23] and Semerth et al. [24] generate models which fulfill OCL constraints by optimization and reasoning, respectively. However, when compared to our approach, they only consider single-objective optimization, i.e., generated test models need to fulfill all OCL constraints which only considers the model state, and not how to get there. Therefore, they neglect the power of multi-objectiveness [25], which is required for establishing quick fix solutions for edit operations based on multiple conflicting objectives, e.g., considering both solution validity as well as solution length as optimization objectives.

IV. Motivating Example

This section introduces our motivating example, represented by a DSML for modeling service clusters, which is referred to in the following sections. We do so by describing its metamodel including formal constraints, which are employed by our approach for creating advanced editor support.

Metamodel. The DSML metamodel (cf. Fig. 1 upper part) is composed of the classes Cluster, Service, and Server. A Service is defined in terms of *designSpeed*, i.e., an Integer representing the speed for which a service is intended to be used, *type*, which may either be *BESTEFFORT*, *WEAKCONTRACT*, *FAILSAFE*, or *IMPORTANT*, and has to be provided by exactly one cluster. A Cluster has a *designSpeed*, i.e., an Integer representing the speed for which a cluster is intended to be used, as well as one or multiple containing servers. Moreover, a cluster is associated with one or more services and up to one backup cluster. Finally, each Server has a *speed* that it provides.

Formal Constraints. Formal constraints are defined for servers, clusters, and services (cf. Listing 1). For example, a Service is constrained by the type of service that it provides. In detail, it has to either have a *BESTEFFORT* service type with no further restrictions or, in case of a *WEAKCONTRACT* service type, its associated cluster has to be designed for a speed that is equal or greater than the designed speed of the service itself. A *FAILSAVE* service type means that some backup cluster has to be provided. The *IMPORTANT* service type further extends the restrictions associated with the *FAILSAFE* service type by requiring the designed speed

```
– Service
invariant speedFulfilled: type =
  ServiceType::BESTEFFORT or (
    designSpeed <= providedBy.designSpeed and
      (if type = ServiceType::IMPORTANT then
        designSpeed <= providedBy.backup.designSpeed
      else type = ServiceType::WEAKCONTRACT or
       providedBy.backup <> null endif));
```

Listing 1. Selected constraints for the motivating example.

of the associated backup cluster to be equal or greater than the designed speed of the service itself.

Example Model. We consider the example as shown in the lower part of Fig. 1. We have a single *important* service with speed *4* which is provided by the Cluster *WebCl*. This cluster has a low-speed backup which is not sufficient (cf. speedFulfilled) since its design speed is lower than the speed required by the main service. In the following, we will show how this example will be handled by our extended validation, content-assist and quick fix providers.

V. The Approach

Our approach has been realized in the *IntellEdit framework*. The functionality of our framework is applied both during the generation of *Advanced DSML Editors* (cf. Fig. 2) as well as during their execution (cf. Fig. 3).

In the following, we illustrate our framework in terms of the generation and execution of advanced DSML editors and detail on our generic approach for the validation of constraints, the production of content-assist suggestions, as well as the production of quick fix solutions.

A. Generation of Advanced DSML Editors with IntellEdit

IntellEdit is a Java framework build on top of the Eclipse Modeling Framework (EMF) [26], Xtext [5], Multiobjective Evolutionary Algorithm (MOEA) framework [27], and OCL [8]. It has been specifically created to realize our intentions of leveraging the advantages of comprehensive language definitions. Therefore, the language engineer first applies the EMF facilities, to specify a DSML *Metamodel* based on Ecore, i.e., a common standard for metamodeling. Next, the language engineer augments the *Metamodel* with *Textual Formal Constraints* (cf. Fig. 2) in terms of OCL constraints that enable to specify, e.g., restrictions, such as, equations having equal values on both sides. Thus, for a model to be valid, all constraints defined in its associated metamodel must be fulfilled.

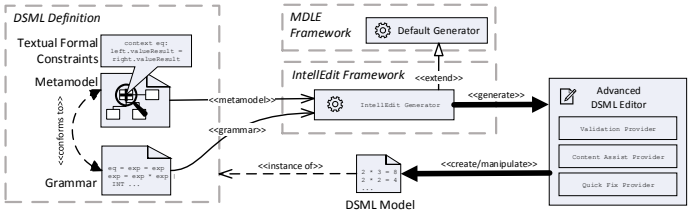


Fig. 2. IntellEdit Framework: Adv. DSML Editor Generation.

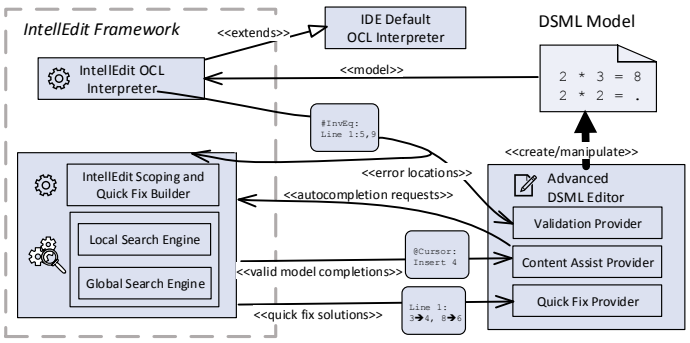


Fig. 3. IntellEdit Framework: Adv. DSML Editor Usage.

Having completed the *DSML Definition*, i.e., representing the language definition, our framework generates a conforming *Advanced DSML Editor*, which provides a customized *Validation Provider*, *Content Assist Provider*, and *Quick Fix Provider*. To do so, we apply both EMF default facilities as well as the *IntellEdit Generator*. Moreover, by separating classes used for validation, content-assist, and quick fix solutions from other editing classes, *IntellEdit* enables straightforward injection into automatically generated files.

During the execution of the generated *Advanced DSML Editor*, which is used to construct and manipulate models that represent instances of the DSML definition, the editor issues run-time requests to our framework. Subsequently, *IntellEdit*, which employs the MOEA framework during this step, establishes advanced validation, content assist, and quick fix results and replies them to the *Advanced DSML Editor*.

B. Generic Approach for Advanced Constraint Validation

Editors that are generated by state-of-the-art tools tend to visualize single inconsistencies in terms of the context in which the violation occurs instead of a more exact location, e.g., the union of all minimal error causes. This is problematic, as the language modeler is presented with imprecise information on the correctness and incorrectness of the model. Proper visualization of error causes can contribute to a better modeling process that avoids subsequent errors [19].

Definition of Change Action Requirement Types. In our approach, we consider change action requirement types that are constructed and evaluated in evaluation trees. For the purpose of acquiring potentially relevant error locations, we perform a runtime-analysis of the expression evaluation, by comparing *expected result* with *actual result*, and return locations where deviations occur. To do so, we store the expected result value as one or more of the following eleven change action requirement types that provide a good basis for finding some correcting changes with decent speed in the first two layers of our search algorithm. These change action requirement types can be seen as a specialization of the general model constraints to basic constraints on individual model element properties. First, different types of expected results, generally corresponding to a simple set of conditions or single values conditions, are represented by equal(v) and different(v) and denote that expressions should have the same value or a different value than the given one. Secondly, *true* and *false*

correspond to equal(*true*) and equal(*false*), respectively. Next, the expected results increase(v) and increaseexcl(v) as well as decrease(v) and decreaseexcl(v) for Integers denote that a value should be at least or at most a specific value. Moreover, contains(v) and excludes(v) denote that an expected set of results should contain or should not contain the value v. Likewise, the anticipated size of an expected set of results is indicated by minsize(c) and maxsize(c). Finally, the definition of change on the expected result indicates that the criteria of a particular expression are not known.

Moreover, the deviation of many typical OCL constraint evaluation can be encoded using these basic requirement types, including setting properties, various collection operations, simple inequalities, Boolean operators and single-parameter functions with a computable inverse set.

```
1  name: fixgen
2  input: f ⊆ Car, (e, v) ∈ {(Expression, Result)}
3  output: SURE m : (Expression, Result) → Car // result fixes
4    f ← f \ check(f, v)
5    m ← (e, v) ↦ felse ∀a, b : (a, b) ↦ {}
6    u ← f //unhandled fixes
7    for p ∈ availableGenerators(e.type())
8      (h, m_s) ← p.get(e, v, f)
9      u ← u \ h
10     m ← (a, b) ↦ (m(a, b) ∪ m_s(a, b))
11   if u ≠ ∅
12       for (e_s, v_s) ∈ subresults(e, v)
13           m ← (a, b) ↦ (m(a, b) ∪ ((e_s, v_s), Change))
14   for (e_s, v_s) ∈ subresults(e, v)
15       m_s ← fixgen(m(e_s, v_s), (e_s, v_s)
16       m ← (a, b) ↦ (m(a, b) ∪ m_s(a, b))
```

Listing 2. Change Action Requirement generation.

Evaluation of Change Action Requirement Types. The basic generation of change action requirements *Car* as defined above is described in Listing 2. In case the evaluation of a change action requirement yields a positive result, i.e., the requirement is fulfilled, changes of existing sub-evaluations are not required and hence, propagation is skipped. On one hand, expression types for which expected results are known may be specified accordingly using a change action requirement generator. On the other hand, expression types for which expected results are not known have their most general expected result, i.e., change, propagated to all sub-expressions.

There are several change action requirement generators which can be used in our approach. For operations with a finite operation table and a certain target value, all possibly expected results of a sub-expression are determined by looking at the operation table, i.e., set of operation results for input values.

```
1  input f : (x_1, ..., x_n) → y Function, c_1, ..., c_n current assignment,
2        vexpected value
3  output r : P((x_1, ..., x_n)) //set−minimal changes
4    b = f^{-1}(v)
5    b_h ← {{(i, k_i)|k_i ≠ x_i}|(k_1, ..., k_n) ∈ b}
6    b_h ← b_h \ {A ∈ b_h, ∃B ∈ b_h : B ⊊ A}
7    r = {(r_1, ..., r_n)|A ∈ b_h, r_i = if ∃(i, z) ∈ A then z else x_i end}
```

Listing 3. Preparation of change action requirements generator operation for finite functions.

In detail, assuming an operation $f(x_1, ..., x_n) = v$, with current sub-expression evaluations $y_1, ..., y_n$ for all sub-expressions and expected value v, a suitable change $\{y_k \rightarrow y'_k, ...\}$ fulfills $f(y_1, ..., y'_k, ...y_n) = v$. The computation of all suitable set-minimal changes, which is computationally intensive, is performed once for every operation on application startup and is depicted in Listing 3. Listing 4 shows an algorithm which actually applies such a fix to an existing evaluation. It handles all $\mathrm{equal}(x)$ requirements by trying to find base values which make the function return the expected value x and tries to recursively apply these change action requirements.

TABLE I
REDUCED TABLE FOR MAKING THE OR EXPRESSION TRUE.

OR	A:false	A:true
B:false	A → *true*, B → *true*	-
B:true	-	-

Table I shows a simple example of such a table for making *OR true*. Only in the case of both *A* and *B* being false, propagation is triggered. In that case, both *A* and *B* may become true.

Further, for the Boolean conditions $v_1{=}v_2$ and $v_1{\neq}v_2$ and expected values true and false, the expected values can be propagated to equal(v_2) and equal(v_1) for the first and second sub-expression, i.e., $v_1{=}|{\neq}v_2$:=*true*|*false*, respectively. For the other combinations, different can be applied. Similarly, increase and decrease are utilized for the inequalities $<$ and $>$, respectively. Likewise, for inclusion and exclusion in set memberships of expected Boolean values, contains and excludes may be used, respectively. Furthermore, expected Integer values of set sizes are converted with minsize and maxsize. Inclusions and exclusions of set selection operators, i.e., select, $\{x \in S|\mathsf{cond}(x)\}$, are mapped to ($i$) inclusions and exclusions of the source set S and (ii) an expected value of *true* for every object that should be contained in the set and *false* for every object that should not be contained in the set. Set collections, i.e., collect, $\bigcup_{y \in S} x \in f(y)$, propagate their excluded elements both to $f(y)$ and to remove y from S where an unwanted value is calculated and their included elements yield an inclusion in $f(y)$ as well as any additions to S. The monotone operations $\cup$ and $\cap$ allow to directly propagate the expected results. Likewise, additions allow propagating increase and decrease requests.

Following the propagation of the expected result, potentially erroneous sub-expressions indicate where the expected result does not match the actual result. As a result of the applied grammar reflecting most model access operations, particularly

```
1  input: (e, v) ∈ {(Expression, Result)}, fCar
2  output: (h, m_s) ∈ {Car, Reqtable}
3  h ← {f_s|f_s ∈ f, ∃x : f_s = equal(t)}
4  m_s ← (a, b) ↦ {}
5  for equal(t) ∈ h
6      s_r ← {(e_i, x_i)|(e_i, x_i) ∈ subresults(e, v))
7      s ← generateChangeReq(e, x_1, ..., x_n, t)
8      if s = ∅ //No fix, so don't handle!
9          h = h \ equal(t)
10     else
11       for (e_i, x_i) ∈ s_r //Add requirement values
12         m_s = (a, b) ↦ m_s ∪ {equal(t_i)|(t_1, ..., t_n) ∈ s}
```

Listing 4. Application of change action requirements generator operation for finite functions.

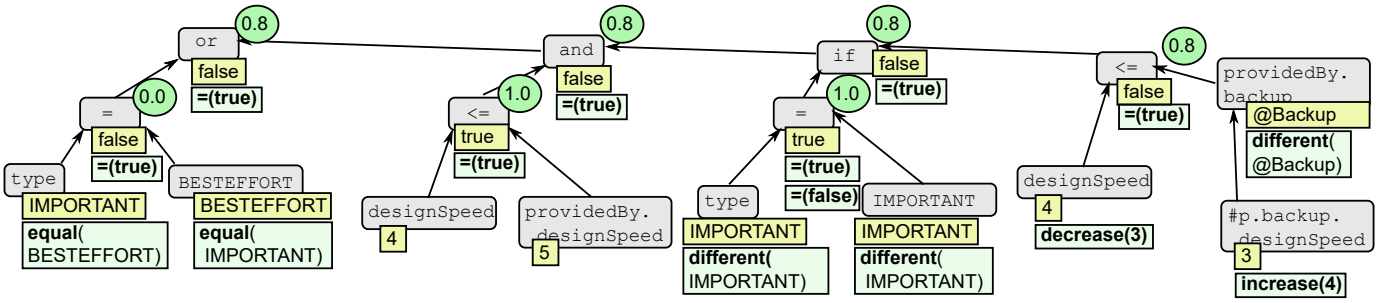


Fig. 4. Failed evaluation tree for the running example expression (cf. Listing 1).

object feature access operations and object instance selection operations, an error marker as well as annotations are placed at locations where sub-expressions are violated and therefore the expected result is not valid. By fulfilling change requirements for a sub-expression, this sub-expression could evaluate to true and thus contribute to making the whole expression true. It might be sufficient to fulfill a subset of all change requirements.

Let us consider the evaluation tree of the failed constraint of our running example as shown in Fig. 4. The syntactic tree is shown in a monospace font. Below each subexpression node, the current result is listed first, followed by the expected result. The numbers in the top right indicate the equality of actual result versus expected result. The constraint failed because the type was *IMPORTANT* and not *BESTEFFORT* and the design speed of the backup was not high enough. To find out what can be done to repair the constraint, we set the expected value to *true* and propagate it. The Boolean expression or is *true* if at least one subexpression is *true*, so we propagate the expected value *true* to both subexpressions. The first subexpression is an *equal* which is *true* if the value of *type* is equal to *BESTEFFORT*, so we set the expected value of *type* to *BESTEFFORT* and the expected value of *BESTEFFORT* to *IMPORTANT*. To make the and Boolean expression *true*, both subexpressions should return *true*. The first subexpression already returns *true* so there is nothing to propagate further and the *designSpeed* feature also does not need to be highlighted. However, the second subexpression does not yet return *true*. An if may evaluate to *true* if the condition is fulfilled and the then-subtree is *true* or the condition is not fulfilled and the else-subtree is *true*, so we try both cases in the conditional equals. This equals expression is currently *true* and can be made *false* by setting either subexpression to a different value. For space reasons, we only discuss making the then-part *true*. It is *true* if the *designSpeed* is at most the design speed of the backup, so the backup may be changed, the service's *designSpeed* may be decreased or the backup's *designSpeed* may be increased.

C. Generic Approach for Advanced Content-Assist

Our approach ranks discovered content-assist suggestions, such that most favorable solutions are placed first. In principle, solutions violating the lowest number of constraints are considered favorable. Constraints are recursively evaluated by matching the closest match of a set of expected values with actual values of sub-expressions. The *closeness measure* in our approach is established by computing distance metrics for types. In detail, metrics for Strings and numbers are established by computing the Levenshtein distance and the numeric equality $e^{-|v_1-v_2|}$, respectively. Moreover, Boolean operators are mapped to double operators where *true* and *false* map to *0* and *1*, respectively. Additionally, and and max are mapped to min and max, respectively.

If we consider the example in Fig. 4, the only part of the expression which is not completely true or false is the rightmost inequality where the inequality is nearly fulfilled (measure 0.8). This value is propagated to the top so the constraint is considered mostly fulfilled.

D. Generic Approach for Advanced Quick Fix Solutions

Quick fix solutions are actions—placed at locations where model values are changed—that can be executed to increase model quality by decreasing the number of violated constraints. Zeller defines the cause of an effect as the minimal difference between the world where the effect, i.e., a constraint violation in our case, occurs and an alternate world where the effect does not occur [28]. Hence, to find the actual cause for a constraint violation we have to search for the closest possible world in which the violation does not occur. Thus, we established quick fix solutions by ranking them according to the least change principle, i.e., favoring solutions causing minimal differences for how some effect comes to existence.

Our approach takes use of different SBSE-techniques for discovering quick fix solutions by applying a three-stage search process, which involves local as well as global search, with varying neighborhood search [29] for the exploration of the search space. Moreover, all involved search runs are both performed in the background and continuously deliver solutions that are eventually displayed to the editor user. In case a model change occurs, previously discovered quick fix solutions are re-evaluated based on the current model. In general, new values and value changes are randomly sampled for each specific data type. Moreover, changes are sampled in a way that small changes are more likely than large changes. For example, the likelihood of changing $n+1$ characters is only 10% of the likelihood of changing n characters.

Local Search. Our generic approach performs local search in two stages, i.e., by a small local search and a large local search. In general, model changes based on a single violated constraint are explored, which either resolve the violation,

i.e., constraint becomes true, or guide the constraint resolution closer to true, i.e., the involved model change has a high score of resolving the violation. A simple hill climbing algorithm with backtracking is used for the local search [30]. Of course, locally resolving an error may yield to new violations in other model areas. Therefore, as a countermeasure, our approach sorts quick fix solutions in terms of (i) their score of resolving the involved single violation and (ii) how many violations exist, i.e., involving existing violation and new violation, in the model after applying the solution.

The *small local search* explores model changes where the expression analyzer may find concrete possible changes for resolving a single expression. In detail, model changes include the application of equal, contains, excludes, increase, and decrease on objects features to reach the expected results. The attempted value of model change operations increase and decrease refers to the border of what is applicable.

The *large local search* considers changes for single violated expressions as well. However, it considers model changes involving manipulation of all feature values as well as all object instances that occur as part of a constraint expression in which the evaluated result does not match the expected result.

Global Search. Our generic algorithm for global search applies genetic search techniques allowing arbitrary model changes on the entire model for discovering a broad range of quick fix solutions that may not yet have been discovered by local search algorithms. Hence, as a short-cut, quick fix solutions that result from the global search may directly be applied by the editor user. Practical implementations may consider multi-objective search algorithms, such as the NSGA-II algorithm, i.e., also applied in *IntellEdit*, which allows considering change amount as well as the number of fixed and violated constraints. Hence, the editor user is presented with the results of the current Pareto-front of such changes.

Our motivating example model (cf. upper-part of Fig. 1) may be repaired by a variety of repair actions that directly fulfill the expression and may thus be proposed to the editor user. For example, reducing the quality of the Service to *BESTEFFORT* and increasing the design speed of the backup.

E. Running Advanced DSML Editors with IntellEdit

During the creation and manipulation of *DSML Model*s by the editor user (cf. Fig. 3), several interactions between the *Advanced DSML Editor* and *IntellEdit* occur to provide editor assistance. Moreover, to make the most appropriate use of resources consumed by our framework, any change made to the model is immediately communicated to our framework. The two major components of our framework are represented by the *IntellEdit OCL Interpreter* and the *IntellEdit Content-Assist and Quick Fix Builder*, which contains a *Local Search Engine* as well as a *Global Search Engine*.

The general workflow entails the delivery of identified error location details by our OCL interpreter to both our internal content-assist and quick fix builder as well as the external *Validation Provider* in the editor implementation. As a result, the editor user is provided with the validation results of the model being currently created or manipulated in the editor. Next, the editor's *Content Assist Provider* requests and retrieves any available results from our content-assist and quick fix builder. There, for our motivating example model, a line stating "`type = `" for our main Service will result in our content-assist builder to prefer to insert "`BESTEFFORT`" at the current cursor location. Finally, the editor's *Quick Fix Provider* is continuously being equipped with quick fix solutions by our content-assist and quick fix builder. Hence, as soon as solutions like changing the type to `BESTEFFORT` or increasing the Backup cluster speed to `4` are found, they are made available to the quick fix provider and can, therefore, be applied by the editor user operating the editor.

F. Current Limitations

First, the propagation for operations with a finite operation table and a certain target value has (only) been implemented for the Boolean operations and, or, not, implies and xor. Secondly, if the type of a collection source set S is finite, additions that do not yield a matching $f(y)$ are not (yet) avoided in our prototypical implementation. Third, the content-assist algorithm considers feature additions and updates from at most 1000 domain feature values. Forth, relying on the definition of cause and effect [28], applying IntellEdit quick fix solutions can make the failure disappear but may not necessarily represent a correction but at least a good workaround. Fifth, for specific expressions such as and, the OCL evaluation engine may only produce parts of the result, e.g., if the first subexpression is *false*, the second subexpression will not be evaluated which reduces the number of displayed validation errors. This could be fixed by changing the evaluation engine. Sixth, arbitrary function calls that combine the value of more than one variable can not be handled other than propagating change. Errors in an OCL constraint with such function calls may thus only be fixed by the second and third layer of our search algorithm.

VI. Evaluation

The evaluation of our approach implemented by the IntellEdit framework has been conducted separately for our enhanced validation, content-assist suggestions, and quick fix solutions. We applied the DSML of our running example for randomized generations of example instances that each contains 20 objects as well as 100 associations and feature values.

A. Validation

The validation results provided by IntellEdit have been evaluated based on whether it improves the estimation of locations where a model should be changed to resolve errors. We introduced erroneous changes at arbitrary locations starting from a valid model. We consider error feature locations to be accurate if they show exactly those locations as erroneous. In detail, we considered error locations to be correct if (i) the feature is indicated as erroneous for every deleted feature value, (ii) the container of the feature is indicated as erroneous for deleted objects, and (iii) the whole object is indicated

TABLE II
VALIDATION EVALUATION RES. (WEIGHTED/UNWEIGHTED).

Feature	IntellEdit	XText
Precision	44.8% / 55.1%	14.0% / 21.4%
Recall	37.5% / 77.1%	41.5% / 65.2%
F-Measure	0.408 / 0.643	0.209 / 0.322

TABLE III
CONSTRAINT VIOLATIONS IN THE GENERATED MODELS.

Use of IntellEdit Suggestions	0%	25%	50%	75%	100%
Avg. Cons. violations	8.8	6.5	3.8	2.4	0.0

as erroneous for created objects. Moreover, if a complete object is indicated as erroneous, all its features are indicated as erroneous as well. Finally, we determined precision and recall of our approach and of Xtext by comparing the set of erroneously indicated features and objects.

Table II shows the evaluation results of the IntellEdit validation compared with the Xtext validation for indicating erroneous locations in a model. In total, we performed 6442 evaluations on 50 generated models. The left-hand numbers represent the average precision and recall per single changed feature and the right-hand numbers represent the average precision and recall per single change. Hence, the left-hand numbers focus on large changes and the right-hand numbers on small changes. Our results state that the validation precision achieved by IntellEdit is nearly three times as high as that achieved by Xtext. Thus, the IntellEdit validation displays only erroneous features instead of indicating the whole object as erroneous and hence provides more accurate evidence on which parts of the model should be changed. In terms of recall, IntellEdit produces a higher recall per expression, i.e., performing better on expressions leading to small model changes, and a lower recall per edited feature. However, the recall for larger changes in our evaluation results is low in general. Hence, the validation of IntellEdit doubles the F-measure when compared with Xtext. To summarize, the validation results produced by IntellEdit improve existing precision three-fold while keeping recall measures of Xtext intact. Thus, IntellEdit indicates locations in the model that are more likely to be relevant for repairing violated constraints.

B. Content-Assist Suggestions

The purpose of content-assist suggestions provided by IntellEdit is to support editor users in constructing consistent models. Hence, to evaluate content-assist suggestions, we first randomly generate model containment hierarchies and then assign features by using the result of our content-assist suggestions that has the highest score in 0% (i.e. completely randomly), 25%, 50%, 75%, and 100% of the cases, i.e., simulating the combination of random suggestions with IntellEdit-provided suggestions. Finally, we compare the number of violated constraints (cf. Table III).

Our results indicate that the IntellEdit content-assist provider is effective in preserving constraints in models. On one hand, highly scored IntellEdit content-assist suggestions, i.e., suggestions that are listed on top, do not introduce constraint violations in the model. On the other hand, valid random assignments that do not consider eventual restrictions introduced by formal constraints, i.e., similar to those suggested by the Xtext default content-assist provider, are likely to introduce constraint violations in the model.

C. Quick Fix Solutions

The purpose of evaluating quick fix solutions provided by IntellEdit is to find out if they lead to a model with fewer constraint violations. In detail, we randomly generate models containing random values and apply up to ten IntellEdit quick fix suggestions created as a result of our three-stage search process. Moreover, to avoid the risk of simply deleting violated parts that subsequently may yield empty models, we do not select quick fix suggestions based on remaining constraint violations but on newly fulfilled violated constraints.

Our results show that applying quick fix suggestions on 97 randomly generated models leads to a reduction of the total number of violated constraints in 89 of the same models. Moreover, the number of violated constraints could not be improved in eight models. In detail, one model contained the same number of violations and seven models introduced further violations. To summarize, the quick fix solutions provided by IntellEdit were able to reduce the number of violated constraints from 7.9 to 2.6 (about 67%).

D. Threats to Validity

In general, our evaluation is limited to our applied experimental object, i.e, the DSML for modeling service clusters, and hence may not be a representative of all kinds of languages. Moreover, the faults introduced in our faulty models may deviate from faults introduced by human users. Similarly, workflows of human users, which intend to resolve faults, may deviate from those suggested by our content assist and quick fix solutions. In fact, even for the running example, the typical change aimed at, i.e. increasing the speed of existing servers or adding new servers to meet the service requirements, is not found in our framework because making the contract weaker is considered as just a single change fulfilling all constraints while the change aimed for would need more model changes, resulting in high costs, without any intermediate improvements. Further, the error feature location in our evaluation assumes that a revert of the erroneous change is correct. However, there may exist different changes that can lead to good results. Hence, our evaluation does not clarify whether increased precision will also be beneficial for the editor user. Likewise, it is not known whether editor users mainly prefer to create models that do or do not violate constraints. For example, the content-assist provider of the Xtext default DSML editor allows to introduce violations that may or may not hinder the editor user in the creation of models. However, we hypothesize that picking content-assist suggestions that do not introduce violations, i.e., listed before other IntellEdit-provided content-assist suggestions, are more fruitful for the editor user. As a result, the current threats

TABLE IV
STRUCTURAL CONSTRAINTS IN LANGUAGE WORKBENCHES.

Language Workbench	Formal Constraint Definition	OCL	Manual Constraint Validation	Automated Constraint Validation	Manual Scoping	Automated Scoping	Manual Quick Fix	Automated Quick Fix
Xtext	◒	◒	●	●	●	◒	●	◒
Jetbrains MPS	◒	○	●	●	●	●	●	○
Rascal	○	○	●	○	N/A[2]	○	●	○
Melange	◒	◒	●	○	○	○	○	○
Spoofax	◒	○	●	●	●	○	○	○
Epsilon	●	●	●	○	○	○	●	○
Whole Platform	○	○	○	○	○	○	○	○
DrRacket	●	○	●	●	○	○	●	○
Eco	○	○	●	○	●	○	○	○
Ensō	○	○	◒	◒	○	○	○	○
MontiCore	○	○	●	○	○	○	○	○
MetaEdit+	●	○	●	●	●	○	●	○
SugarJ	◒	○	●	◒	●	◒	○	○
Visual Studio	○	○	●	○	●	○	○	○

Legend: ● = Full Support; ◒ = Partial Support; ○ = No Support

to validity will be tackled in the future by conducting an industrial user study considering multiple real-world languages to evaluate whether our evaluation results can be reflected by an increase in usability of DSML editors as well as efficiency in the process of model creation carried out by human modelers.

VII. STRUCTURAL CONSTRAINTS IN LANGUAGE WORKBENCHES

For the purpose of evaluating the applicability of our approach, which requires the specification of structural constraints as part of the language metamodel for creating automated support of advanced editing capabilities, we investigated existing languages workbenches listed by the Language Workbench Challenge 2016[3] that have been updated at least once within the last three years as well as Microsoft Visual Studio [31]. Hence, the intention of this comparison is to explore capabilities of existing language workbenches in terms of (i) formal constraint definition, (ii) manual specification and automated generation of constraint validation, (iii) manual specification and automated generation of content-assist, and (iv) manual specification and automated generation of quick fix solutions. Additionally, we intended to establish the compatibility of existing language workbenches with our approach and *IntellEdit* in particular.

[2]Not enough information for an evaluation of manual scoping in Rascal could be retrieved.

[3]Language workbenches listed by the Language Workbench Challenge 2016 can be found online at http://2016.splashcon.org/track/lwc2016.

In detail, we reviewed the following levels of language editor feature support offered by languages, which may be created by the language workbench under investigation. First, we examined the feasibility of language workbenches in their definition of languages that include formal constraints, such as OCL constraints. Second, we explored the range of support that is provided to language designers for manually specifying invariant-enforcement mechanisms in languages, which are generated by the investigated language workbench. Third, we inspected language workbenches in terms of their support for manually specifying content-assist amendment mechanisms, i.e., their intended capability for formulating content-assist suggestions, within applicable contexts. Fourth, we evaluated a language workbench's intended support for manual specification of quick fix solutions. Fifth, in the case of language workbenches that provide the capability of restricting languages by constructing formal constraints, we examined their degree of automating the generation of invariant-enforcement mechanisms as well as quick fix solutions available in resulting language editors.

Our review took into account several associated language workbench artifacts including (i) documentation, (ii) publications, (iii) direct communication with developers of the language workbenches, and (iv) implementation. We also took into account that a language workbench may not provide the support for structural constraints but still offer approaches for the implementation of customized validation, content-assist, and quick fix solutions. Note that, the "Onion" language workbench, listed in the Language Workbench Challenge, has been omitted because publication, documentation, and implementation is not publicly available. Hence, our investigation took into account Xtext [5], JetBrains MPS [32], Rascal [33], Melange [34], Spoofax [35], Epsilon [36], Whole Platform [37], DrRacket [38], Eco [39], Ensō [40], MontiCore [41], MetaEdit+ [42], SugarJ [43], and Microsoft Visual Studio [31]. Note that the results of our investigation (cf. Table IV), which includes direct communication with tool developers, may be still prone to misinterpretations. Hence, a more extensive comparison, e.g., involving the implementation of comprehensive DSMLs in each workbench, may produce deviating results.

We found that none of the workbenches offer both (i) the capability of enhancing language definitions with formal constraints, such as OCL, and (ii) the application of formal constraints in the automated generation of advanced editor support, i.e., validation, content-assist, and quick fix implementations. Moreover, we found that structural constraints imposed to languages, which are produced by the investigated language workbenches, is performed through either application of (i) workbench-specific languages, (ii) workbench-specific aspects, (iii) GPL code, or (iv) a combination of such.

Full support for structural constraints is provided in Epsilon, DrRacket, and MetaEdit+. However, Epsilon does not represent an actual language workbench, which may be used to generated a DSML, but a family of languages, including the Epsilon Validation Language (EVL), based on OCL. Hence,

EVL can be used to formulate constraints that may be used in conjunction with other tools, e.g., Xtext, in the creation of DSMLs. In detail, EVL may be used to augment and evaluate OCL invariants in Ecore models. Therefore, our approach can be applied in conjunction with EVL as well as profit from advantages of EVL over OCL [16], e.g., support for dependent constraints and enhanced flexibility in context definition. DrRacket allows formal specification as well as automated validation of contracts, manual validation by creating custom functions to display error locations, and specification of manual quick fix solutions by defining functions that make contracts valid. MetaEdit+ employs both a metamodeling language for the specification of graphs, objects, properties, ports, roles, and relationships as well as a scripting language for the specification and automated validation of structural constraints. Moreover, MetaEdit+ allows to manually specify validation, content-assist, and quick fix implementations. However, MetaEdit+ represents a non-generative approach, i.e., language definition and language use is done in the same tool instance, and hence only allows the non-generative part of our approach to be applied.

Xtext, JetBrains MPS, Melange, Spoofax, and SugarJ provide partial support for structural constraints. Both Xtext and Melange build on EMF by supporting language specifications in Ecore metamodels, which may be augmented with OCL constraints as done in our IntellEdit framework. Melange itself does not generate DSMLs but may be used in conjunction with other EMF-based tools, such as Xtext or Sirius [44], for generating DSML implementations. Hence, the capabilities of DSMLs specified with Melange are bound to EMF-based tools that are applied for generating language implementations. Additionally, Melange provides their own metalanguage for specifying model types based on the definition of groups of related types, i.e., a set of constraints over admissible model graphs. Both Xtext and Melange apply the default OCL interpreter provided by EMF, which offers automated validation that is limited in terms of displaying useful error messages. Moreover, Xtext also provides facilities for manually specifying content-assist and quick fix implementations and automatically offers basic content-assist and quick fix solutions for DSMLs.

The JetBrains MPS language workbench offers their own languages for specifying structural constraints. First, MPS offers automated validation and content-assist for constraints specified in their dedicated constraint language as well as manual specifications of validation, content-assist, and quick fix solutions by resorting back to GPL implementations. Second, structural constraints in MPS are limited to the expressiveness of their dedicated constraint language, which does not represent the full support by OCL. The Spoofax language workbench also employs their own constraint specification language which seems to only cover a subset of OCL. For example, Spoofax does not allow the specification of iterators in structural constraints. Hence, MPS and Spoofax may only partially profit from the advantages provided by our approach.

Restrictions in SugarJ may be defined using regular expressions that enable, together with imports of predefined syntactic extensions, validation, and content-assist capabilities. However, as a result of missing support for quick fix implementations as well as limited expressiveness of regular expressions, our approach may not be applicable in SugarJ.

Visual Studio offers no support for formal constraints. Validators and rules which change model elements depending on other model elements can be manually implemented to define validation, scoping, and some limited repairs. This approach could be applicable if OCL support would be added to Visual Studio, but this is unlikely since EMF/OCL is implemented in Java which is not the language of choice for Microsoft.

VIII. Conclusion and Future Work

We presented an approach for leveraging language analysis for the automated generation of advanced editors in modern language workbenches. Hence, by applying our approach, language designers can construct formal constraints as part of their language definition to overcome the high effort of implementing advanced editing features by hand. Consequently, editor users are empowered with the benefits of such features, which include enhanced validation, content-assist suggestions, and quick fix solutions that are beyond those created by state-of-the-art language workbenches. The evaluation of our approach yields (i) a two-fold improvement in validation over existing solutions, (ii) content-assist suggestions that do not introduce new constraint violations, and (iii) quick fix solutions that reduce the number of violated constraints by about 67%.

Concerning ongoing and future work, we are currently in the process of extending the evaluation of our approach by conducting experiments involving substantially larger DSMLs as well as DSMLs that are used in an industrial context. The results produced by such an evaluation will provide further insights on performance, scalability, and practical feasibility of our approach. Moreover, we intend to precisely determine any advances that the application of our approach may introduce. In detail, an extensive comparison of IntellEdit with other solutions will be conducted by implementing a comparable DSML in each of them and evaluate their capability in terms of provided validation, content assist, and quick fix solutions. Further, we want to measure the effect of applying different primitive type conversions, e.g., multiplication instead of min for the and operator, on the quality of produced content assist and quick fix results. Finally, in case additional evaluation results provide further evidence in favor of applying our approach, we plan to perform an empirical study, which will be conducted within an industrial context, measuring the effects on efficiency and productivity of language designers as well as editor users applying our approach.

Acknowledgment

We graciously thank the authors and developers of the language workbenches mentioned in Section VII for their effort in supporting us by discussing the capabilities of their tools.

REFERENCES

[1] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[2] J. Gray, S. Neema, J. Tolvanen, A. S. Gokhale, S. Kelly, and J. Sprinkle, "Domain-Specific Modeling," in *Handbook of Dynamic System Modeling.*, 2007.

[3] U. Frank, "Multi-perspective enterprise modeling: foundational concepts, prospects and future research challenges," *Software and System Modeling*, vol. 13, no. 3, pp. 941–962, 2014.

[4] M. Fowler, "Language workbenches: The killer-app for domain specific languages," 2005. [Online]. Available: http://www.martinfowler.com/articles/languageWorkbench.html

[5] M. Eysholdt and H. Behrens, "Xtext: Implement your Language Faster than the Quick and Dirty Way," in *Companion Proc. of OOPSLA*. ACM, 2010, pp. 307–309.

[6] J. Favre, "Languages evolve too! Changing the Software Time Scale," in *Proceedings of IWPSE*, 2005, pp. 33–44.

[7] B. Meyers and H. Vangheluwe, "A framework for evolution of modelling languages," *Sci. Comput. Program.*, vol. 76, no. 12, pp. 1223–1246, 2011.

[8] J. Cabot and M. Gogolla, "Object constraint language (OCL): A definitive guide," in *Proceedings of SFM*, 2012, pp. 58–90.

[9] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency Management with Repair Actions," in *Proceedings of ICSE*, 2003, pp. 455–464.

[10] T. Mens, R. V. D. Straeten, and M. D'Hondt, "Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis," in *Proceeding of MODELS*, 2006, pp. 200–214.

[11] P. Neubauer, "Towards model-driven software language modernization," in *Proceedings of the Doctoral Symposium and Projects Showcase @ STAF*, 2016, pp. 11–20.

[12] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[13] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[14] J. Whittle, J. E. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014.

[15] P. Neubauer, A. Bergmayr, T. Mayerhofer, J. Troya, and M. Wimmer, "XMLText: From XML Schema to Xtext," in *Proceedings of SLE*, 2015, pp. 71–76.

[16] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages," in *Rigorous Methods for Software Construction and Analysis*, 2009, pp. 204–218.

[17] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *Proceedings of ASE*, 2008, pp. 99–108.

[18] A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *Proceedings of ASE*, 2012, pp. 220–229.

[19] ——, "Determining the cause of a design model inconsistency," *IEEE Trans. Software Eng.*, vol. 39, no. 11, pp. 1531–1548, 2013.

[20] M. Harman, "The Current State and Future of Search Based Software Engineering," in *Proceedings of FOSE*, 2007, pp. 342–357.

[21] Á. Hegedüs, Á. Horváth, I. Ráth, M. C. Branco, and D. Varró, "Quick fix generation for dsmls," in *Proceedings of VL/HCC*, 2011, pp. 17–24.

[22] M. A. A. da Silva, A. Mougenot, X. Blanc, and R. Bendraou, "Towards Automated Inconsistency Handling in Design Models," in *Proceedings of CAiSE*, 2010, pp. 348–362.

[23] S. Ali, M. Z. Z. Iqbal, A. Arcuri, and L. C. Briand, "A Search-Based OCL Constraint Solver for Model-Based Test Data Generation," in *Proceedings of QSIC*, 2011, pp. 41–50.

[24] O. Semeráth, Á. Barta, Á. Horváth, Z. Szatmári, and D. Varró, "Formal validation of domain-specific languages with derived features and well-formedness constraints," *Software and Systems Modeling*, pp. 1–36, 2015.

[25] K. Deb, "Multi-objective optimization," in *Search methodologies*. Springer, 2014, pp. 403–449.

[26] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[27] "Multiobjective Evolutionary Algorithms (MOEA) framework," http://moeaframework.org, accessed: 2016-12-15.

[28] A. Zeller, *Why programs fail - a guide to systematic debugging*. Elsevier, 2006.

[29] P. Hansen and N. Mladenović, "Variable neighborhood search," in *Search methodologies*. Springer, 2014, pp. 313–337.

[30] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Artificial intelligence: a modern approach*. Prentice Hall, 2003, vol. 2.

[31] "Modeling SDK for Visual Studio," https://msdn.microsoft.com/en-us/library/bb126413.aspx, accessed: 2016-12-15.

[32] M. Voelter, "Language and IDE modularization and composition with MPS," in *Proceedings of GTTSE*, 2011, pp. 383–430.

[33] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of SCAM*, 2009, pp. 168–177.

[34] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J. Jézéquel, "Melange: a meta-language for modular and reusable development of DSLs," in *Proceedings of SLE*, 2015, pp. 25–36.

[35] L. C. L. Kats and E. Visser, "The spoofax language workbench: rules for declarative specification of languages and IDEs," in *Proceedings of OOPSLA*, 2010, pp. 444–463.

[36] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack, "The Epsilon Generation Language," in *Proceedings of ECMDA-FA*, 2008, pp. 1–16.

[37] R. Solmi, "Whole platform," Ph.D. dissertation, University of Bologna, 2005.

[38] M. Flatt, "Creating languages in Racket," *Communications of ACM*, vol. 55, no. 1, pp. 48–56, 2012.

[39] L. Diekmann and L. Tratt, "Eco: A Language Composition Editor," in *Proceedings of SLE*, 2014, pp. 82–101.

[40] T. van der Storm, W. R. Cook, and A. Loh, "Object Grammars," in *Proceedings of SLE*, 2012, pp. 4–23.

[41] H. Krahn, B. Rumpe, and S. Völkel, "Monticore: a framework for compositional development of domain specific languages," *CoRR*, vol. abs/1409.2367, 2014.

[42] K. Smolander, K. Lyytinen, V. Tahvanainen, and P. Marttiin, "MetaEdit - A Flexible Graphical Environment for Methodology Modelling," in *Proceedings of CAiSE*, 1991, pp. 168–193.

[43] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann, "SugarJ: library-based syntactic language extensibility," in *Proceedings of OOPSLA*, 2011, pp. 391–406.

[44] V. Viyovi, M. Maksimovi, and B. Perisi, "Sirius: A rapid development of DSM graphical editor," in *Proceedings of INES*, 2014, pp. 233–238.

Virtual Textual Model Composition for Supporting Versioning and Aspect-Orientation*

Robert Bill
Institute of Software Technology and Interactive Systems, TU Wien
Vienna, Austria
bill@big.tuwien.ac.at

Patrick Neubauer
Department of Computer Science, University of York
York, England
patrick.neubauer@york.ac.uk

Manuel Wimmer
CDL-MINT, TU Wien
Vienna, Austria
wimmer@big.tuwien.ac.at

Abstract

The maintenance of modern systems often requires developers to perform complex and error-prone cognitive tasks, which are caused by the obscurity, redundancy, and irrelevancy of code, distracting from essential maintenance tasks. Typical maintenance scenarios include multiple branches of code in repositories, which involves dealing with branch-interdependent changes, and aspects in aspect-oriented development, which requires in-depth knowledge of behavior-interdependent changes. Thus, merging branched files as well as validating the behavior of statically composed code requires developers to conduct exhaustive individual introspection.

In this work we present VirtualEdit for associative, commutative, and invertible model composition. It allows simultaneous editing of multiple model versions or variants through dynamically derived virtual models. We implemented the approach in terms of an open-source framework that enables multi-version editing and aspect-orientation by selectively focusing on specific parts of code, which are significant for a particular engineering task. The VirtualEdit framework is evaluated based on its application to the most popular publicly available Xtext-based languages. Our results indicate that VirtualEdit can be applied to existing languages with reasonably low effort.

CCS Concepts • **Software and its engineering Model-driven software engineering**; **Domain specific languages**; *Software configuration management and version control systems*;

Keywords Aspect-oriented modeling, model-driven engineering, model virtualization, aspect weaving, model versioning

ACM Reference Format:
Robert Bill, Patrick Neubauer, and Manuel Wimmer. 2017. Virtual Textual Model Composition for Supporting Versioning and Aspect-Orientation. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136037

1 Introduction

Model composition [19, 24], i.e., also referred to as system composition in a wider engineering perspective, presents a basic model-driven engineering (MDE) process. It involves the combination of multiple models for a variety of operations, such as the identification of conflicts across input models and undesirable emergent properties in composed models [30]. Thus, model composition represents a foundation for essential model management tasks such as model transformation, model comparison, and model merging [3, 29].

State-of-the-art model composition approaches often present limitations in editing composed models, such as the ability to edit arbitrary composed models. Thus, to build a composed model it is necessary to perform a variety of different operations including merging and splitting of multiple input models, which is usually achieved by establishing and maintaining dedicated model transformations. Moreover, to enable users to edit composed models as well as synchronize any changes from the composed model to respective input models and vice versa, it is common to cultivate and sustain yet another set of model transformations or resort to bi-directional transformations, i.e., requiring less, but more complex, individual transformations.

Although model transformations are employed to realize model composition scenarios, they do not possess appropriate means to ease or overcome their manual creation and maintenance. For example, an intrinsic requirement for performing splitting or merging of models includes the fabrication of a result for the union of a given set of model elements. Consequently, current solutions require the developer to manually handle a variety of different model transformations and operations and thus lead to complex, tedious, and

*Updates to the artifact available at http://virtualedit.big.tuwien.ac.at

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136037

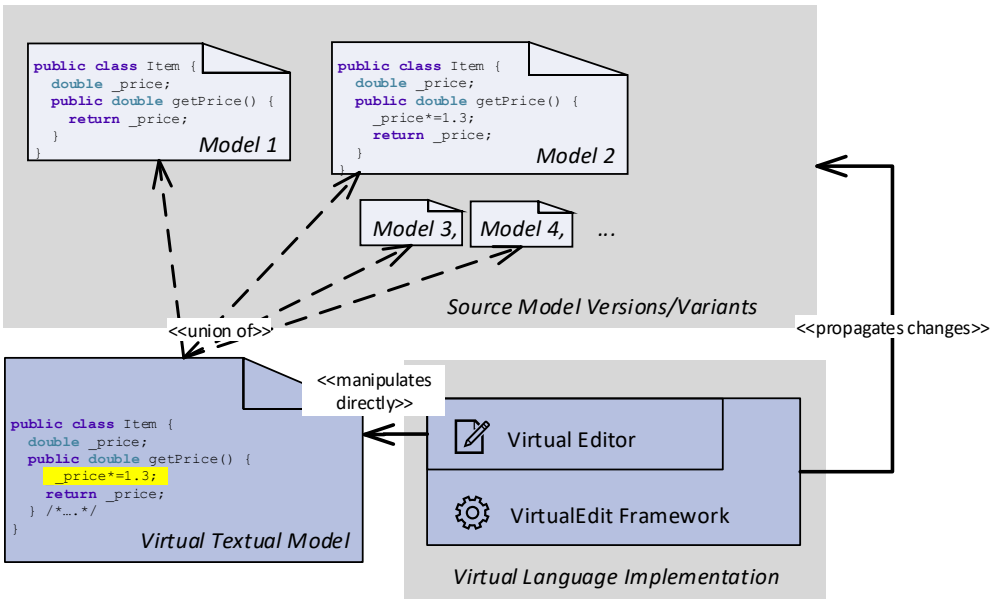


Figure 1. Loading multiple models into a merged virtual model.

time-consuming tasks for construction and maintenance of model composition solutions.

In this paper we present an approach to significantly ease dynamic model composition for multiple models by overcoming manual construction and maintenance of model transformations through the combination of text-based model composition and virtualization[1] in what is subsequently referred to as VirtualEdit. By employing VirtualEdit, several transformations and operations necessary for performing model composition, such as union, are provided by instantiating virtualization concepts and hence fully preserve the ability to edit both, the composed model as well as the input models.

To gauge the prospects of our approach and the validity of its implementation, we evaluate VirtualEdit within the domain of aspect-oriented modeling (AOM) [33] and model versioning [5] with real-world languages from Github.

In the next sections we describe (*i*) two use cases within the domain of model versioning and aspect-oriented modeling, (*ii*) VirtualEdit by illustrating a simple demonstration case and presenting design rationales of multi-version models, their augmentation in a virtual editor, and the synchronization of changes, (*iii*) an evaluation based on a set of real-world languages retrieved from Github, (*iv*) a section of related work, and (*v*) the conclusion including a presentation of future work.

[1]In the most general form, virtualization refers to a concept, which creates the illusion of dealing with a real object, whereas being a proxy mechanism that redirects access and manipulation requests to the virtualized object.

2 Use Cases

In this section, we describe the basic architectures for employing the VirtualEdit framework in the domain of (*i*) model versioning, i.e., management of concurrently evolving models as well as (*ii*) AOM, i.e., management of woven models, which consist of base models and multiple aspect applications, while dynamically displaying or withholding applied aspects.

In both cases, the user edits a single virtual model that represents a composition of different models. The behavior of the VirtualEdit framework includes the following. First, in case a user dynamically changes the focus, the displayed view is automatically adapted. Secondly, in case a user issues an edit operation, the operation is propagated to suitable base models.

In general, there are two types of merge procedures to be distinguished, which include (*i*) merging multiple models, such as a base model and aspects applied on that, into a single model and (*ii*) merging multiple versions of the same model. In the former, the application of aspects translates to an addition of model differences to a base model of which the union of models is built Changes are only propagated to a single suitable difference. In the latter, changes are propagated to all relevant models.

2.1 Use Case 1: Model Versioning

Fig. 1 presents an overview of VirtualEdit for the use case of editing multiple model versions at the same time. The VirtualEdit framework reads in all model versions in form of a structured tree, i.e., each model may have zero or more successors. Our model representation format uses

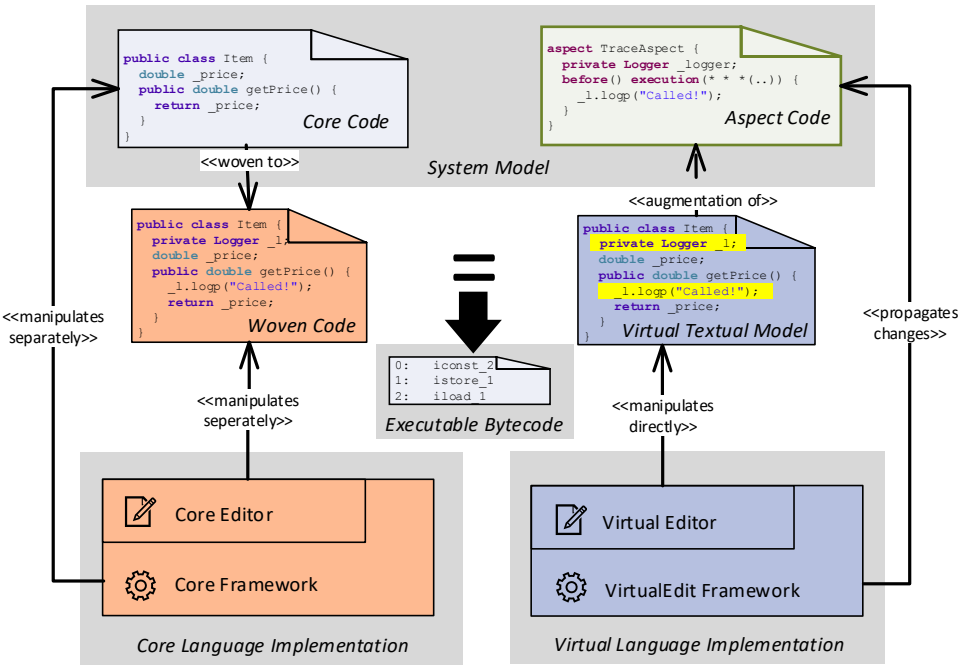


Figure 2. Conventional AOM approach (left-hand side) compared with our proposed virtualization approach (right-hand side).

IDs for identification. If there are no IDs, artificial IDs are calculated by matching each model with it's successor[2]. Then, the virtual edit framework builds the union of all models and displays them in form of a (virtual) textual model - similar to how current version management tools show (text) merges, but being model-aware and supporting an unlimited amount of models. The model elements, which are presented in our use cases, are equivalent to code snippets that only occur in the highlighted subset of all models. A user may select to view only a subset of all model versions considered at a given point in time by which the editor updates its content accordingly. In case changes are performed, these changes are propagated back to all source models of models that are active in the current view.

2.2 Use Case 2: Aspect-Oriented Modeling

Fig. 2 presents an overview of the AOM example in comparison with the conventional procedure of manipulating and debugging a *system model*, which is composed of both *core code*, and *aspect code*. Usually, the *Core Editor*, e.g. the Eclipse Java Editor, of the *Core Language Implementation*, e.g., the AspectJ framework, which is built on Java, manipulates the system model, which is transformed to *woven code*, i.e., representing aspect code intertwined with core code, or directly to *executable byte code*, i.e., no intermediate *woven code* is produced. Finally, a compiler, such as the Java compiler, transforms woven code to *executable byte code* that can be debugged by employing the *Core Debugger*.

In contrast, in our approach the *Virtual Language Implementation* represents a virtualized version of the *Core Language Implementation*, that enables performing operations which require extensive effort when compared with their operation in terms of the *Core Language Implementation*. For example, usually there are no indicators in the *woven code* that state where and how *core code* has been modified by *aspect code*. Consequently, a user can not differentiate between non-generated code, i.e., *core code*, and generated code, i.e., the *woven code* produced by the weaving process.

In contrast, our virtual language implementations apply a *Virtual Editor* that produces a *Virtual Textual Model*, i.e., equal to woven code, by applying model transformations, which are enriched with meta-data that allows the differentiation of non-generated and generated parts of a model. In detail, such meta-data, which contains information that associates elements with being part of either source or target of the transformation as well as if they have been modified. In other words, in our approach, the *Virtual Textual Model* essentially represents a particular view to the *system model*. Furthermore, the virtual editor allows disabling and enabling the visualization of particular parts of the *system model* which potentially decrease overall complexity and alleviate speed of versioning tasks due to a reduction of code that previously required a manual investigation by the developer.

[2]The current implementation uses EMF Compare to derive the matches, but is designed to be extensible

Item.vjava

```
1 package shoppingcart;
2
3 public class Item {
4     String _id;
5     double _price;
6
7     ...
8     public double getPrice ( ) {
9         return _price ;
10    }
11    ...
12 }
```

TraceAspect.vaspect

```
1 aspect TraceAspect {
2     private Logger _logger = new Logger();
3     pointcut traceMethods(): execution(* * *(..));
4     before() traceMethods() {
5         _logger.logp((CLASS_NAME+("."+METHOD_NAME)));
6     }
7 }
```

Figure 3. The Aspect "TraceAspect" (upper-right) an the base code in the Eclipse Editor (left).

3 The Approach

This section describes VirtualEdit, i.e., our virtual text-based model composition approach, which has been implemented in terms of the VirtualEdit framework[3]. We implemented VirtualEdit based on MDE technologies, in particular the Eclipse modelling framework (EMF) [28], the graph transformation framework Henshin [4] and the language workbench Xtext [11].

VirtualEdit provides a base implementation for arbitrary Xtext-based domain-specific modeling languages (DSMLs), and thus, may be applied to various languages. Its capabilities can be enabled in arbitrary Xtext-based DSMLs by replacing their binding from the original Xtext to our virtualized editor. This is achieved by changing four static lines of code and adding a dependency.

In the rest of this section we first demonstrate our virtual textual model composition approach in the context of AOM and then present a detailed report on the implementation of VirtualEdit by focusing on the design rationale of (multi-version) models, their augmentation in the virtual editor, and the synchronization of model changes.

[3]A ready-to-use virtual machine image and Eclipse instance of the VirtualEdit framework can be retrieved online from http://virtualedit.big.tuwien.ac.at.

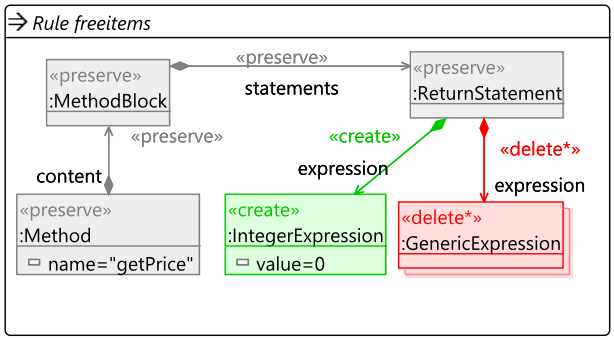


Figure 4. Henshin rule "freeitems" making items free.

3.1 Demonstration Case

The demonstration case represents the functionality of a shopping-cart, which has been originally provided by Laddad et al. [20]. For sake of brevity, we focus on the Item class, which models a shopping item with a price that can be purchased (cf. left part of Fig. 3). In terms of aspects, "TraceAspect" (cf. upper-right part of Fig. 3) depicts a typical AOM monitoring technique based on logging method calls. Moreover, the Henshin rule "freeitems" (cf. Fig. 4) represents, purely for demonstration purposes, a malicious aspect that has been introduced by a developer to make products free.[4]

3.2 Design Rationale and Realization

In this subsection we present the design rationale used for our approach for aiming towards multi-versions and aspect-oriented model representations, respectively, as well as its realization in the realm of metamodeling frameworks and language workbenches.

3.2.1 Data Structure

To achieve *dynamic model composition* in a generic way, requires establishing a novel data structure for indicating the location of model elements in order to replace them within their conventional structure. In order to support dynamic visualization and inhibition of specific aspects without requiring the re-computation of the effect that such actions have on other aspects, and thus woven code, a model representation that allows adding and removing certain deltas for any occurring delta is required. However, even simple structures, such as sequences with Integer indices give counter-intuitive results when employing conventional delta structures. For example, if such lists are used for virtual model composition, each list index has to be updated for every addition or removal that is depicted in the delta model and thus may not fulfill editor performance requirements. Hence, if such a solution is considered, it leads to the following problems.

[4]Henshin only deletes matched elements, i.e., not the newly created IntegerExpression

First, assume the list [a,d], where first b, then c, and finally d is added to get [a,b,c,d]. A typical index-based delta representation could be [add(b,1),add(c,2)]. Applying only the second delta on [a,d] would yield [a,d,c] which does not represent the expected result, i.e., [a,c,d]. Secondly, as a result of the importance of the order in which deltas are being applied, efficiently adding or removing deltas requires the manual specification of resolutions for certain types of conflicts [1].

Thus, we need a model composition that builds a group $(\Delta, \oplus)$ for applying model deltas, i.e., that $\oplus$ is commutative, associative, has a neutral element and has inverse elements for each element. This means that we can unapply a delta by building the inverse and adding it to the result model. A model has the same representation as a model delta. Hence, to solve this, we represent models $\Delta = (I, e, c, A, R)$ as functions with object identifiers I. The metamodel is assumed to be a typical MOF-based metamodel with attributes $\mathcal{A}$, references $\mathcal{R}$, and classes C. The model contains (i) an *existence* set $e : \text{Set}_I$ defining which objects exist, (ii) a classification function $c : I \to \text{Set}_C$ associating an object to its class and all super classes and implemented interfaces, (iii) attributes $A \ni a_{i,V} : I \to \text{List}_V$ associating a list of values of type V to each object and (iv) references $R \ni r_i : I \to \text{List}_I$ associating a list of identifiers to each object. Although unordered, unique attributes and references could map to a set instead of a list from a model representation point of view, we use lists to maintain the specific element order in the edited document. Moreover, instead of representing a containment explicitly, we recursively change the existence value when an object is added to or removed from a containment. As a result, all feature values are kept in case a previously removed object is recreated. There may be additional sets $e_r : \text{Set}_I$ for each resource to define which elements are directly contained in a resource. Finally, models of multiple resources are summed up in order to get the model of the complete resource set.

3.2.2 Groups (Sets and Lists)

Additionally, to get a group, we define sets and lists as follows. A set of type V is defined as $\text{Set}_V : V \to \mathbb{N}$, where an element is in the set if and only if the function value is greater than zero. A list of type V is defined as $\text{List}_V : P \to \text{Set}_V$, where each $p \in P$ is a sequence of integers. Thus, allowing the definition of a lexicographic order for elements of P.

Consequently, insertions can be performed at any location. For example, the indices of an element that is inserted between elements with indices `[1]` and `[2,1]` might be `[1,1]`, `[1,6]` or `[2,-1,0]`. In particular, arbitrary order-preserving suffixes may be generated to ensure that list-position-clashes can not occur. For instance, if the current position is `[1,2]` and the identifier of the aspect is `9`, then the resulting position amounts to `[1,2,9]`. Further, in case unique Integer identifiers can not be assigned to a particular aspect, the aspect's URI is added instead, and thus, re-establishes an aspect's uniqueness.

3.2.3 Functions and Identifiers

All functions are stored as partial functions. Functions of type $I \to \mathbb{N}$ return the value 0 for undefined identifiers. Functions of type $V \to R$, with R being a function will return a default function, i.e., the function without any value stored, for each parameter. We assume that $\text{sdom} : (A \to B) \to P(A)$ will return the actual assigned domain for each function. With that, $\text{sdom}(a \oplus b) = \text{sdom}(a \cup b) = \text{sdom}(a) \cup \text{sdom}(b)$ As example, consider the `Item` class depicted in Fig. 3 which currently has two attributes `_id` and `_price`. This model excerpt would look as follows in our representation. We assume the ID of the Item class to be i_{Item}. Then, the existence set contains that object and two objects for the attributes, i.e., $e = \{i_{\text{Item}}, i_{\text{atid}}, i_{\text{atprice}}\}$ which is expressed as $e = f(i) = \{i_{\text{Item}} \mapsto 1, i_{\text{atid}} \mapsto 1, i_{\text{atprice}} \mapsto 1, i \mapsto 0$ else, where the last part $i \mapsto 0$ is the default value, not stored, and omitted in the following. $\text{sdom}(e)$ would yield $\{i_{\text{Item}}, i_{\text{atid}}, i_{\text{atprice}}\}$.

The `Item` class's only type is`class` and the other objects are attributes. Thus, the classification function c is defined as $c : f(i) = \{i_{\text{Item}} \mapsto \{\text{class}\}, \{i_{\text{atid}}, i_{\text{atprice}}\} \mapsto \{\text{attribute}\}\}$. We have three attributes: two references, one for storing objects in a class and one for storing attribute types and one attribute for storing names.

The class-attribute reference is defined as $r_i(i) = \{i_{\text{Item}} \mapsto [i_{\text{atid}}, i_{\text{atprice}}]\}$, with the list containing two elements, e.g. at position [1,0] and [2,0], which would yield the representation $l(p) = \{[1,0] \mapsto (i_{\text{atid}} \mapsto 1), [2,0] \mapsto (i_{\text{atprice}} \mapsto 1)$, i.e., $l([1,0])$ returns a function which associates 1 to the identifier i_{atid} and 0 for all other identifiers, i.e., only the identifier i_{atid} is stored in the position $[1,0]$.

All model operation functions should allow adding and removing models dynamically. A suitable way of that is to make them build a group. Then a model can be removed by adding the inverse. Thus, we base our model operation functions on the usual addition which is a well-known group.

As a result, our model sum function $\oplus$ is defined as follows:

$$a \oplus b = \begin{cases} I \to \mathbb{N}, i \mapsto a(i) + b(i) & a, b \text{ are Sets} \\ V \to R, p \mapsto a(p) \oplus b(p) & a, b \text{ Lists or functions} \\ V \to R \text{ with } R \text{ List,} & \text{Set or general function} \end{cases}$$

3.2.4 Precedence and Conflict Resolution

The model difference function $\ominus$ is defined with $-$ instead of $+$ and presents the inverse of the model sum function $\oplus$. By construction, merge conflicts appear to be resolved implicitly. Deleting objects takes precedence over updating any attribute values and adding new links to these objects[5]. Updating values takes precedence over deleting them. Updating values differently results in both values being added to the feature. However, as the merge is virtual and thus occurs in only in memory, these conflict resolutions are not persisted in the formalization. Custom conflict resolutions

[5]Our implementation deviates from this by currently allowing links to non-existent objects to reduce the number of textual changes.

could be stored as additional delta model that is able to resurrect deleted objects and thus delete incorrectly updated feature values.

For instance, if we want to add an attribute `_logger` to our model, we can add the original model to a model containing the logger attribute with name and type, i.e., $e = \{i_{\text{logger}}\}$, $a_{\text{name,String}} = (i_{\text{logger}} \mapsto ['\_logger'])$, ...

The model union function $\cup$ can be defined analogous:

$$a \cup b = \begin{cases} I \to \mathbb{N}, i \mapsto \max(a(i), b(i)) & a, b \text{ Sets} \\ V \to R, p \mapsto a(p) \cup b(p) & a, b \text{ Lists/Funct.} \\ V \to R \text{ with } R \text{ List, Set or general function} \end{cases}$$

The model intersection function $\bigcap$ is defined similarly, with min instead of max. The main practical difference between *sum* and *union* lies in that the neutral element of the sum is the empty model while the neutral element for both union and intersection is the model itself. Thus, we use the sum to compose changes, i.e., model deltas, and the union to compose multiple pre-existing models.

Please note the semantic differences between four potential ways of combining multiple model versions based on the same metamodel, i.e., combining the models with (*i*) the $\oplus$ operator, (*ii*) the $\cup$ operator, (*iii*) the $\cap$ operator and (*iv*) calculating model differences between model versions and adding these differences to the base model (see Fig. 7). The resulting model of (*i*) and (*ii*) are nearly the same as the sum of positive integers is always greater than zero and the sum of non-negative integers is always zero if all summands are zero. They build a max-model, i.e., a model containing all objects and all feature values of the base models. We use the second variant to ease the implementation of edit operations. Variant (*iii*) builds a min-model, i.e., a model containing only objects and feature-values contained in all base models. Variant (*vi*) uses the previously defined conflict resolution.

3.2.5 Interoperability and Multi-Versions

For interoperability purposes with EMF, we provide an ECore view which provides virtual `EObjects` and virtual `ELists` which are backed by our representational structure. For single-valued features, only the first value in the list is used. For multi-valued features, all values are used. Currently, Strings are considered as atomic values. Thus, two changes to a single String will result in two Strings being stored in the feature slot. If a String would be represented as a list of characters, changes may be included in the more fine-grained character level, and thus, only require the generation of a single String containing both changes.

Multi-Versions can be directly described by the model union function above. We change the union model by modifying all base model functions, i.e., functions returning a natural number, so that they return the target value, i.e., we apply the edit model e to the current virtual union u as $u := u \oplus e$ with $(a \cup b) \oplus e = (a \oplus e) \cup (b \oplus e)$. At any time, we can choose to apply an edit operations only to a selection of model versions.

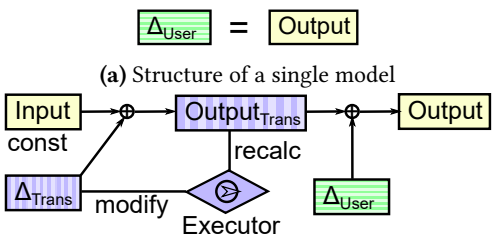
3.2.6 Delta Model Computation

In the following, we present how the delta model is calculated from aspect applications and how aspect applications can be combined. The data structure, which has been defined beforehand, can also be used to define the structure of aspects, i.e., model transformations, as depicted in our demonstration case. Moreover, the model sum is used to add the base model to the derived changes.

In our approach, all transformation application instances have a single output result and a single model Δ_{User} for user changes. Fig. 5a shows the pseudo-transformation structure of a single base model. The base model is both transformation result and user edit model. Fig. 5b shows the structure of a single aspect instance. An aspect transforms an input to build an output. Hence, instead of directly modifying the input model, all modifications are stored in the delta model Δ_{Trans}. Next, in case the transformation is reapplied, Δ_{Trans} is cleared and $\text{Output}_{\text{Trans}}$ is recalculated. Subsequently, user changes are stored in a separate model that has not been seen by the transformation and thus does not affect the results in case the transformation is re-applied. Finally, the final output is both represented by the composition $\oplus$ of transformation output as well as the user delta.

Our approach makes sure that object identifiers, which have been created by transformations, remain constant for multiple transformation executions so that edit operations remain valid.

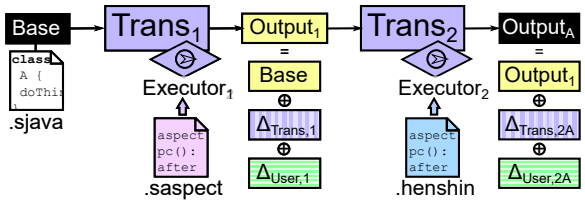
Further, to avoid cases in which aspects accidentally create objects that have been removed by other objects, every created object identifier is prefixed with the aspect's id and every created list position is suffixed with the aspect's index or id. Thus, object identifiers are calculated as a function of the executed transformation rule and its parameters. In detail, user edit operations are stored in a suitable delta, which avoids that aspects immediately undo them when they are re-executed, and trigger the following heuristic: If a part is deleted or added, then the edit operation is stored in the delta of the transformation instance of (*i*) the last deletion or



(a) Structure of a single model

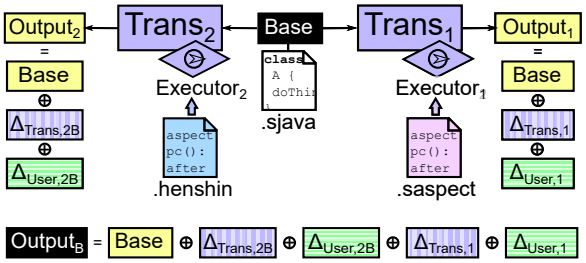
(b) Structure of a single transformation

Figure 5. Structure of transformation providers



(a) chained calculation

Figure 6. Transformation chain generated output model.



(b) Independent calculation

Figure 7. Parallel transformations generated output model.

addition operation of this part, if any or (*ii*) the addition of the containing object. As a result of storing object removals invisibly to transformations, the view, on which an aspect operates, does not contain a removal operation and hence cannot unapply it.

Moreover, we carry out transformations in sequential order as well as storing model-change operations immediately following their creating transformation, i.e., the transformation that initially created the model that has been modified. In detail, the architecture of our approach implicitly isolates aspects, i.e., represented by transformations in our approach, from model-change operations. Additionally, any preceding transformations are enabled to initialize models without limitations, which otherwise may be imposed by aspects.

Fig. 6 and Fig. 7 show two cases of how our approach employs multiple aspects to produce an output result from a base model that result in distinctively computed delta models. First, the result of the first aspect is used as input for the execution of the second aspect (cf. right part of Fig. 6), i.e., representing a chained-calculation. Secondly, both aspects are executed on the base model (cf. Fig. 7). In both cases, the output is the model sum of base model, transformation deltas, and user deltas. However, in the first case, all aspects are independent from each other and in the second case, the second aspects is able to see all changes that have been performed by the first aspect. Thus, in case the second aspect Trans_2 would add a logging statement to each method, our chained-calculation would also add it to methods that have been generated by the first aspect Trans_1.

Our current implementation supports two kinds of transformation providers. First, the generic Henshin [4] transformation provider allows to define aspects by executing Henshin model transformations. Secondly, the AOM-specific transformation provider for our VAspect language creates transformations from aspect definitions. Finally, transformation provider instances, which have distinct transformation and user deltas, are created for each base model of the VJava language.

3.3 Editor Augmentation

In terms of editor-augmentation, we employ a customized Xtext editor to display our virtual model. In detail, we synchronize the Xtext model with our virtual model as a consequence of the Xtext framework not offering direct model manipulation. As a result, the editor is augmented with information from the virtual model, which is synchronized with the Xtext model.

Moreover, for each structural feature value, we determine all sources, i.e., all deltas and possibly the base model for the AOM use case and all source models for the versioning use case, which have contributed to a particular feature value. In the AOM case, we distinguish between different types of features values: (*i*) *nonderived feature values*, i.e., all such sources are user deltas or the base model, (*ii*) *derived feature values*, i.e., all such sources are transformation deltas, (*iii*) *partly derived feature* values, in all other cases.

In the versioning case, we distinguish between (*i*) *base feature values*, i.e., feature values occurring in all source models and (*ii*) *nonbase feature values*, i.e., feature values not occurring in at least one single source model.

Additionally, at least partially derived feature values and nonbase feature values are highlighted in different colors. Also, our implementation provides an aspect/model selection view that enables users to select specific aspects/models, which they want to see in the editor. In detail, the model that is shown in the editor, is calculated as sum of the base model, all transformation user deltas, and (only) transformation deltas of selected aspects (cf. Fig. 8) or the union of all selected source models. Additionally, any time a user selects aspects to be displayed, the editor view is updated accordingly.

Hence, employing our virtual editor on the demonstration case (cf. Section 3.1) enables viewing and hiding particular aspects (cf. left-hand side of Fig. 8) as well as ease the identification and isolation of undesired behavior that has been woven into the final system due to an error in the aspect definition (cf. right-hand side of Fig. 8). Similarly, we can easily see changes done in specific models.

3.4 Synchronization of VirtualEdit and Xtext

The general synchronization workflow of our approach entails that modifications, which are performed on the *Virtual Textual Model* by employing the virtual editor, trigger the execution of alterations that carry out the necessary adaptations of the system model as well as eventual changes in the content visualized by the virtual editor.

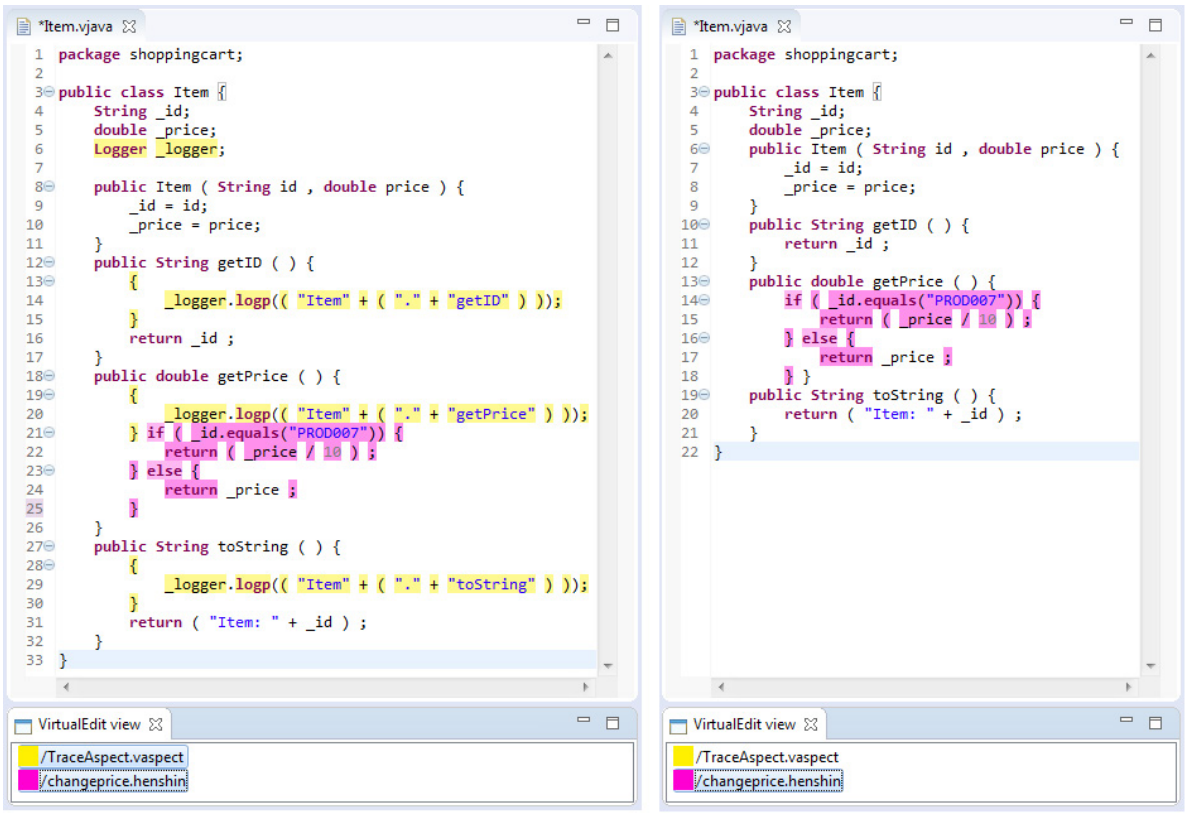


Figure 8. Applying the VirtualEdit virtual editor for the visualization of all available aspects and Henshin rules (left) and the sole visualization of the "`freeitems`" Henshin rule (right).

To ensure the correct matching of elements, we store the target virtual object for each object in the editor as text annotation which we can use to build a correspondence map between Xtext-`EObjects` and VirtualEdit-`EObjects` based on the position of the elements in the text. As a result of the reparse operations performed by the Xtext framework, annotating model elements does not present a viable solution. Hence, the synchronization has to be performed in both directions as well as in a recursive fashion on the root elements of a resource by synchronizing all feature values and their contained model elements. We synchronize feature values by applying only patches to each feature value to avoid unnecessary changes that eventually lead to a loss of formatting.

Xtext model to VirtualEdit model. In this case, if matching elements do not share the same type any more, the type of the VirtualEdit model element is directly changed by changing (only) the object-to-class function. As a result, feature values of repeated type changes are preserved. Elements are created by choosing the correct user edit delta to place the elements in and generating a new URI in that user edit delta.

VirtualEdit model to Xtext model. In this case, if matching elements do not share the same type any more, a new element of the correct type is created and all features values of features which exist in both types are copied. Next, elements are created using standard Ecore facilities for element creation. After the system model has been synchronized, all textual annotations regarding mapping and derivation the status of are updated.

3.5 Current Limitations

Our approach and the VirtualEdit framework currently have the following limitations.

First, the VirtualEdit model composition does not retain Core Model editability for all types of aspects, i.e., it cannot propagate all changes back to the Core Model where it would be possible in principle. In detail, source-level modifiers are represented in terms of primitive operations, i.e., *ADD* (+) and *REMOVE* (-). Therefore, merging, reordering, or interleaving of model elements can not be explicitly represented.

The tooling implementation currently does not make use of all potential features of the approach. For example, (meta-)information contained in the VirtualEdit model like multi-to-single-feature conflicts and exact source locations are not visualized in the editor window.

4 Evaluation

In general, the evaluation of the VirtualEdit framework follows the guidelines for case study research in software engineering [25] and is based on a set of demonstration cases involving the most popular real-world languages of different domains. The objects of study and evaluation results are publicly available on the VirtualEdit project website http://virtualedit.big.tuwien.ac.at.

4.1 Setup

Objective. The objective of the evaluation of our framework is to assess its capability to enable multi-versioning and aspect-orientation applied to real-world languages of different domains.

The cases. Representative cases include a set of publicly available Xtext-based DSMLs and the application of our framework in the context of scenarios that involve multi-versioning and aspect-orientation.

Theory. We hypothesize that the VirtualEdit framework can be applied for enabling multi-versioning and aspect-orientation in real-world Xtext-based DSMLs by selectively focusing on specific parts of their models.

Research questions. RQ1: Is the VirtualEdit framework capable to handle typical multi-versioning and aspect orientation scenarios? ***RQ2***: How integrable is VirtualEdit to real-world Xtext-based language implementations, and thus, offers typical multi-versioning and aspect orientation capabilities?

Selection strategy. First, our selection strategy involves issuing queries to Github for retrieving all projects containing Xtext grammar files. Next, resulting projects are sorted based on their number of stargazers, i.e., amount of users that have added a particular project to their list of starred projects. Finally, the set of our study objects is formed by selecting the five most popular, i.e., highest-ranked according to their number of stargazers, real-world Xtext-based projects that have been retrieved from Github.

Method. The methodology of our evaluation follows the subsequently mentioned steps that are repeated for each object of study. We (*i*) create a new Xtext project by employing the Xtext creation wizard, which creates an exemplary DSML skeleton, (*ii*) replace its skeleton-grammar by a grammar that we retrieved from a real-world Xtext project, (*iii*) execute the project creation workflow to generate an executable DSML implementation, (*iv*) configure the generated implementation to employ VirtualEdit editors for versioning and aspect-orientation, (*v*) select an available real-world model as well as a previous version of the same model available in the history of the real-world Xtext project repository, (*vi*) apply a model transformation that conducts changes to the model, and (*vii*) load both the historical as well as the current model with our VirtualEdit model editor, respectively.

4.2 Results

In the endeavour to answer RQ1, we first examined the extend of the VirtualEdit framework to support typical concepts that appear in the AOM world [33]. Thus, we considered several tutorials and books on AspectJ, e.g., by Laddad [20] and DZone [17], during the design and implementation of the VAspect language, which covers recurring AOM concepts, such as, pointcuts, i.e., matching conditions, and advices, i.e., code modifications. Further, we found that many Java-based aspect languages instrument byte code instead of producing woven code.

To answer RQ2, we validated the applicability of our framework to our set of study objects. First, we found out that the three projects `wesnoth/wesnoth`, `eclipse/smarthome`, and `ufoai/ufoai` could successfully be employed by VirtualEdit and two projects, i.e., `antlr4ide` and `Jnario`, could not be employed as a result of their dependence on additional Java files or other grammars, which require the functionality of imports that is not yet supported by our current implementation. Moreover, we found that replacing the application of the default Xtext editor with our virtual editor can be performed with an acceptable amount of effort and thus enable the use of VJava-files as described earlier. However, we found that VirtualEdit requires valid input models with references that are available within non-imported models. Thus, models that contain such references require manual investigation before they can be displayed by the VirtualEdit editor.

Furthermore, the implementation of the code required to execute VAspect aspect definitions, which produce woven code, is sufficiently easy. On one hand, we found out that AspectJ pointcuts and advices can be seen as transformation context and actions, which has been shown for generic aspects in modeling [23]. For example, *execution* together with *before*, *after* or *around* only add the advice code at a specific point and do not even require dynamic conditions. On the other hand, some advices, such as *cflow* may require a static and thread-local variable that has to be checked at runtime. Thus, we hypothesize that most remaining advices, may be implemented with a similar effort.

During the evaluation, we found numerous bugs in our implementation and a potential limitation. Files in the wesnoth language lose their formatting, possibly due to their use of hidden tokens. However, we think that these bugs are not a result of the approach itself, but rather of implementing the approach without testing enough. In fact, several of the most important bugs were detected and fixed as a result of the evaluation.

To summarize, we conclude that, VirtualEdit is capable to handle multi-versioning and aspect orientation scenarios found in selected literature and our current implementation is integrable to a subset of investigated real-world Xtext-based language implementations.

4.3 Validity

Internal Validity. The internal validity of our evaluation is limited to a subset of Java and AspectJ, which have been implemented in terms of VJava and VAspect, respectively. Hence, the compatibility of our approach with the complete set of concepts available in Java and AspectJ, which have not been applied in any of the investigated examples found in books and tutorials but may be depicted in different AOM applications, has still to be evaluated. In other words, during the construction of VJava and VAspect we did not evaluate the impact and possible limitations of using aspect applications as transformations.

External Validity. Although our evaluation is based on real-world languages of different domains, our findings are limited to a set of investigated demonstration cases. To provide a good level of representativeness of the employed cases, we investigated the most popular publicly-available DSMLs. However, we cannot state any results going beyond the selected cases before making a larger study with a statistically significant amount of DSMLs.

4.4 Discussion

Although the current implementation of VJava does not provide dedicated support for debugging, which would allow the developer to set conditional breakpoints on a generated advice and hence focus on a particular aspect during the debugging process, we hypothesize that such a debugger eases the detection and reasoning behind erroneously applied advices, which are based on dynamic values. In other words, the causes of violated requirements may be found with less effort when selectively enabling advice-postconditions. Moreover, our composition allows developers to easily edit code that has been generated by advices as well as preserve such edit-operations during the re-application of aspects. Thus, existing challenges such as code location and data values as well as limitations of existing fault models, which do not claim to be complete, i.e., able to represent any possible kind of fault [10] are tackled by our approach through the concept of virtualization.

5 Related Work

This section discusses work related to our VirtualEdit model composition approach applied by clustering it in (*i*) View-based Modeling, (*ii*) Model Composition, and (*iii*) AOM.

View-based Modeling. Goldschmidt et al. [13] present a survey that analyzes and organizes existing approaches for various view(point)-based aspects in DSMLs, which are scattered across publications, and contributes a taxonomy on view-based modeling from a tool-oriented perspective. For example, their taxonomy includes means to describe editor capabilities such as "bidirectionality", i.e., ability to synchronize models and their views, and "update strategy", i.e., when to execute synchronization transformations. ModelJoin [7] and EMF Views [6] (previously VirtualEMF [9]), enable the creation of model views, which combine models of different metamodels with an SQL-like syntax. Further, they also use EMF but do not provide a virtual textual editor, which enables dynamic visualization and hiding of aspects.

Generally, view(point)-based approaches typically focus on providing multiple different views on the same model as opposed to one complete or filtered view on multiple models. On the contrary, the VirtualEdit approach enables dynamic views as well as direct, multi-language, and language-independent manipulation to which a view may be associated. As a result, dynamic views and direct multi-language model manipulation, which is achieved by our approach, may ease and speed-up the process of debugging due to decreased complexity and accelerated falsification and localization of errors.

Model Composition. Although model composition has been investigated in literature from various angles, such as (*i*) specific application on model families [26], (*ii*) formal semantics and potential composition operators [15, 31], as well as (*iii*) methods for automating the identification and composition of relationships among elements [12, 18], several challenges have been addressed by EMF Views [6] (previously VirtualEMF [9]), which combines heterogeneous and interrelated models in terms of on-demand computed views defined by an SQL-like and Xtext-based query language, including increased efficiency in memory consumption and formation time caused by data duplication. Similarly to Goldschmidt et al. [13], "View Scopes" represent the visualization of a specific selection of elements. Kolovos et al. [18] introduce the "Epsilon Merging Language (EML)" and an approach to merge multiple models, which are based of different metamodels, into one model. However, users have to manually create EML-based matching rules instead of being automatically provided with a virtualized view that presents the merge-result.

AOM. Hovsepyan et al. [16] show that modeling in terms of *all-aspectual processes*, in which concerns are kept separated, increases modeling performance to up to 20% and results in smaller, less complex, and more modular implementations when compared with hybrid processes, in which concerns are composed. Thus, all-aspectual processes also shorten the versioning cycle.

Schoettle et al. [27] present "TouchCORE" (previously "TouchRAM"), i.e., a modeling tool for Concern-Driven Software Development (CDSD). TouchCORE enhances tracability in CDSD by visualizing feature models, e.g., in terms of class diagrams, and thus exemplifies one use case of our more generic approach. In detail, our approach may be applied in CDSD to derive such visualizations but also in other code weaving or code virtualization scenarios.

Mehmood et al. [22] present a systematic mapping study, which identifies two ongoing distinct lines of research: (*i*)

model weaving as special case of a model-to-model transformation and (*ii*) approaches that transform aspect models into a target AO language, such as AspectJ, and thus rely on target language weavers to deal with crosscutting aspects. They state that approaches following the first line of research, are (*i*) rare and limited in the sense that they disregard advanced pointcut specification and (*ii*) predominantly static and therefore unable to weave and un-weave aspects during model execution [14, 32]. Our approach follows the first line of research on AO presented by Mehmood et al. However, instead of composing base and aspect model separately, both core and crosscutting concerns are edited in one place, i.e., in our VirtualEdit editor. As a result, the modeler stays within the all-aspectual process, which has been found to lead to better performance [16], and simultaneously compose core and crosscutting concerns. Moreover, our approach is dynamic and hence prepared for weaving and un-weaving during model execution. Further, the DSMLs in the AOM use case, on which our approach has been applied, may be able to be extended to support advanced pointcut specifications.

Eaddy et al. [10] highlight challenges associated with source-level debugging, in which debuggers strive to maintain the illusion of a source-level view of program execution by maintaining a correspondence between source and compiled code. They emphasize that the consequence of surrendering correspondence, which is a result of applying various transformations, leads to the inability to perform source-level debugging, which makes matching expected and actual behavior difficult for the human debugger. Thus, giving rise to *code location* problems, i.e., displaying the wrong call stack of source line, or depicting byte code instead of source code, and *data value* problems, i.e., incorrect displaying of new fields or variables, that occur when correspondences between source code variables and memory locations have been obscured. Moreover, Eaddy et al. define "full source-level debugging" as a set of six AOM-specific activities that represent an extension of an AOM fault model by Ceccato et al. [8], i.e., itself an extension of Alexander et al. [2]. Additionally, AOM-specific fault models presented in literature do not claim to be complete and thus their application is limited to particular parts of source code that represents one of those AOM-specific activities. Consequently, faults introduced by (*i*) activities that are not covered by the fault model and (*ii*) faults that arise from base code, are neglected by existing AOM-specific fault models. When compared to our approach, we neither impose limitations on AOM-specific activities and thus specific fault types but provide a framework that is capable to deal with complete source code debugging. Furthermore, as a consequence of preserving the correspondence between source and target (woven) code, the problems associated with *code location* and *data value* are implicitly omitted in our approach.

6 Conclusion and Future Work

In this work, we presented a virtual model composition approach to support versioning and AOM by enabling developers to (*i*) dynamically include or exclude source models from the merged model view, (*ii*) dynamically show and hide individual aspect applications without affecting the actual application of aspects by highlighting elements with their different origins and at the same time (*iii*) preserve editing capabilities by eventually redirecting model operations to the base model or the source models or store them in delta models. Moreover, our model representation enables a commutative and associative addition and a subtraction of models as well as change conflicts to be resolved implicitly.

The results of our initial experience report, i.e., evaluating our approach as well as its implementation in the VirtualEdit framework, indicate that both advices and multiple model versions can be suitably represented in virtual code. Hence, we hypothesize that source-level software versioning significantly benefit from virtual views that are created by our VirtualEdit model composition approach.

However, to evaluate our hypothesis for the aspect orientation showcase, the implementation of a debugger, which require considerable effort but has been done several times for different editors, has to be considered.

Therefore, future work involves the extension of our approach as well as its implementation. Regarding the approach, (*i*) the derivation of identifiers will be made customizable [21], such that developers can specify certain model elements as equal, and (*ii*) the transformation execution and model composition, which are currently separated, will be merged by extending transformation providers to directly derive output model from input model and offer means for asynchronous execution of performance-intensive transformations.

Regarding the implementation, (*i*) the performance will be improved by caching complete models and employ incremental transformations, (*ii*) the implementation of features, which address identified limitations (cf. Section 3.5), (*iii*) the implementation of a debugger and other assistive features in order to fulfill the means for conducting a user study to evaluate the impact of our approach on the productivity in the development of dynamically composed systems and (*v*) make more meta-information about the VirtualEdit model accessible in a user-friendly way, a.o. to support the merge process by the visualization and configuration of tentative merges.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions.

This work is supported by the Austrian agency for international mobility and cooperation in education, science and

research (OeAD), financed by funds from the Austrian Federal Ministry of Science, Research and Economy (BMWFW). under Grant No.: ICM-2016-04969 and by Austrian Federal Ministry of Science, Research and Economy (BMWFW) and the National Foundation for Research, Technology and Development (CDG).

References

[1] Marcus Alanen and Ivan Porres. 2003. Difference and Union of Models. In *Proc. of the 6th Intl. Conf. on UML*. Springer.

[2] Roger T Alexander, James M Bieman, and Anneliese A Andrews. 2004. Towards the systematic testing of aspect-oriented programs. *Technical Report, Colorado State University* (2004).

[3] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios S. Kolovos, Ivan Kurtev, and Richard F. Paige. 2006. A Canonical Scheme for Model Composition. In *Proc. of the 2nd European Conf. on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*. 346–360.

[4] Enrico Biermann. 2010. EMF Model Transformation Based on Graph Transformation: Formal Foundation and Tool Environment. In *Proc. of the 5th Intl. Conf. on Graph Transformations*. Springer.

[5] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. 2012. An Introduction to Model Versioning. In *Formal Methods for Model-Driven Engineering - 12th Intl. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM)*. Springer.

[6] Hugo Brunelière, Jokin García Perez, Manuel Wimmer, and Jordi Cabot. 2015. EMF Views: A View Mechanism for Integrating Heterogeneous Models. In *Proc. of the 34th Intl. Conf. on Conceptual Modeling (ER)*.

[7] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. 2016. View-based model-driven software development with ModelJoin. *Software and System Modeling* 15, 2 (2016), 473–496.

[8] Mariano Ceccato, Paolo Tonella, and Filippo Ricca. 2005. Is AOP code easier or harder to test than OOP code. In *Proc. of the 1st Workshop on Testing Aspect-Oriented Program (WTAOP)*.

[9] Cauê Clasen, Frédéric Jouault, and Jordi Cabot. 2011. VirtualEMF: A Model Virtualization Tool. In *Proc. of Advances in Conceptual Modeling. Recent Developments and New Directions (ER)*.

[10] Marc Eaddy, Alfred V. Aho, Weiping Hu, Paddy McDonald, and Julian Burger. 2007. Debugging Aspect-Enabled Programs. In *Proc. of the 6th Intl. Symposium on Software Composition (SC)*.

[11] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proc. of the ACM Intl. Conf. Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA'10)*.

[12] Franck Fleurey, Benoit Baudry, Robert B. France, and Sudipto Ghosh. 2007. A Generic Approach for Automatic Model Composition. In *Proc. of MODELS*.

[13] Thomas Goldschmidt, Steffen Becker, and Erik Burger. 2012. Towards a Tool-Oriented Taxonomy of View-Based Modelling. In *Proc. of Modellierung*.

[14] Iris Groher and Markus Voelter. 2007. XWeave: models and aspects in concert. In *Proc. of the 10th Intl. Workshop on Aspect-Oriented Modeling*.

[15] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2007. An Algebraic View on the Semantics of Model Composition. In *Proc. of the 3rd European Conf. on Model Driven Architecture- Foundations and Applications (ECMDA-FA)*.

[16] Aram Hovsepyan, Riccardo Scandariato, Stefan Van Baelen, Yolande Berbers, and Wouter Joosen. 2010. From aspect-oriented models to aspect-oriented code?: the maintenance perspective. In *Proc. of the 9th Intl. Conf. on Aspect-Oriented Software Development (AOSD)*.

[17] Kabir Khan. 2008. An Introduction to Aspect-Oriented programming with JBoss AOP. https://dzone.com/articles/an-introduction-aspect-oriente.

[18] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006. Merging Models with the Epsilon Merging Language (EML). In *Proc. of the 9th Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*.

[19] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. 2006. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proc. of the Intl. Workshop on Global Integrated Model Management (GaMMa'06)*.

[20] Ramnivas Laddad. 2003. *AspectJ in action: practical aspect-oriented programming*. Dreamtech Press.

[21] Philip Langer, Manuel Wimmer, Jeff Gray, Gerti Kappel, and Antonio Vallecillo. 2012. Language-Specific Model Versioning Based on Signifiers. *Journal of Object Technology* 11, 3 (2012), 4:1–34.

[22] Abid Mehmood and Dayang N. A. Jawawi. 2013. Aspect-oriented model-driven code generation: A systematic mapping study. *Information & Software Technology* 55, 2 (2013), 395–411.

[23] Katharina Mehner and Gabriele Taentzer. 2005. Supporting Aspect-Oriented Modeling with Graph Transformations. In *Proc. of the Workshop on Early Aspects*.

[24] Dirk Ohst, Michael Welle, and Udo Kelter. 2003. Differences between versions of UML diagrams. In *Proc. of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE)*.

[25] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. 2012. *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley.

[26] Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. 2010. Developing next generation ADLs through MDE techniques. In *Proc. of the 32nd ACM/IEEE Intl. Conf. on Software Engineering (ICSE)*.

[27] Matthias Schöttle, Nishanth Thimmegowda, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. 2015. Feature modelling and traceability for concern-driven software development with TouchCORE. In *Companion Proc. of the 14th Intl. Conf. on Modularity*.

[28] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.

[29] Greg Straw, Geri Georg, Eunjee Song, Sudipto Ghosh, Robert B. France, and James M. Bieman. 2004. Model Composition Directives. In *Proc. of the 7th Intl. Conf. on UML*.

[30] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. 2014. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software and System Modeling* 13, 1 (2014), 239–272.

[31] Antonio Vallecillo. 2010. On the Combination of Domain Specific Modeling Languages. In *Proc. of the 6th European Conf. on Modelling Foundations and Applications (ECMFA)*.

[32] Jon Whittle, Praveen K. Jayaraman, Ahmed M. Elkhodary, Ana Moreira, and João Araújo. 2009. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. *Trans. Aspect-Oriented Software Development* 6 (2009), 191–237.

[33] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elizabeth Kapsammer. 2011. A Survey on UML-based Aspect-oriented Design Modeling. *ACM Comput. Surv.* 43, 4 (2011), 28:1–28:33.

Computer Languages, Systems & Structures 53 (2018) 90–120

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Automated modelling assistance by integrating heterogeneous information sources

Mora Segura Ángel[a,*], Juan de Lara[a], Patrick Neubauer[b], Manuel Wimmer[b]

[a] Modelling & Software Engineering Research Group, Universidad Autónoma de Madrid, Spain
[b] Business Informatics Group CDL-MINT, Business Informatics Group, TU Wien, Austria

ARTICLE INFO

Article history:
Received 8 November 2017
Revised 10 January 2018
Accepted 26 February 2018
Available online 5 March 2018

Keywords:
Modelling
(Meta-)modelling
Modelling assistance
Domain-specific languages
Language engineering

ABSTRACT

Model-Driven Engineering (MDE) uses models as its main assets in the software development process. The structure of a model is described through a meta-model. Even though modelling and meta-modelling are recurrent activities in MDE and a vast amount of MDE tools exist nowadays, they are tasks typically performed in an unassisted way. Usually, these tools cannot extract useful knowledge available in heterogeneous information sources like XML, RDF, CSV or other models and meta-models.

We propose an approach to provide modelling and meta-modelling assistance. The approach gathers heterogeneous information sources in various technological spaces, and represents them uniformly in a common data model. This enables their uniform querying, by means of an extensible mechanism, which can make use of services, e.g., for synonym search and word sense analysis. The query results can then be easily incorporated into the (meta-)model being built. The approach has been realized in the Extremo tool, developed as an Eclipse plugin.

Extremo has been validated in the context of two domains – production systems and process modelling – taking into account a large and complex industrial standard for classification and product description. Further validation results indicate that the integration of Extremo in various modelling environments can be achieved with low effort, and that the tool is able to handle information from most existing technological spaces.

1. Introduction

Model-Driven Engineering (MDE) advocates an active use of models throughout the software development life-cycle. Thus, models can be used to specify, analyse, test, simulate, execute, generate code and maintain the software to be built, among other activities [1–3].

Models are sometimes built with general-purpose modelling languages, such as the Unified Modelling Language (UML) [4]. In other cases, modelling is performed using Domain-Specific Languages (DSLs) [5]. DSLs contain tailored domain-specific primitives and concepts accurately representing the abstractions within a domain, which may lead to simpler, more intensional models. The abstract syntax of a DSL is described by a meta-model, which is itself a model. Meta-models are

* Corresponding author.
E-mail addresses: Angel.MoraS@uam.es (M.S. Ángel), Juan.deLara@uam.es (J. de Lara), neubauer@big.tuwien.ac.at (P. Neubauer), wimmer@big.tuwien.ac.at (M. Wimmer).
URL: http://miso.es (M.S. Ángel), https://www.big.tuwien.ac.at/ (P. Neubauer)

https://doi.org/10.1016/j.cl.2018.02.002
1477-8424/

typically built using class diagrams, describing the set of models considered valid. Thus, the construction of models and meta-models is a recurrent and central activity in MDE projects [6].

High quality models and meta-models are pivotal for the success of MDE projects. They capture the most important concepts of a domain or describe the features of a system. Nevertheless, they are mostly built in an unassisted way, with no mechanisms for reusing existing knowledge. This situation contrasts with modern *programming* IDEs, which support code completion or provide help for using a given API [7,8]. However, in the MDE field, the modeller normally has the burden of creating the model from scratch. For this reason, modellers would greatly benefit from flexible access and reuse of existing knowledge in a domain. This knowledge might be stored on various technological spaces [9,10], including the modelling technical space, but also the XML, ontologies, and RDF technical spaces.

In order to improve this situation, we propose an extensible approach that provides assistance during the modelling process. In our proposal, we extract the information from an extensible set of different technical spaces. For example, in the XML technical space, DTDs or XML schemas as well as specific XML documents are covered by the assistant; while in the modelling technical space, meta-models and models can be queried. This heterogeneous information is stored in a common data model, so that it can be queried and visualized in a uniform way. The query mechanism is extensible and can make use of services, e.g., for synonym search or word sense analysis. The results of the queries are prioritized and aggregated for all information sources in the repositories and can then be incorporated into the (meta-)model under construction.

We have realized this concept in Extremo and provide an open source Eclipse plugin, which is freely available at the Extremo project website[1]. The web site includes short videos illustrating the main concepts explained in this paper as well as a set of resources, which have been used during the evaluation. Extremo's architecture is extensible and modular by the use of Eclipse extension points, and enables the addition of new information sources and types of queries. The assistant has been designed to be easily integrated with external modelling environments, also through extension points.

We have evaluated our approach under several perspectives. First, we show Extremo's usefulness to create DSLs in two case studies. The first one is in the area of process modelling for immigration procedures and the second is in the area of standard-conforming industrial production systems. We have evaluated its extensibility by describing its integration with a variety of modelling tools, ranging from graphical to tree-based editors. In order to evaluate format extensibility (i.e., the ability to import from new technical spaces), we perform an analytical evaluation of the degree of coverage of the data model. The query mechanism is tested by describing a catalogue of common queries for object-oriented notations. Finally, we address a discussion and the lessons learned from the results of the evaluation.

In comparison with our previous work [11], we provide extensions for a set of different technical spaces that include constraint interpreters and an extensible query mechanism. Moreover, Extremo's internal data model has been extended to handle level-agnostic information, i.e., for an arbitrary number of meta-levels. For example, we have integrated XML schemas and multi-level models [12] as information sources, and integrated Extremo with further modelling and meta-modelling environments. Finally, we report on an evaluation based on process modelling and production systems domain case study, using the eCl@ss standard [13] and provide an analytical evaluation on the generality of the data model we propose.

The rest of this paper is organized as follows. Section 2 provides an overview of the approach and its motivation. Section 3 explains the main parts of the assistant: the handling of heterogeneous sources (Section 3.1), the ability to perform queries on them in a uniform and extensible way (Section 3.2), and the handling of constraints (Section 3.3). Section 4 describes the extensible and modular architecture of the assistant, and how it can be integrated with modelling and meta-modelling tools. Section 5 evaluates the approach under three different perspectives, which include the usefulness for (*i*) language engineering, (*ii*) data extensibility, and (*iii*) integrability with external tools. Section 6 compares with related work, and Section 7 presents the conclusions and lines for future research.

2. Motivation and overview

Many technical tasks in software engineering require from access to knowledge found in a variety of formats, ranging from documents in natural language, to semi-structured and structured data. There is a current trend to make such information readily available and easy to embed in different types of artefacts generated during the software construction process [14]. For example, in the programming community, there are efforts to profit from code repositories, and Q&A sites like StackOverflow to automate coding and documentation tasks [15–17]. Some of these approaches are based on a phase of artefact collection, followed by their preprocessing and storage into a uniform database, which then can be queried using appropriate languages [16]. Following this trend, our objective is to make available to the model engineer a plethora of (possibly heterogeneous) resources that can be queried in a uniform way, and embedded into the model being built.

In general, the task of creating a high quality meta-model is complex because it involves two roles: (*i*) a domain expert, who has in-depth knowledge of a particular domain and (*ii*), a meta-modelling expert, who is experienced in object-oriented design and class-based modelling. Nevertheless, many times, the meta-modelling expert is left alone in the construction of a meta-model, or needs to make a decision based on tacit domain knowledge or under-specified language requirements. In

[1] http://miso.es/tools/extremo.html.

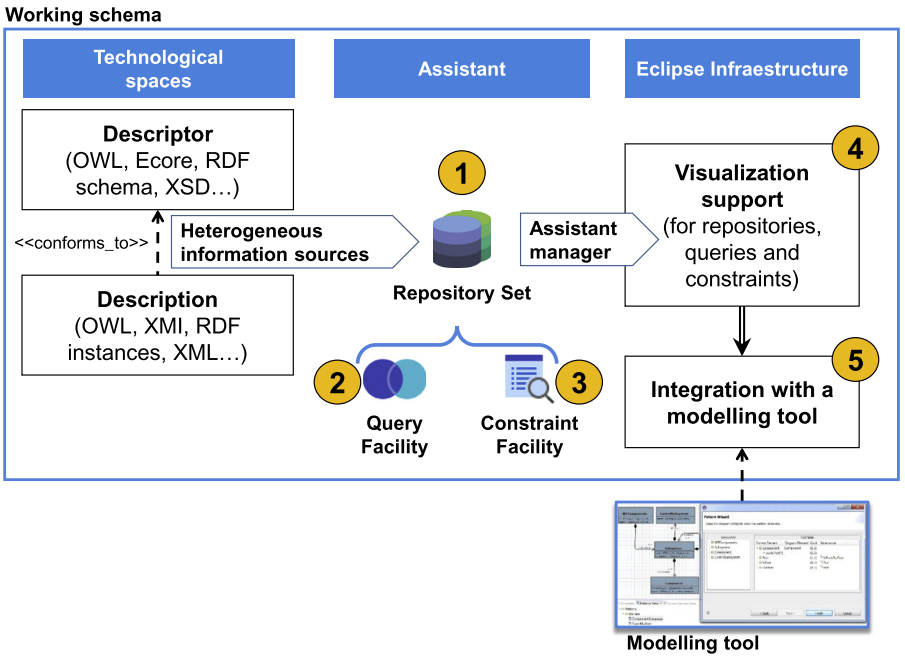


Fig. 1. Overview of our approach.

this scenario, the meta-modelling expert takes the role of the domain expert too, which may lead to mistakes or omissions, compromising the quality of the meta-model.

Meta-models within a domain are not completely different from each other, but they sometimes have recurring patterns and use common idioms to represent concepts [18,19]. For example, while building a language to describe behaviour, designers normally resort to accepted specification styles, including variants of languages such as state machines, workflow, rule-based or data-flow languages, enriched with domain-specific elements. The language designer can obtain this information from sources like meta-models, class diagrams, ontologies, XML schema definitions (XSD) or RDF documents. Moreover, having access to a variety of information sources helps in obtaining the necessary domain knowledge, vocabulary and technical terms required to build the meta-model. This situation also applies when building models, instances of a given meta-model. In this case, it may be helpful to have a way to query information sources and knowledge bases. These queries may use the data types present in the meta-model to help filtering the relevant information.

For this purpose, we have devised a modelling assistance approach, whose working scheme is shown in Fig. 1. The approach is useful for creating models at any meta-level. Our proposal is based on the creation of a set of repositories (label 1 in the Figure), in which heterogeneous data descriptions (OWL ontologies, Ecore meta-models, RDF Schemas, XML schema definitions), and data sources (OWL, RDF data, EMF models, XML documents) are injected.

Our system represents this heterogeneous data using a common data model, so that information sources can be stored in the repository in a uniform way. The system provides extensible facilities for the uniform and flexible query of the repository (label 2). We provide basic services for synonym search and word sense analysis, and a predefined catalogue of queries, which can be externally extended. The repository can also store heterogeneous constraints, and we support their evaluation using an extensible facility (label 3). The results of the queries for each source in the repository are aggregated and ranked according to their suitability and relevance. These query results and the information sources themselves can be visualized (label 4). Although the assistance system is independent of any modelling tool, it has been designed to be easy to integrate with a wide range of tools (label 5).

We have identified several scenarios where the assistant is useful. They can be generally classified in three areas. Firstly, for creating models and meta-models. Second, to create artefacts describing a set of other artefacts, like in model-based product lines [20,21] or model transformation reuse [22]. Finally, to evaluate quality aspects of a set of (perhaps heterogeneous) resources. More in detail, our approach is useful:

- As support for the development of new domain meta-models. This way, domain concepts and vocabulary can be sought in external sources such as XML documents or ontologies.
- To create models for a particular domain. In this case, model elements conforming to the same or similar meta-model can be incorporated into the model being built, and heterogeneous information sources can be queried in order to extract concrete data values.
- To design a “concept” meta-model [22]. A concept is a minimal meta-model that gathers the core primitives within a domain, e.g., for workflow languages. Furthermore, concepts can be used as the source meta-model of a model

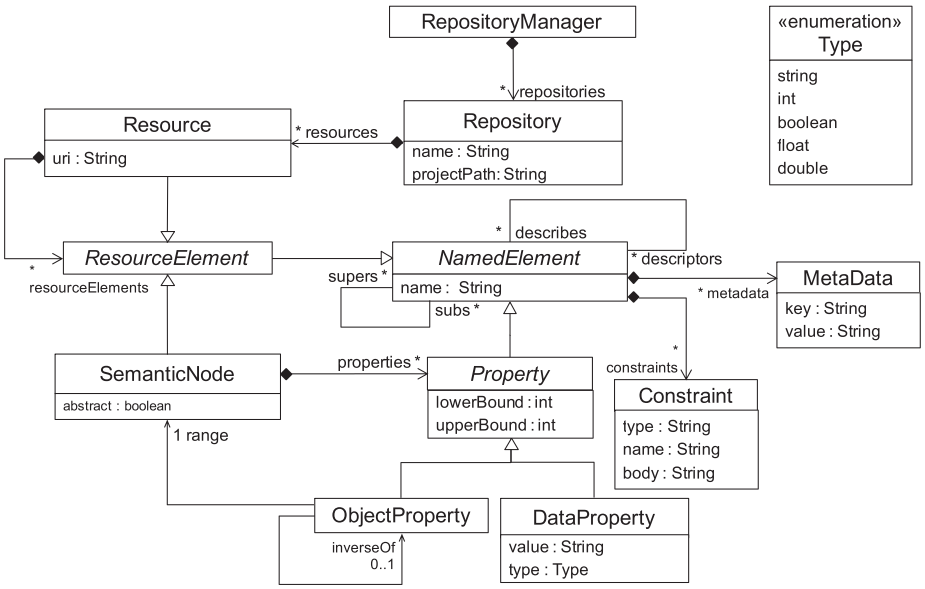


Fig. 2. The common data model (package dataModel).

transformation, becoming reusable, so they can be bound to a particular meta-model. This task implies the querying and understanding of a variety of meta-models for a particular domain and therefore the assistant becomes useful.

- To aggregate multiple existing models into a model-based product line [20,21]. In this approach, a description of the space of possible features of a software system is created, typically through a feature model. The choice of features implies the selection of a software model variant. This way, there is a need for understanding an existing family of models and to describe their variability. One possible approach is to merge or superimpose all model variants (leading to a so called 150% model) and use "negative variability", which selectively removes deselected artefacts [20,21].
- To detect "bad smells" [23] or signs of bad modelling practices [24] in a set of resources. This is possible, as we can perform queries over a repository to e.g., detect isolated nodes, or find abstract classes with no children. Moreover, our query mechanism is extensible, so that technology-specific or domain-specific queries can be created.

While our approach is useful in all these scenarios, to focus the paper, we concentrate on the modelling and meta-modelling scenarios.

3. The ingredients of a (meta-)modelling assistant

In this section, we detail the three main parts of our approach: (*i*) the handling of heterogeneous sources through a common data model (Section 3.1), (*ii*) the uniform support for queries (Section 3.2), and (*iii*) the managing of constraints (Section 3.3).

3.1. Handling heterogeneous sources

The first part of our approach adresses to the need for integrating several data sources stored in a variety of formats, providing a mechanism to organize and classify such resources. For this purpose, we rely on a common data model for storing the information of an arbitrary number of meta-levels. Heterogeneous sources, like XML or Ontologies can then be integrated by establishing transformations into our common data model (cf. Fig. 2). As we will see in Section 4, we have designed an extensible, component-based architecture that permits adding support for new sources – with so called format assistants – externally.

In detail, each file or information source is represented by a Resource, which can be aggregated into Repository objects. Each resource contains a collection of SemanticNode, i.e., entities that are added to account for different technical spaces [25]. In other words, semantic nodes are elements that gather knowledge from (original) source elements and hence, serve as an abstraction for managing heterogeneous information. Resources can be nested to account for hierarchical organization. For example, in MetaDepth and EMF, models and packages are nested, respectively.

Resources, nodes and properties are NamedElements that are identified by their name and can both act as descriptor, i.e., be a type or class, and be described by other elements, i.e., be instances of other elements. Further, our common data model can accommodate instance-relations found in heterogeneous technical spaces, which include (*i*) descriptor-only elements, such as, meta-classes in meta-models, (*ii*) described-only elements, such as objects in models,

Table 1
Mapping different representation technologies to the common data model

Common Data Model	EMF	OWL	XSD
Meta-level (types)			
Resource	Ecore file/EPackage	OWL file	XSD file
SemanticNode	EClass	OWL Class	xs:element
Property (abstract)	EStructuralFeature	rdfs:domain	xs:complexType xs:element
ObjectProperty	EReference	owl:ObjectProperty	Nested xs:element IDREF attribute
DataProperty	EAttribute	owl:DatatypeProperty	xs:attribute
Property.supers	EClass.eSuperTypes	Inverse of rdfs:subClassOf	xs:element type attribute
Constraint	OCL EAnnotation	N/A	xs:restriction
Model/Data level (instances)			
Resource	XMI file	OWL file	XML file
SemanticNode	EObject	Individual	XML element
ObjectProperty	Java reference	owl:ObjectProperty	Nested xs:element IDREF attribute
DataProperty	Java attribute	owl:DatatypeProperty	XML attribute

and (*iii*) elements that are descriptors of other elements and are described by others simultaneously, such as clabjects[2] as applied in multi-level modelling [26]. Hence, our data model is meta-level agnostic, as we represent with the same concepts both models and meta-models, classes and objects, and attributes and slots, leading to simplicity and generality [27]. Moreover, our data model can accommodate elements that are described by several elements. Thus, we can accommodate non-exclusive class membership of objects, such as found in Ontologies, and modelling approaches supporting multiple typing, like MetaDepth [28] or the SMOF OMG standard [29].

NamedElements can be associated with MetaData to account for technology-specific details that do not fit into our common data model. For example, when reading an EMF meta-model or a MetaDepth model, it may be necessary to store whether an object property represents a composition or the potency[3] of an element, respectively. Additionally, a Resource, which is also a NamedElement, can manifest conformance relations between artifacts, such as models and meta-models, or XML documents and XML schema descriptions (XSDs), and thus permits representing simple mega-models [30].

SemanticNodes can take part in generalization hierarchies (multiple inheritance is supported), and be tagged as *abstract*. Generalization hierarchies can be supported at any meta-level (i.e., not only at the class level), to account for approaches where inheritance can occur at the object level [31]. A node is made of a set of properties (DataProperty) and a set of links to other nodes (ObjectProperty), both defining cardinality intervals. Similar to nodes, properties unify the concept of *attribute*, i.e., a specification of required properties in instances, and *slot*, i.e., a holder for values. The common data model supports a range of neutral basic data types (Type enumeration), such as string, int, boolean and double. For generality, the value of the property is stored as a String. Finally, any element can have Constraints attached. The handling of heterogeneous constraints will be explained in Section 3.3.

Table 1 shows how several technologies can be mapped to our data model. We consider the modelling space (in particular the Eclipse Modelling Framework (EMF) [32], a widely used implementation of the Meta Object Facility (MOF) OMG standard [33]), ontologies and XSDs. In all three cases, we show how to map elements at different meta-levels into our common data model. Section 5.3 will assess the generality of our data model by means of an analytical evaluation.

EMF supports two meta-levels, and the mapping is direct. Fig. 3 shows a schema of how the translation from EMF into our data model is performed. The figure shows on the top that both meta-models (called *ecore* models) and models (typically serialized in XMI format) are transformed into Resources. In this case, the Resource object from the model is described by the Resource of the meta-model. The elements within both meta-models and models follow this translation scheme as well. Both EClasses and EObjects are transformed into SemanticNodes with a suitable descriptor relation, and similar for references and attributes. At the model level (in the compiled mode of EMF) links and slots are represented as Java fields, whose value is obtained via getter methods. Fig. 4 depicts the translation of XSDs, i.e., acting as language definitions and XML documents. This case is conceptually similar to EMF. The figure shows on the top that an schema, typically serialized in XSD format, is transformed into a Resource of our common data model. Then, as a result of XSDs being described in the XML format, the Resource object from the document is described by the Resource of the schema.

The elements within the schema and the document follow this translation as well. For example, an XML element is transformed into a SemanticNode with a descriptor relation to an xs:element. Moreover, an XML element or XML attribute is transformed either into an ObjectProperty or a DataProperty depending on the type it specifies. In particular, in case of xs:IDREF an ObjectProperty is created, while a DataProperty is created for any other type. Finally, in the case of On-

[2] A clabject is a model element that has both type and instance facets and hence holds both attributes, i.e., field types or classes, and slots, i.e., field values or instances.

[3] The potency of an element is represented by zero or a positive number that accounts for the instantiation-depth of an element at subsequent meta-levels [26].

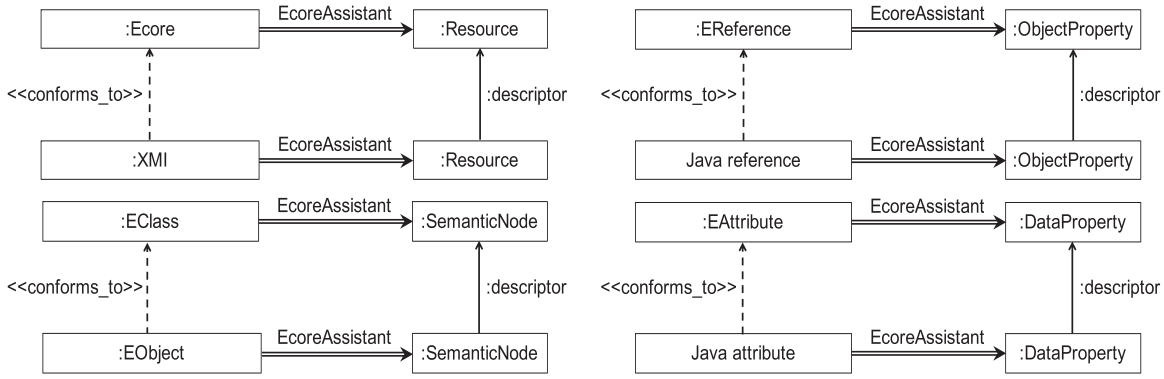


Fig. 3. Injecting EMF (meta-)models into the common data model.

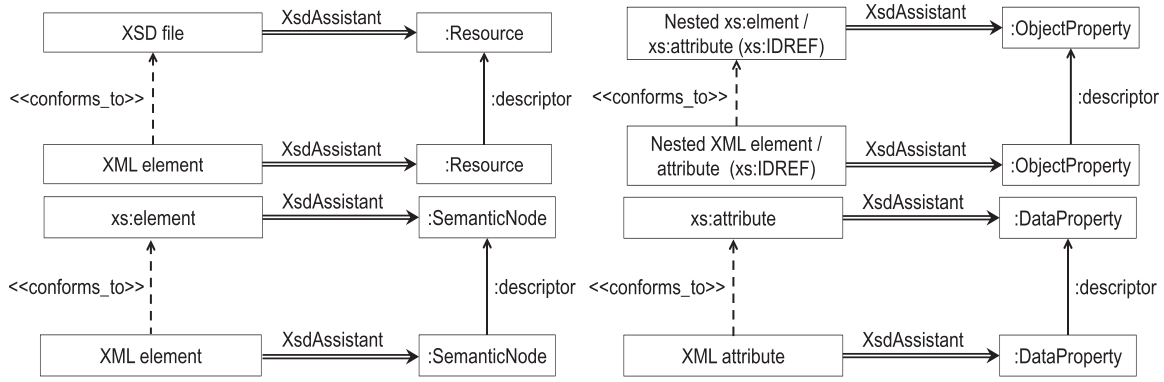


Fig. 4. Injecting XML schema descriptions into the common data model.

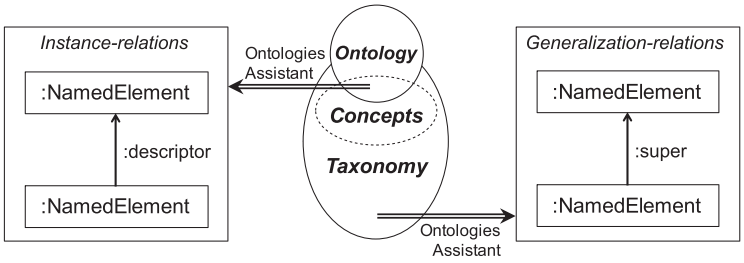


Fig. 5. Injecting ontologies into the common data model.

tologies there are no explicit meta-levels (Fig. 5). Then, classes may have several descriptors, and individuals (instances of a class) can have several classifiers, covering the different levels in the representation of concepts. Thus, SemanticNodes and Properties can take part in generalization hierarchies, all of them represented by the taxonomy. The technical realization of new format assistants will be explained in Section 4, while Section 5.3 will evaluate the generality of the data model.

3.2. Querying

Once the heterogeneous information is transformed into a common representation, it can be queried in a uniform way. As information conforming to many different schemas may be available in the data model, the query mechanism provided needs to support the flexible exploration of the information gathered. Moreover, oftentimes, the developer may only have a vague idea on what to search, hence a mechanism for inexact matching [34] is needed. This mechanism should be able to provide the user with not only exact matches, but with other related elements as well.

Thus, we refrained from using directly standard model query languages, like the Object Constraint Language (OCL) [35], because they would assume a precise, unique domain meta-model (while may need to query several models conformant to different domain meta-models), and rely on exact matches of queries. Moreover, queries would need to be re-expressed

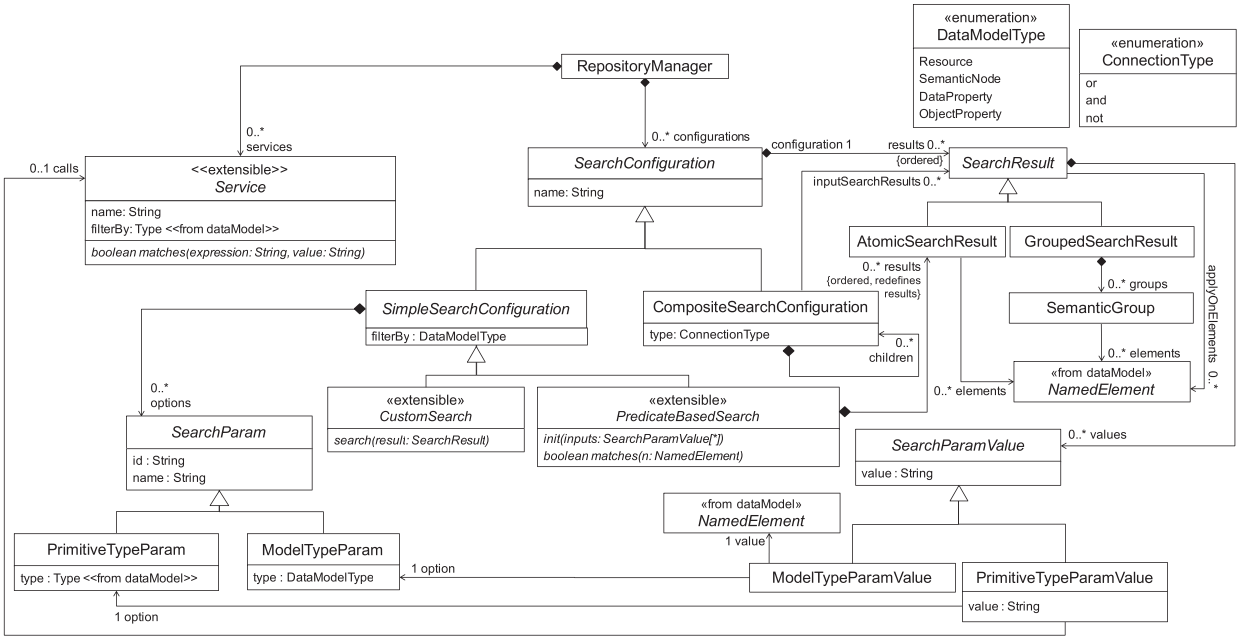


Fig. 6. Meta-model of our extensible query mechanism (package queries).

using the meta-model in Fig. 2, instead of in terms of the domain meta-models, which may lead to cumbersome expressions. Finally, we need our query mechanism to work at any meta-level.

Fig. 6 shows the meta-model describing our extensible query approach. The meta-model provides a mechanism for configuring queries (by specifying the necessary query inputs, class SearchParam), while the actual query results are reified using objects of type SearchResult. The results can be aggregated in groups (through class GroupedSearchResult), or returned atomically (class AtomicSearchResult).

Our approach supports two types of queries: atomic (SimpleSearchConfiguration) and composite (CompositeSearchConfiguration). Atomic queries are configured by declaring a set of SearchParams (representing the expected input parameters from the user), specifying the type of element it searches (filterBy attribute). Composite queries are formed by combining other (atomic or composite) queries, through the and, or and not logical connectives. Composite queries are issued by combining the result sets of previous (simple or composite) queries (reference CompositeSearchConfiguration.inputSearchResults), and may be nested (composition reference CompositeSearchConfiguration.children).

Atomic queries can follow two styles: predicate-based (class PredicateBasedSearch) or custom (class CustomSearch). In both cases, user-defined queries are expected to extend these two abstract classes. In practice, as we will show in Section 4, this is realized by profitting from Eclipse extension points. Their difference relies on how the iteration is performed: internally (driven by the engine), or externally (driven by the user code); and on the possibility of grouping the results. In practice, predicate-based queries are easier to specify, while custom queries permit more control on how outputs are presented.

Predicate-based queries select their result by providing a boolean predicate, evaluated on NamedElements. Their result is atomic (i.e., not divided in groups), made of all NamedElement objects satisfying the predicate. This kind of queries feature internal iteration. This way, our engine is responsible to traverse the repository and iteratively pass each object (of the type expected by filterBy) to the matches method. Alternatively, CustomSearch queries are responsible to both traverse the elements in the repository (external iteration) and select those elements fulfilling the query. This is a more flexible way of querying, which may construct results aggregated in groups.

When a custom query is to be executed, the search method receives a SearchResult object, which initially contains a set of SearchParamValue objects with the input parameter values (as entered by the user), and optionally a set of elements over which the query is to be executed (applyOnElements collection). If applyOnElements is empty, then the query is assumed to search through the whole repository. The input parameters of a search can either be of primitive data type (class PrimitiveTypeParamValue), or model elements of a given kind like Resource, SemanticNode, DataProperty or ObjectProperty (class ModelTypeParamValue). After the execution of the search method the SearchResult must point to the selected element. In the grouped elements output case (class GroupedSearchResult), the search method implementation must decide how to split the members from the elements collection by the definition of SemanticGroups.

Please note that different invocations to the search method results in different SearchResult objects, placed in the results ordered collection. This enables having a history of searches performed, which can be useful in explaning the provenance of the different elements in a model being built.

In a predicate-based query, we rely on a matches method that must evaluate if the list of elements from the input collection belongs to the output (elements collection) or not. In this case, results are saved as AtomicSearchResults. Before the iteration starts, the init method is invoked, receiving a collection of input values (objects of type SearchParamValue), corresponding to the query parameter values input by the user.

3.2.1. Query services

As seen in the meta-model of Fig. 6, queries can make use of Services through an extensible mechanism (extending meta-class Service). Services are functions for certain data types from our data model, such as strings or integers. Therefore, services can be used by a particular query on SearchParamValues of a given type. Next we describe some of the most prominent services we provide.

Inexact matching Searchs that rely on exact string matching are likely to provide poor results, as entities can be named using compound names ("ProcessModel"), or using different derivations (e.g., "processing", "processed"). This way, we provide a means for *inexact matching* [34] to increase the possibilities of finding useful information. The service is based on two techiques: (i) detection of compound names, and (ii) comparing words using their roots. This service is available on search inputs of type String (i.e., when the corresponding PrimitiveTypeParam is typed as String).

Regarding the first technique, it is quite common to find entities in meta-models or ontologies whose name is the concatenation of two or more words, most often in camel case. Our service considers the complete word, and also its different parts in the comparison. Regarding the second technique, comparing word pairs (e.g., "process", "procedure") might throw unsatisfying, or too general, results, even if they belong to the same word sense. For this reasone, the service uses the Porter stemmer algorithm [36], a method to reduce the comparison of two words to their lexical roots.

Word synonym expansion and ranking The exploration of a domain is a difficult task for two reasons: (*i*) in a new domain, with no experience background, a developer may lack the knowledge base about the vocabulary that defines the domain and (*ii*) in a known domain, a developer can lose sight of the main parts of the domain and, as a consequence, build a poor meta-model. Thus, it might be useful to increase the name-based searchs to consider synonyms relevant for the domain. However, words have synonyms with respect to different senses and therefore better search results are obtained by ruling out unrelated senses. For example, the word "glass" has different senses, e.g., to refer to "tumbler or drinking container" and to "lens to aid in vision".

Thus, we offer a service that, given a set of terms, expand them creating a list of synonyms, ranking them according to the relevance for the domain of the input terms. The service is inspired by the Lesk algorithm [37] and evaluates each term in a sentence assuming that every term tends to share a common semantic field with its siblings. For that purpose, we use Wordnet (a lexical database for the English language) [38] to select, among the available entities, the most suitable candidates to match the common semantic fields from an input term list. The list of candidates is obtained by the use of a rule-based system. In the system, we assign points to the whole list of senses provided by Wordnet while they are evaluated to discover whether they fit to the target domain or not.

The points are given by an strategy depending on which rule is more important during the evaluation. With the service we provide a set of strategies but new strategies can be added as well. Next, we present the rules for assigning those points and the strategies defined:

We consider the following entities to calculate the ranking:

U is the universal set of all possible terms, or words.

$T = \{t_1, \ldots, t_n\} \subseteq U$ is a set of input terms, or words, e.g., "process", "activity", "task".

$S = \{s_1, \ldots, s_m\}$ is a set of senses, for example the term "process" can refer to "set of actions" (e.g., s_1) or "a series of changes in the body" (e.g., s_2).

We consider phrases (denoted by p_i), made of sequences of terms. We write $t \in p_i$ to denote that term t occurs in the phrase p_i.

We assume the following linguistic functions and predicates, which can be provided by systems like Wordnet:

- Given a term $t \in U$, function $sense(t)$ returns the different senses of t. For example $sense(process) = \{s_1, s_2\}$.
- Given a sense $s \in S$, function $syn(s)$ returns a set of terms, which are synonyms with respect to the sense s.
- Given a term $t \in U$, function $defs(t)$ returns a set of definitions (a set of sentences) of term t.
- Given a term $t \in U$, function $exs(t) = \{p_1, \ldots, p_m \mid t \in p_i\}$ returns a set of example sentences, where each p_i contains the term t.
- Given two terms, predicate $syn(t, t')$ holds if both terms are synonyms, independently of the sense.
- Given two terms $deriv(t, t')$ holds if the terms have a linguistic derivative path.

Given a set of terms T, our system returns a set of terms $O = T \bigcup_{t \in T} \{u \in U \mid syn(t, u)\}$, made of the input terms plus their synonyms where the words in O are ranked according to a number of points, given by the following rules:

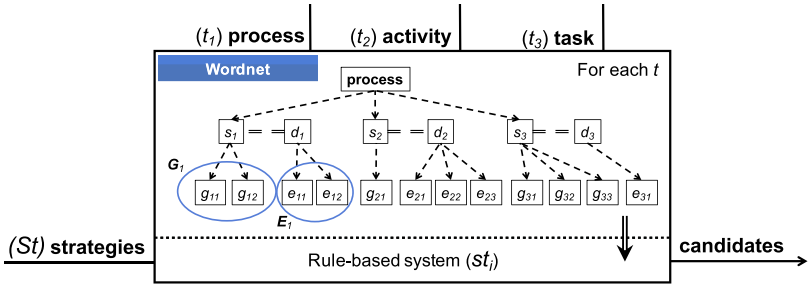


Fig. 7. Example execution of the synonym expansion and ranking service.

$\mathbf{R}_1$: Synonyms of input terms

$$\forall t, t' \in T \bullet syn(t, t') \Rightarrow points(t') := points(t') + p_1$$

$\mathbf{R}_2$: Synonyms of sense synonyms

$$\forall t \in T,\ \forall s \in sense(t),\ \forall g \in syn(s) \bullet$$
$$syn(t, g) \Rightarrow points(g) := points(g) + p_2$$

$\mathbf{R}_3$: Linguistic derivatives of sense synonyms

$$\forall t \in T,\ \forall s \in sense(t),\ \forall g \in syn(s) \bullet$$
$$deriv(t, g) \Rightarrow points(g) := points(g) + p_3$$

$\mathbf{R}_4$: Synonyms in definitions

$$\forall t \in T,\ \forall s \in sense(t),\ \forall g \in syn(s),\ \forall p \in defs(t) \bullet$$
$$g \in p \Rightarrow points(g) := points(g) + p_4$$

$\mathbf{R}_5$: Synonyms in examples

$$\forall t \in T,\ \forall s \in sense(t),\ \forall g \in syn(s),\ \forall p \in exs(t) \bullet$$
$$g \in p \Rightarrow points(g) := points(g) + p_5$$

Several strategies can be used to assign points to every rule:

- All the same: Each $\mathbf{R_i}$ receives the same quantity of points (p_i).
- Synonyms first: $\mathbf{R_1}$ and $\mathbf{R_2}$ receive more points than the rest of rules.
- Definitions first: $\mathbf{R_4}$ receives more points than the rest of rules.
- Examples first: $\mathbf{R_5}$ receives more points than the rest of rules.
- Custom: A distribution based on a custom criteria.

Example. Fig. 7 shows an execution schema for the service. Firstly, a set T of terms is received by the service. For each term (t_i), a tree of senses (s_{ij}), definitions (d_i), examples (e_{ij}) and synonyms (g_{ij}) is formed, using information from Wordnet. The set of rules are applied for each term according to a selected strategy (st_i). For example, for the list of terms $T = process, activity, task$ for a custom strategy that assigns 1000 points to the value p_1, 80 to p_2, 20 to p_3, 100 to p_4 and 20 to p_5, the following output list of ranked candidates is obtained:

1000 task
1000 process
1000 activity
320 job
320 chore
300 summons
260 body process
260 bodily process
260 bodily function
240 procedure
200 treat
200 physical process
200 outgrowth
200 appendage
180 natural process

180 natural action
100 work on
100 work
100 unconscious process
100 swear out
100 sue
100 serve
100 march
100 litigate
100 action
0 ~~undertaking~~
0 ~~project~~
0 ~~operation~~
0 ~~mental process~~
0 ~~labor~~
0 ~~cognitive process~~
0 ~~cognitive operation~~
0 ~~activeness~~

where the service discards the words with no points.

- **Numeric intervals** For numeric properties, instead of writing concrete values, it is possible to enter intervals (e.g., [1..5] or [2..*]) to achieve more flexible queries.

3.2.2. Examples

In this section we present two examples, illustrating the query extensibility mechanisms and the services provided. The list of all available queries is shown in Table A.1 in the appendix.

Fig. 8 depicts a predicate-based search that gathers the results atomically. The upper part of the figure represents an excerpt of the queries package shown in Fig. 6, which is extended with a query (SemanticNodeNameSearch) and the "Inexact Matching" service explained in Section 3.2.1.

The lower part of the figure shows an instance model containing the parameter of the search (object PrimitiveTypeParam), its input value (object PrimitiveTypeParamValue) and the search results (semantic nodes referenced from object searchResult). Overall, the input value "family" is received by the search, which is then passed to the service. The service checks whether the attribute value of the semantic node matches with the expression or not. Finally, the set of selected semantic nodes are attached to the AtomicSearchResult object.

Next, Fig. 9 shows a composite search made of the conjunction of the search result of two queries: the SemanticNodeNameSearch query and NumberOfChildrenSearch (which searches for nodes with more than a number of children through a generalization hierarchy, cf. Table A.1). The composite query returns SemanticNodes belonging to both search results, as schematically depicted by the Venn diagram in the rop-right corner of the lower model.

3.3. Handling constraints

In our data model, SemanticNodes are used to store the knowledge found in an element, including properties and links to other nodes. Additionally, they may include restrictions to be satisfied by their instances.

For example, in meta-models, a class may contain OCL invariants that all objects of such class should obey. Other modelling technologies, like MetaDepth allow attaching constraints to whole models [12]. Similarly, elements in XML schemas may contain restrictions on properties of a given data type. These may be used to define patterns on strings (regular expressions), restrictions on the string length (min/max/exact length), and handling of white spaces, among others. While OWL does not directly support integrity constraints, extensions have been proposed for this purpose [39,40].

Thus, our data model includes a facility to store and interpret heterogeneous constraints. Constraints can be attached to any NamedElement, which includes Resources, SemanticNodes and Properties, covering the main scenarios in the different technical spaces. Constraints have a type identifying the kind of constraint (e.g., OCL), a name, and a body.

In order to support evaluation of heterogeneous constraints (OCL, XML schema restrictions, etc) our approach is extensible. This way, constraint interpreters can be incorporated by extending class ConstraintInterpreter, declaring the constraint types they can handle, and implementing the evaluate method. The method receives a constraint and an instance of the element the constraint is attached to, and returns a boolean indicating if the element satisfies the constraint. As we will see in Section 4, the addition of constraint interpreters is done through an Eclipse extension point. Similar to query results,

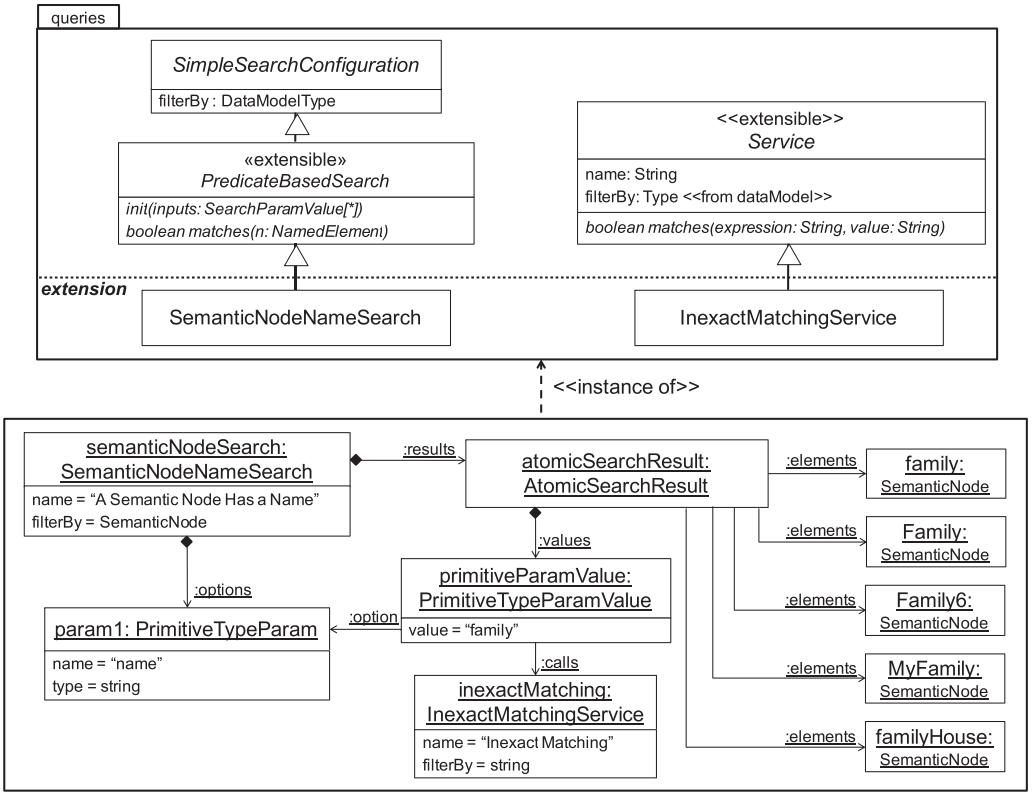


Fig. 8. An instance of a predicate-based search (PredicateBasedSearch) with an atomic result (AtomicSearchResult).

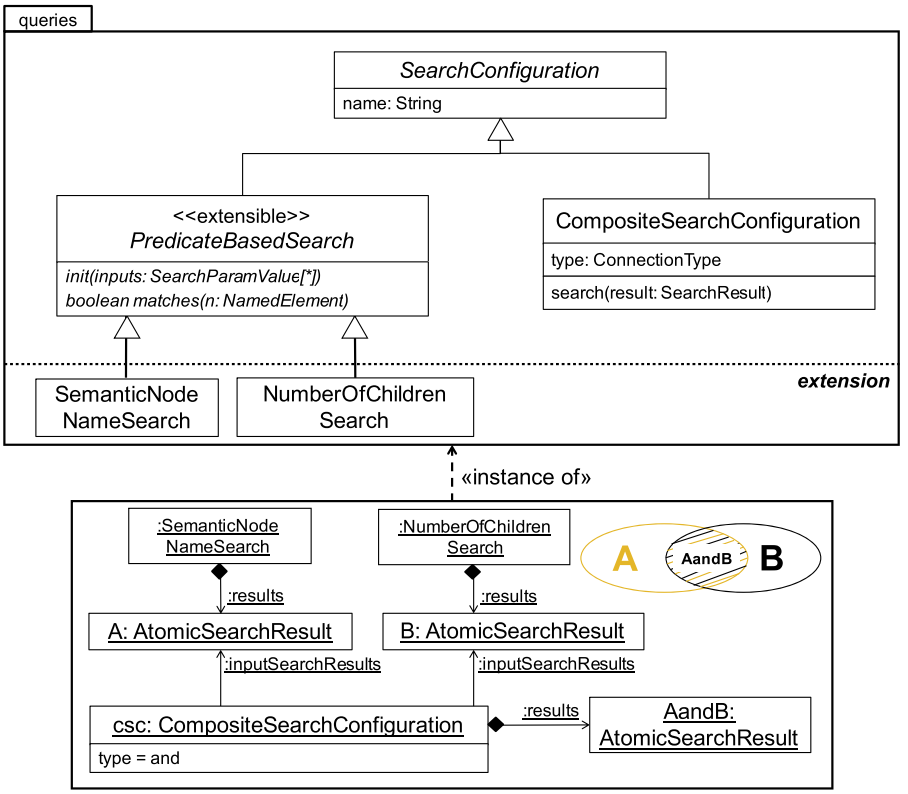


Fig. 9. An and-type composition (CompositeSearchConfiguration) of two searches.

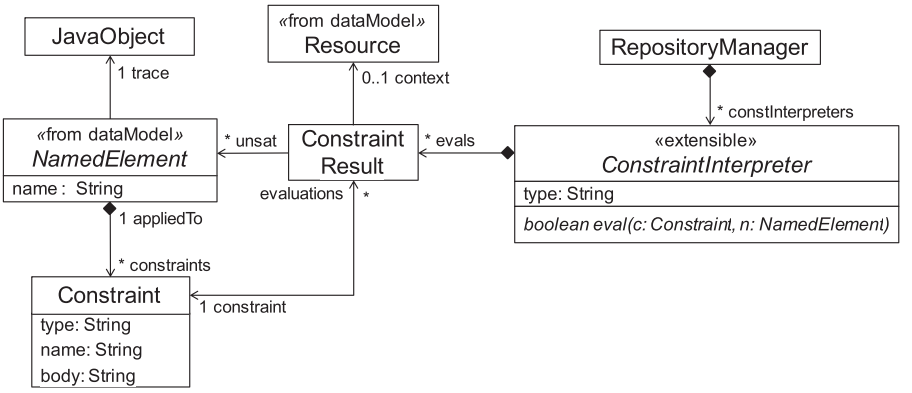


Fig. 10. Meta-model of the extensible constraint handling mechanism.

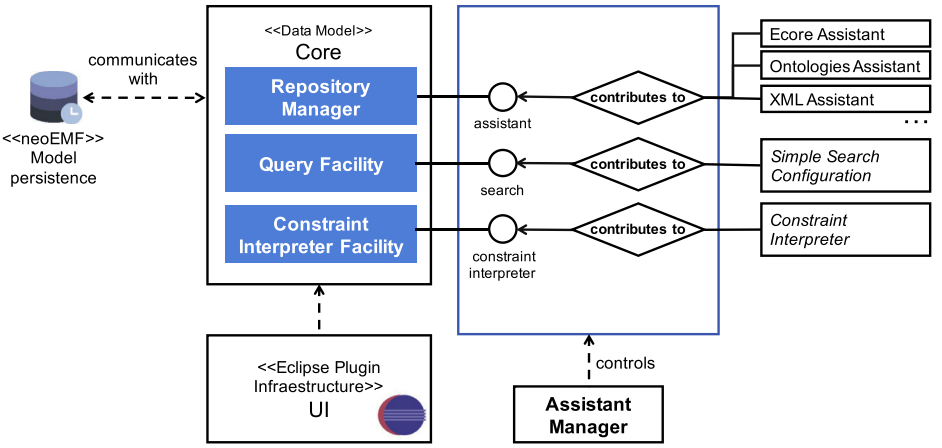


Fig. 11. Architecture of EXTREMO.

constraint results are reified using ConstraintResult objects, which hold elements that do not satisfy the constraint, and in addition organizes results in the context of the enclosing Resource.

4. Architecture and tool support

We have realized the previously presented concepts in a tool called EXTREMO. It has been implemented as an Eclipse plugin, is open source, and is freely available at http://miso.es/tools/extremo.html. The web page includes videos, screenshots and installation details. A schema of EXTREMO's architecture is shown in Fig. 11.

EXTREMO is made of a Core component, which provides support for the common data model and includes subcomponents for the query and constraint handling facilities. The Core component can be extended in different ways, e.g., to provide access to heterogeneous information sources, as shown in Fig. 11. This extensibility is realized through Eclipse extension points. These are interfaces that can be implemented by external components to provide customized functionality.

To allow for scalability, the repository is persisted using NeoEMF [41], a model persistence solution designed to store models in NoSQL datastores. NeoEMF is based on a lazy-loading mechanism that transparently brings into memory model elements only when they are accessed, and removes them when they are no longer needed.

The UI component permits visualization and interaction with the resources and query results. A set of views, commands, wizards and actions has been defined to control the access to the repository, the list of query results and the constraint validation facilities. By extending this component, it is possible to integrate EXTREMO with external modelling tools.

Next, we will detail the structure of the Core subsystem in Section 4.1, while the UI subsystem will be described in Section 4.2.

4.1. The core subsystem

Fig. 12 shows the main components of the Core subsystem: (*i*) a Repository Manager, which controls the access to the common data model and assists in the process of incorporating a new technological space; (*ii*) a Query Facility, which

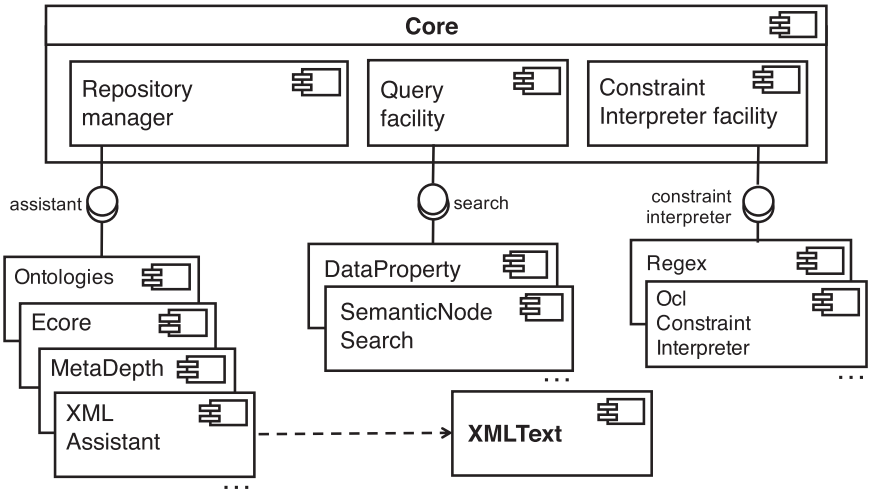


Fig. 12. Architecture of the Core component.

supports our query mechanisms in an extensible way; and (*iii*) a Constraint Interpreter Facility, which provides flexible support for different constraint formats and interpreters.

For each part, a set of extension points have been defined, and the figure shows some example implementations of these. The first extension point (assistant) permits adding support for new data formats. It requires implementing a mapping from the format-specific structure, such as XML, to the common data model, as described in Section 3.1.

We predefined a set of assistants as well as a framework for their creation, which permits their conceptual organization as model-to-model transformations. Hence, we provide an abstract class with a number of methods, which need to be overriden for the particular assistant, and act as rules in a transformation. In these methods, it is possible to create one or more elements in the common data model, hence supporting one-to-many and many-to-one mappings.

To facilitate the construction of new assistants, it is possible to define class hierarchies, and hence reuse import functionality. In addition, our scheme can profit from existing transformations between two specific technical spaces. For example, if one had a transformation from XSD to Ecore, and an assistant for Ecore (translating Ecore to Extremo's common data model), then an assistant for XSD can be built by first translating into Ecore and then invoking the Ecore assistant. This is the way we implemented the XsdAssistant (see Fig. 4) [42–44].

The second and third extension points provide extensibility for queries. Conceptually, user defined queries extend the meta-model of Fig. 6. The extensions allow defining custom, predicate-based, and composite queries by subclassing from the corresponding classes in Fig. 6. Finally, the last extension point permits contributing support for evaluating new types of constraints, extending the meta-model in Fig. 10.

4.2. The UI subsystem

Fig. 13 shows the architecture of the UI subsystem. It is made of contributions to the Eclipse infrastructure to visualize and interact with the repository, to visualize the search results and the list of elements that satisfy a constraint.

The UI subsystem is composed of: (*i*) a Search wizard dialog, that receives the list of search configurations; (*ii*) a Resource Explorer, a mechanism based on Zest (see https://www.eclipse.org/gef/zest/), a component for graph-based visualization, which provides support for filtering and layouts; and (*iii*) a set of view parts, that reflect the current state of the repository set model, the query results (as instances of the class SearchResult) and the constraints validation. All of them can be sorted or filtered by means of an extension point.

As an example, Fig. 14 shows Extremo in action. In particular, it shows the query dialog by which any of the defined queries can be selected, input values can be given to each of its input parameters, and services can be selected depending on the types of the parameters. In the lower part, the figure shows the repository view, with some resources and their content; and the search result view, which permits browsing through the query results. It must be noted that semantic nodes in the repository view have distinct icons, depending on whether they contain data, object properties or constraints, and on whether they are types, instances or both.

Fig. 15 shows the resource explorer. In particular, it shows on the right an instance of our common data model and the relationships between nodes. Since the resource explorer is based on a component for graph-based visualization, the SemanticNode instances are represented as nodes and the ObjectProperty instances are represented as edges of the graph. The left part shows how the resource explorer can be invoked from every resource with a action contribution to the pop-up contextual menu of the repository view.

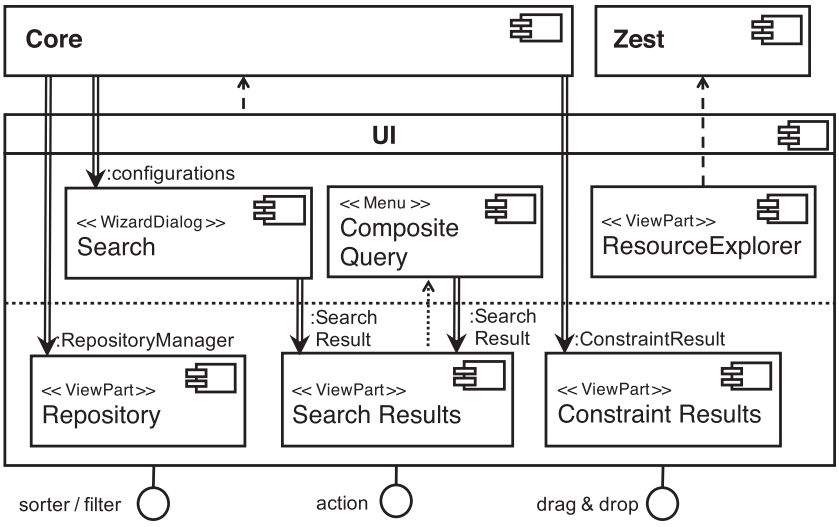


Fig. 13. Architecture of the UI component.

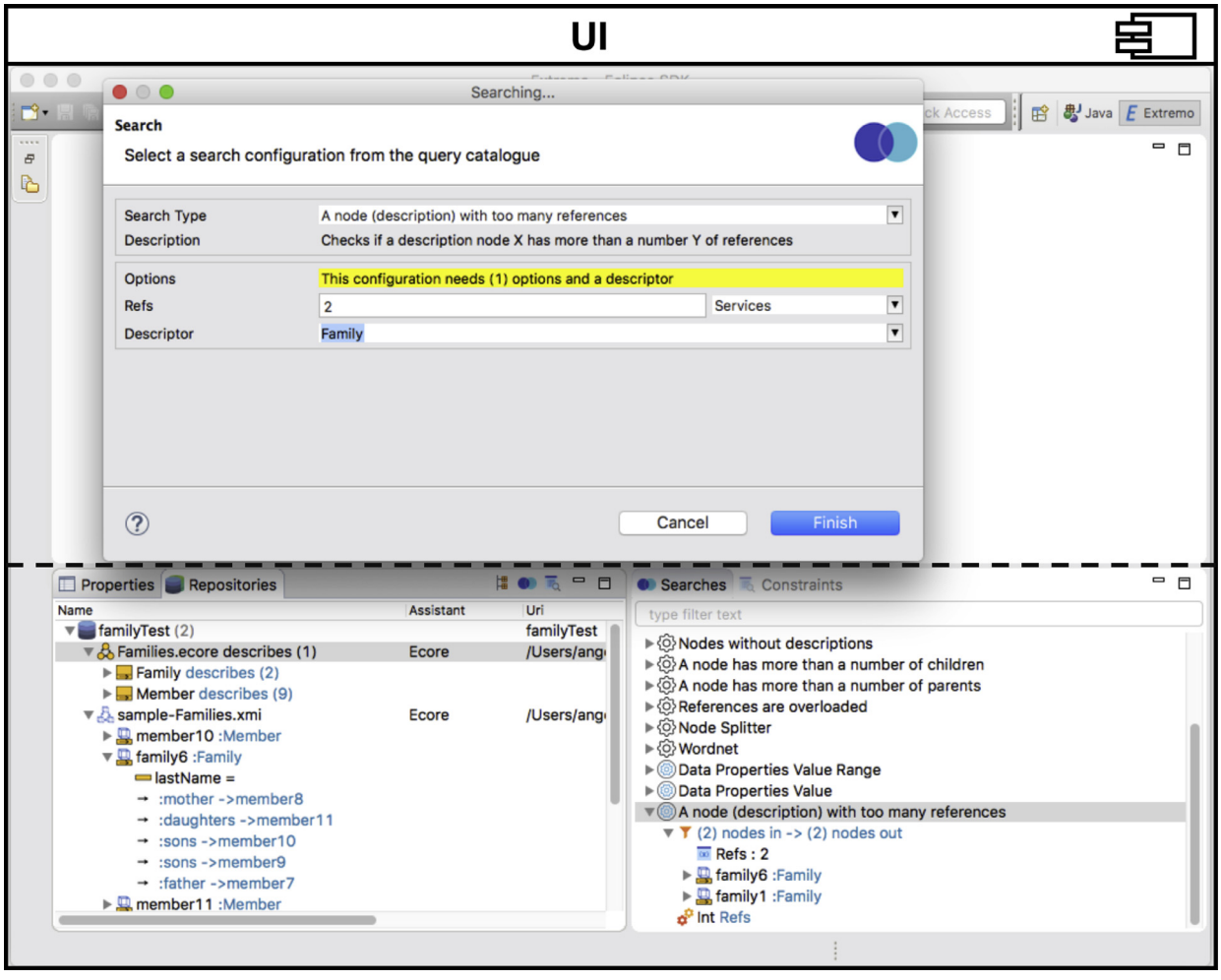


Fig. 14. Extremo in Action: Search Wizard Dialog and views.

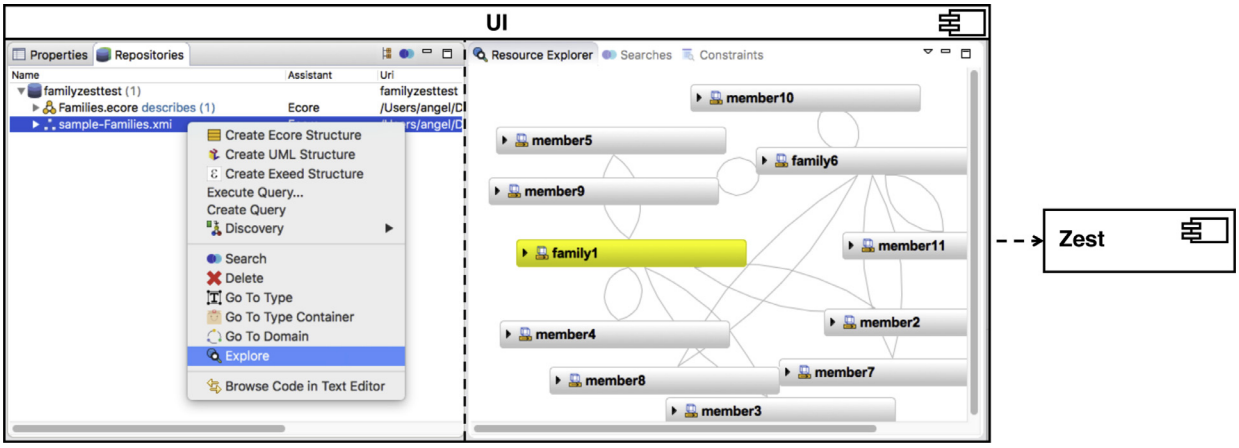


Fig. 15. EXTREMO in Action: Resource Explorer.

The UI subsystem has been designed so that it can be flexibly integrated with external modelling tools. For that purpose, two extension points have been defined. The first one enables the addition of an action to the view parts, as a reference in the contextual menu and the toolbar. The other extension point enables the drag operation from the views and the dropping of the selected information into any graphical editor based on Graphical Editing Framework [45]. This is the underlying framework of Eclipse graphical editors. We will explain these two integration styles in the next two sections.

4.2.1. Integration with a modelling tool by means of actions

In this section, we present the integration of the core components of EXTREMO with a modelling tool using an action contribution to the view parts of the UI subsystem. We will illustrate this section with the UML model editor,[4] a solution based on a TreeEditor. This kind of editor is an extensible part of the Eclipse infraestructure.

Fig. 16 illustrates the working scheme of this integration style, where an instance of our data model (Fig. 2) selected by the user (e.g., via a query) is incorporated into a UML model [4]. The elements selected by the user (label 1) are two non-abstract SemanticNode objects. The first one contains a DataProperty (d1, typed as string) and an ObjectProperty that refers to the second SemanticNode. The addition of a new action contribution is enabled by an extension point. The action contribution provides the mapping between our common data model and the intrinsic model of the TreeEditor. The action contribution must extend the ExtensibleViewPartActionContribution class (label 2) and implement an abstract method that creates the instances of the target model, conceptually working as a model-to-model transformation (label 3). Finally, a UML model is created (label 4). The initial two SemanticNode objects are mapped into two non-abstract Class objects, the initial DataProperty object is mapped into a Property object and the initial ObjectProperty object is mapped into an Association object.

Fig. 17 shows how this is achieved in practice. The repository view of EXTREMO is shown in label 1. In this view, the user can select a number of elements that are to be incorporated into the UML model (editor shown in label 3). For this purpose, a contextual menu offers the different action contributions (label 2). The UML editor in the figure shows already the Ecore elements incorporated into the UML model.

4.2.2. Integration with a modelling tool by means of drag and drop

The second integration style enables the addition of a drag operation from the views and the dropping of a set of NamedElements into a graphical editor based on Graphical Editing Framework [45], the underlying framework of Eclipse graphical editors.

We will illustrate this section with DSL-tao [19], an Eclipse plugin for the construction of DSLs using a pattern-based approach. The underlying representation of DSL-tao is Ecore. Thus, in this case, Fig. 18 shows an example that transfers an instance of our data model (Fig. 2) into an instance of Ecore.

As in the previous section, the portion of the model selected by the user (label 1) contains two non-abstract SemanticNode objects. The first one contains a DataProperty (d1, typed as string) and an ObjectProperty that refers to the second SemanticNode. The addition of a new dropping contribution is enabled by an extension point. The dropping contribution provides the transfer of a set of NamedElements selected from the views and the catching of the drag and drop event. The dropping contribution must extend the IExtensibleGEFDnDContribution interface (label 2) and implement a method resolving the graphical viewer editor that receives the selected elements (label 3). Finally, an Ecore model is created (label 4). The initial two SemanticNode objects are mapped into two EClass objects, the initial DataProperty object is mapped into an EAttribute object and the initial ObjectProperty object is mapped into an EReference object.

[4] UML2-MDT, http://www.eclipse.org/modeling/mdt.

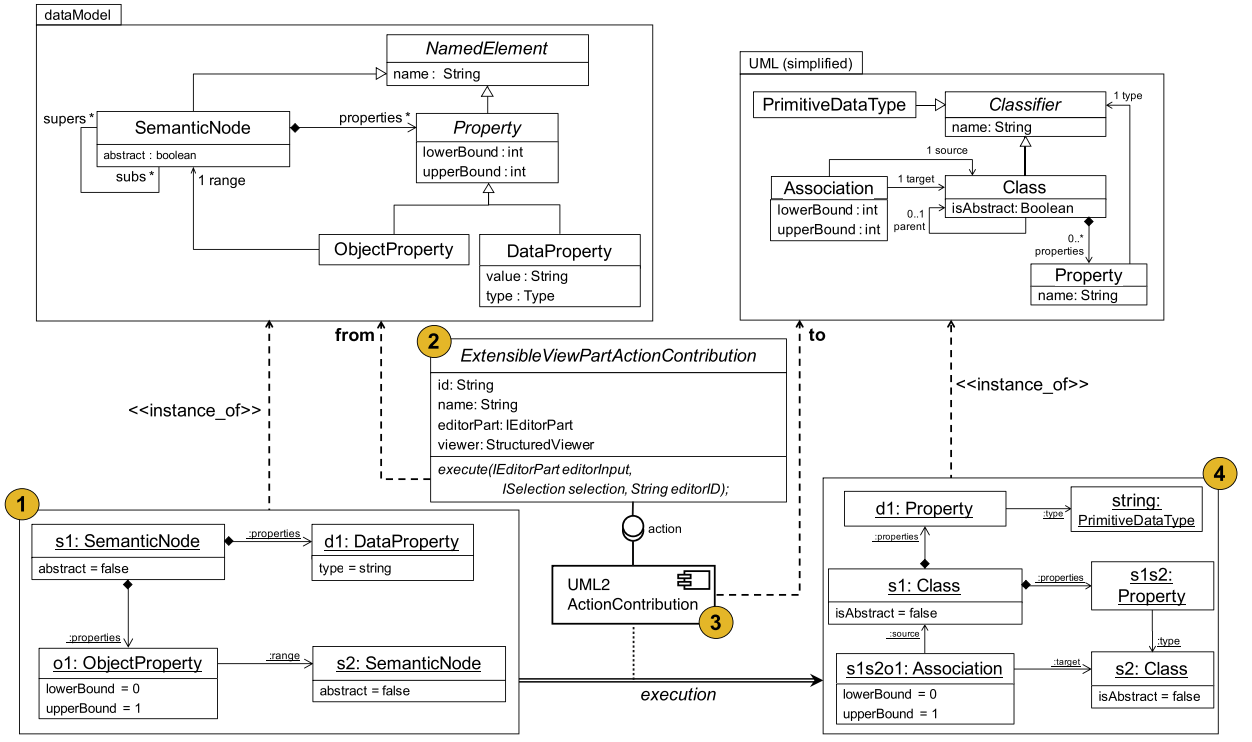


Fig. 16. Example: action-based integration.

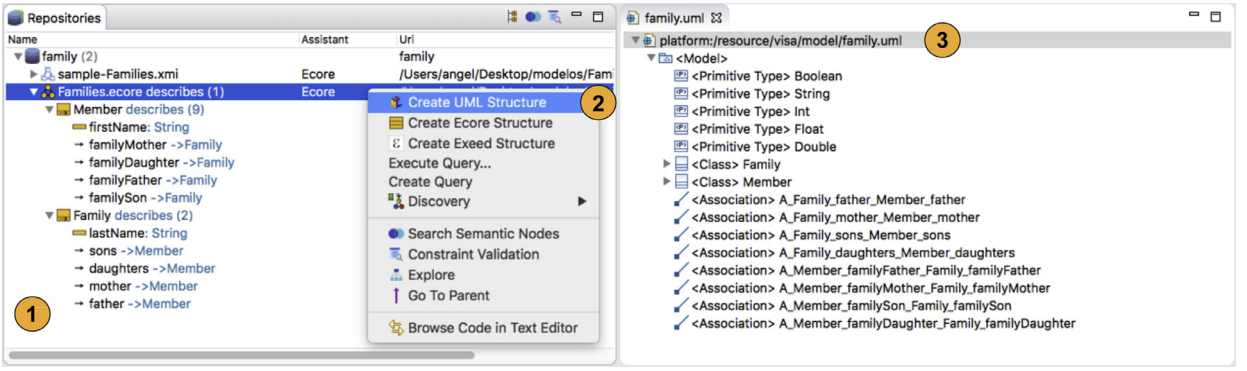


Fig. 17. Integrating Extremo with the UML2 modelling tool using an action contribution.

Fig. 19 shows how this is achieved in practice. The Figure shows the main canvas of DSL-tao, which permits building graphically a meta-model using the palette to the right. The bottom of the figure shows the views contributed by Extremo. In this case, the user may initiate a drag from some model element in the view and drop it into the canvas (labels 2 and 3). In the Figure, the user has incorporated class Member into the meta-model, while it is also possible to incorporate attributes and references.

5. Evaluation

We present an evaluation of our approach under different perspectives. We start by describing the research questions we address in the evaluation in Section 5.1, followed by three different evaluations in Sections 5.2–5.4, and finishing with a discussion of threats to validity in Section 5.5.

5.1. Research questions

By conducting this evaluation, we aim at solving the following research questions:

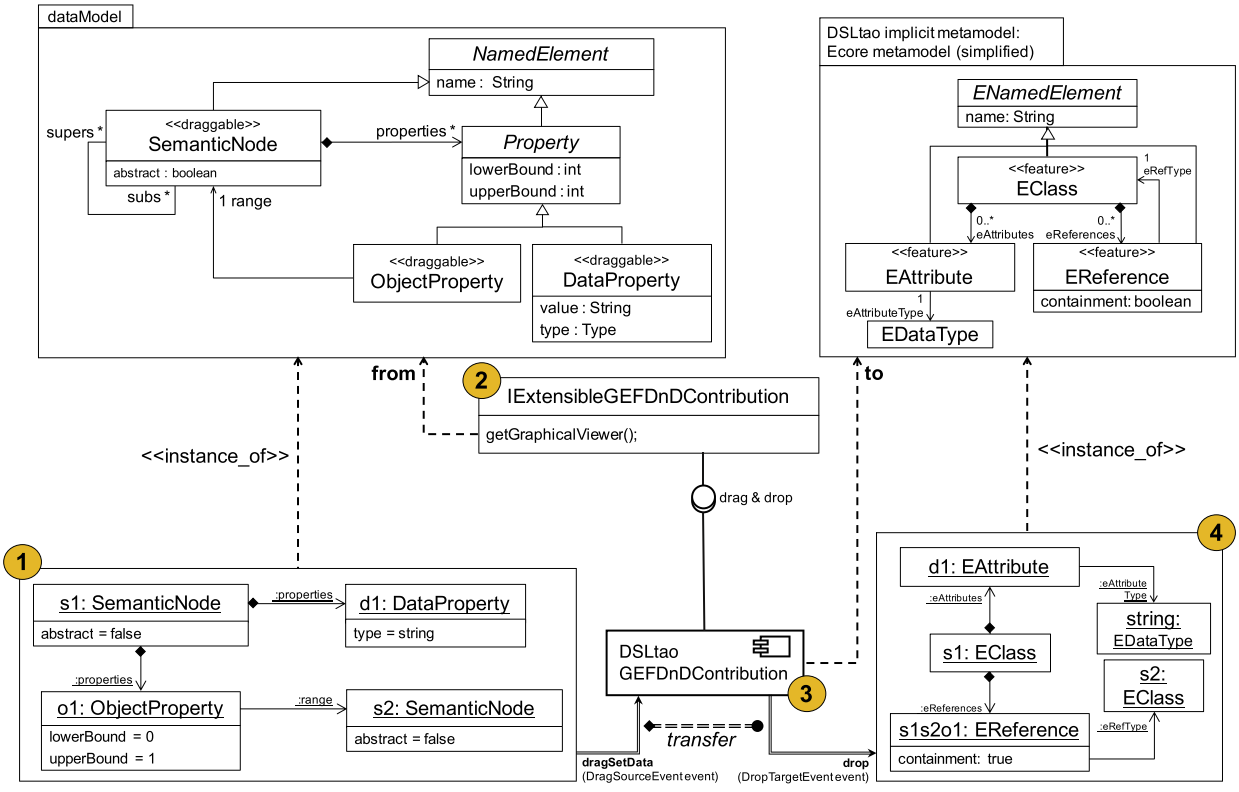


Fig. 18. Example: drag and drop based integration.

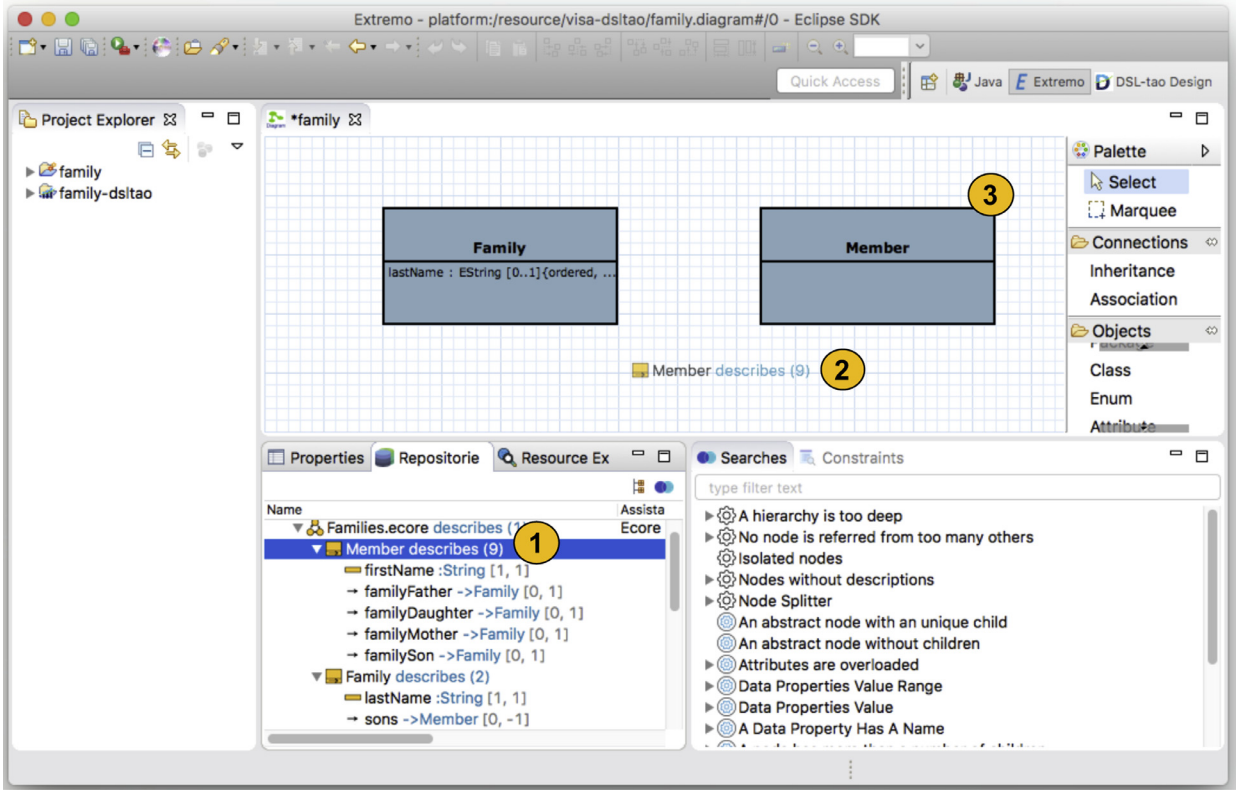


Fig. 19. Integrating Extremo with DSL-tao using a drag and drop contribution.

RQ1: How useful is EXTREMO *to solve practical problems in Language Engineering?.* In order to assess the usefulness of EXTREMO, we will use two demonstration cases [46]. In both of them, EXTREMO was used to support the development of complete DSL implementations for process modelling in the immigration domain (Section 5.2.1) and modelling of production systems (Section 5.2.2).

RQ2: How capable is EXTREMO *to represent information coming from different technological spaces?.* As previously described, the range of formats currently supported by EXTREMO can be extended. This way, Section 5.3 presents an analytical evaluation to assess how well EXTREMO's data model covers the space of possible information description approaches.

RQ3: How integrable is EXTREMO*?.* One of the salient features in EXTREMO is the possibility of integrating it with third-party tools. Having already successfully integrated a number of tools with EXTREMO, we intend to assess the reach and limitations of the integration mechanism. This question is approached in Section 5.4.

5.2. Evaluating usefulness

In this section, we evaluate usefulness by presenting two demonstration cases. The first one is on process modelling in the immigration domain (Section 5.2.1), while the second one is on modelling of production systems (Section 5.2.2). In the cases, EXTREMO was used in combination with two different modelling tools (DSL-tao and the Ecore tree editor), while assistants for four different technologies were used (Ecore, Ontology, CSV and XML). The purpose of the cases is evaluating the power of combining heterogeneous information sources for language engineering.

Characteristics of the two selected cases are (*i*) they cover both structural and behavioral modelling languages, (*ii*) they combine information from standardized modelling languages and from very focused domain-specific languages, and (*iii*) modelling languages are integrated with small to large domain descriptions coming from different technological spaces. They differ in the variety of technical spaces and the size of resources being considered.

For both cases we will highlight the use of our framework over a sequence of phases, which include (*i*) the scope of the language and an example of model, (*ii*) the collection of resources required, (*iii*) the resource import and querying of the common data model, (*iv*) the construction of a language meta-model and creation of instances, and (*v*) the results obtained.

5.2.1. Immigration process modelling

In the first demonstration case, we present the construction of a DSL for describing administrative e-Government processes for immigration.

Scope of the language. We will construct a language for the modelling of processes to issue visas according to the American regulation[5], called IWML. The language will include elements showing the variety of offices, concepts from the domain of immigration and agents involved in the process and generic elements of workflow languages like BPMN [47].

Example model. Fig. 20 shows an example IWML model, which contains (*i*) domain-specific knowledge related to the agents (DOS, DHS, Medical Service, User), and the artefacts (the I-130 form, the variety of Visas and the Payment method) involved in the process, (*ii*) specific activities needed in immigration processes, such as ≪Application Process≫, ≪Date Selection≫, or ≪Interview≫, (*iii*) more generic elements – such as tasks, events, and gateways – typically found in process modelling languages. In the figure, a User is required to fill in a standard I-130 form and to set a date for an interview with the embassy personnel. Then, the user is interviewed by the US Department of Home Security (DHS) and they perform a study of the application submitted. Afterwards, the Department of State (DOS) validate the data. In a parallel process, the candidate is expected to go through a Medical Examination. Once both the ordinary verifications and the medical report have been checked, the document is dispatched. Depending on the characteristics of the applicant various types of visa should be issued. In this case, a E1 type Visa is approved and consequently issued; otherwise, the document is denied.

Resource collection. In order to gather all the required elements we defined a set of repositories and resources according to the data model shown in Fig. 2. A list of resources were taken from: (*i*) the Object Management Group (OMG)[6] (http://www.omg.org/spec/); (*ii*) Ecore repositories, such as the ATL zoo (http://web.emn.fr/x-info/atlanmod/), which includes meta-models for BPEL, DoDAF, Gantt, ODP, SPEM and some others; and (*iii*) Ecore files available on the OMG repository with standard meta-models, such as BMM, BPMN, CMMN, DMN, SBVR and SPEM. For the domain-specific concepts, open resources were required. In our case, we took CSVs from the U.S. Open Data Portal (https://www.data.gov/) and US e-Government domain ontologies (http://oegov.org/), which are available in OWL and RDF formats. This second repository contains ontologies that model the set of United States government facets. These include the Government ontology, the U.S. Government ontology, the Department of Homeland Security ontology and Department of State ontology.

Table 2 summarizes the number of instances of the resource collection by focusing on the meta-level and, in the case of the ontologies, taking into account also the individuals. Even though in this case the resources imported were not

[5] https://www.usa.gov/visas.

[6] The OMG is the standarization body behind many modelling standards such as UML, SysML, MOF or BPMN.

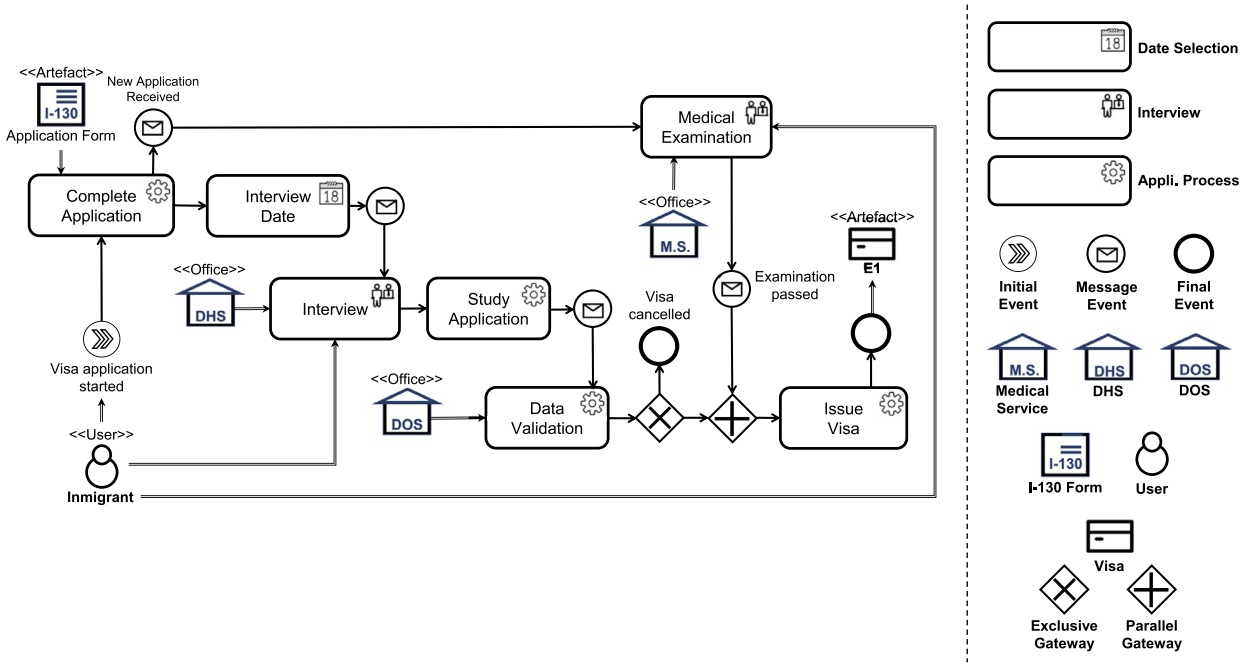


Fig. 20. Example process for issuing a visa with IWML (left) and legend (right).

Table 2
Number of collected instances of different Ontology and EMF-concepts, respectively.

Ontology concept	Number of instances	EMF concept	Number of instances
OWL file	22	Ecore file/EPackage	30
OWL Class	141	EClass/EDataType	2484
Individuals	1118	EReference	892
owl:ObjectProperty	35	EAttribute	326
owl:DatatypeProperty	91	OCL EAnnotation	0

large (this will be evaluated in the second case study) we can appreciate that Extremo was able to gather information coming from different technological spaces. The total size of the reused artefacts amounted to around 2.4 MB of data.

Resource import. We imported the Ecore meta-models taken from the OMG and the ATL zoo by the application of our EcoreAssistant (cf. Section 3.1) and the domain-specific concepts by applying our OntologyAssistant and the CsvAssistant.

Meta-model Construction. The meta-model was developed using DSL-tao, integrated with Extremo following the approach described in Section 4.2.2 by means of a drag and drop extension point. Fig. 21 shows a moment in the construction of the meta-model. In particular, it shows the Eclipse IDE with the Extremo perspective open (label 1), which includes the *Resource Collection* previously mentioned in the repository view (label 2) and the BPMN.ecore resource visualized in the resource explorer (label 3).

The meta-model construction phase involved some of the queries listed in Table A.1, such as finding types of forms, organizations and gateways. Once a query has been issued, the resulting elements can be highlighted on the original resource (label 4). Finally, a set of semantic nodes can be dropped over the DSL-tao design diagram (label 5).

Result. Fig. 22 shows the final result obtained. In detail, IWML has elements originating from the set of meta-models that describe workflows, such as Gateways, Events or Task. Moreover, IWML is composed of some domain-specific concepts taken from the set of domain ontologies, like DOS, DHS or MedicalService; CSVs, such as the type of Visas or the set of Users. Roughly 22% of the classes in our solution have been obtained from different ontologies and CSVs, 48% of the classes have been obtained by combining different representations of process modelling meta-models and the rest (30%) have been added by hand taking information from the government websites. This suggests that Extremo is useful as a help in the construction of a complex meta-model, as we were able to build the meta-model by reusing elements from different heterogeneous information sources.

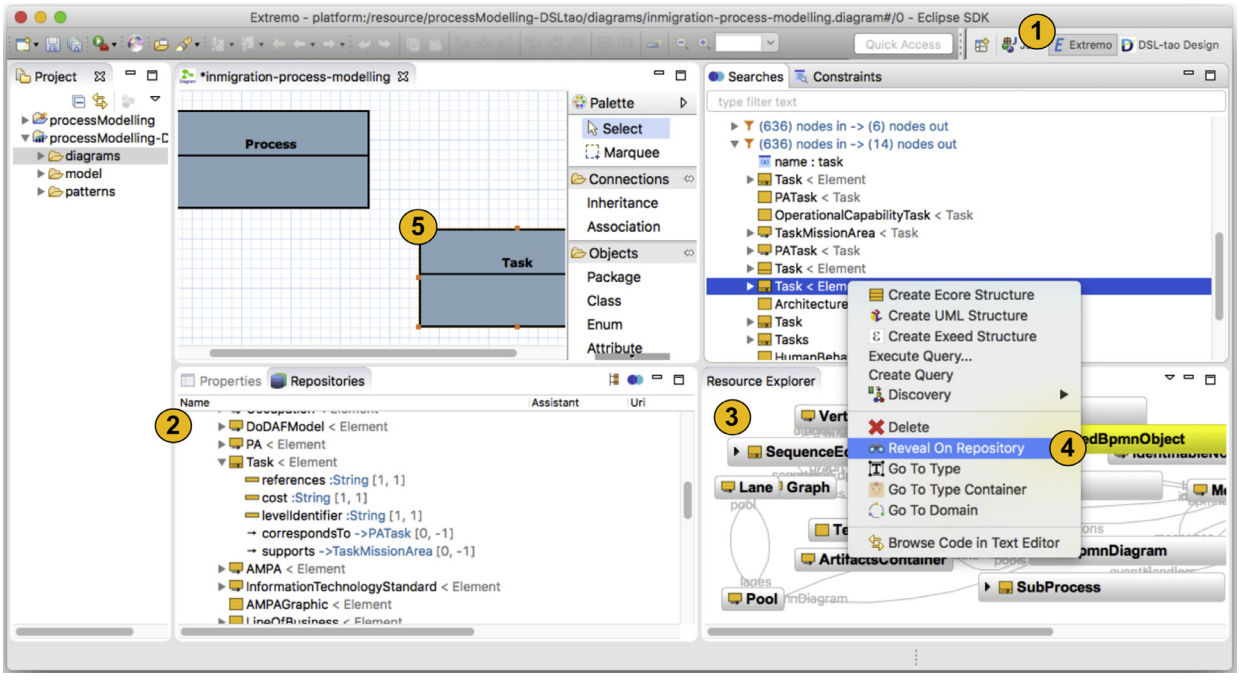


Fig. 21. Integration of EXTREMO with DSL-tao (from the perspective of the case study).

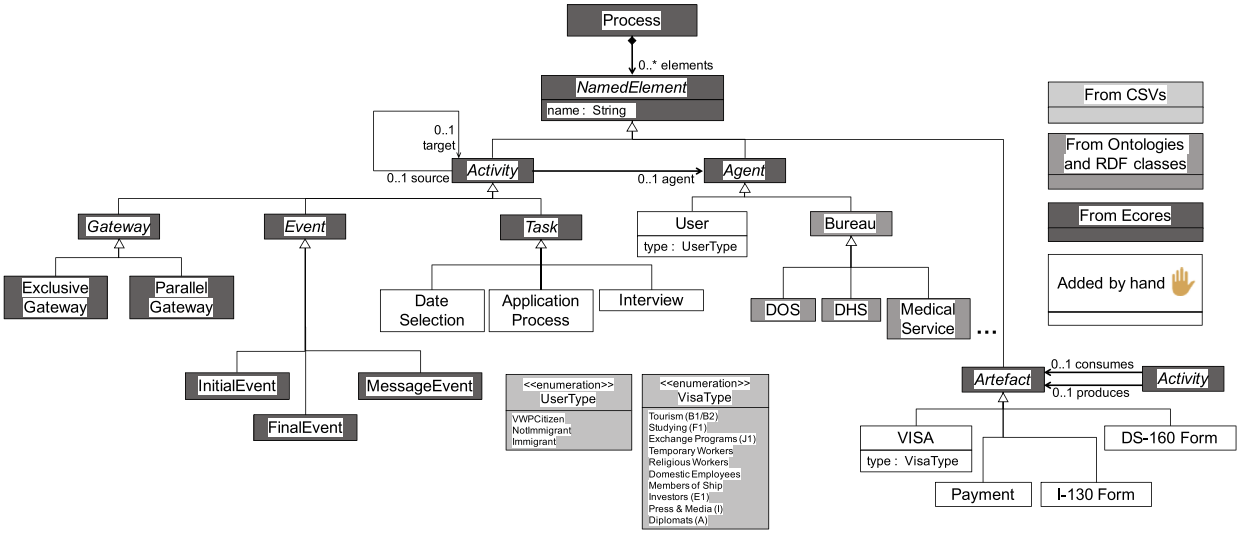


Fig. 22. Excerpt of the processes for immigration meta-model.

5.2.2. Industrial production system modelling

In the second case, we present the development of a DSL for industrial production systems to enable the construction of models that conform to an interoperable cross-industry standard for products and services called eCl@ss.[7] The goal of this case study is to reduce the number of potential candidates for conveyor-belt system-components that are available in the eCl@ss-standard, and thus to conveyor-belt system-modellers, by applying EXTREMO for constructing the desired language.

Scope of the language. We construct a language for the modelling of production systems conforming to the eCl@ss-standard called EPML. The language includes elements from conveyor-belt systems that must fulfill the constraints

[7] An ISO/IEC-compliant international standard for the unified and consistent definition and classification of products, materials, and services alongside typical supply chains by the use of commodity codes.

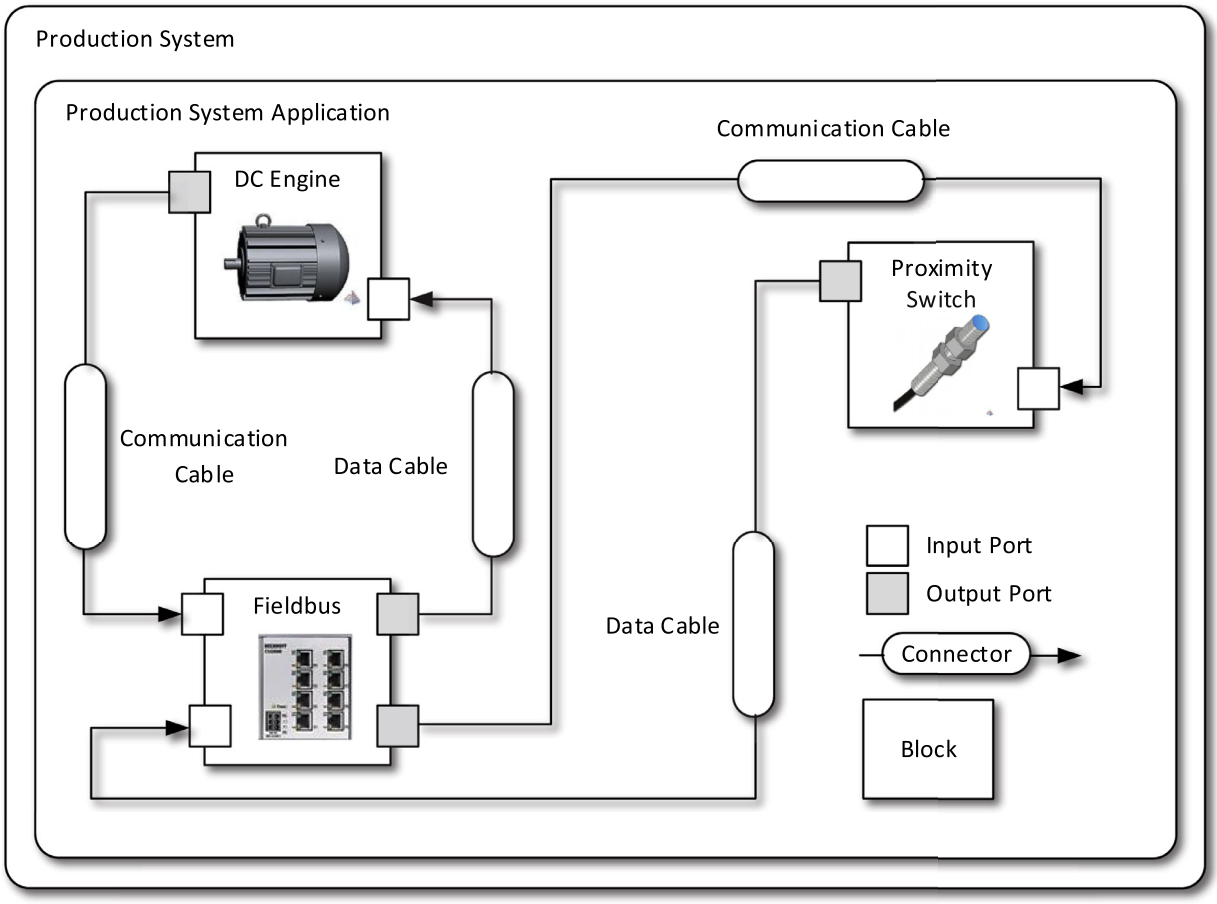


Fig. 23. Abstract graphical representation of the conveyor-belt production system model.

imposed by the eCl@ss-standard and the GEMOC initiative.[8] such as the Signal Process Modelling Language (SigPML)—a DSL dedicated to data flow processing.

Example model. Fig. 23 shows an example EPML model, which contains (*i*) electrical drives (DC Engine), (*ii*) communication cables or ready-made data cables that represent SigPML connectors, (*iii*) PC-based controls or field buses, i.e., decentralized peripherals, which represent controls, (*iv*) controls in terms of inductive proximity switches, and (*v*) sets of rectangular industrial connectors, which represent connector systems. In the figure, a DC Engine is connected to a Fieldbus through a communication cable and a data cable with a set of industrial connectors. The Fieldbus also has a connection with a Proximity Switch through a communication cable and a data cable using the industrial connectors that the Proximity Switch has.

Resource collection. In order to gather all the required elements we define a set of repositories and resources according to the data model shown in Fig. 2. The resources were taken from: (*i*) the GEMOC initiative, such as SigPML defined in form of an Ecore meta-model; (*ii*) the eCl@ss-standard, defined in form of several XML schema definitions (XSDs) and XML instances.

Table 3 summarizes the number of instances of the eCl@ss-standard as well as the SigPML by focusing on the meta-level. However, domain-specific concepts in the eCl@ss-standard measure a substantial size, i.e., only the basic and advanced specifications in the English language consist of 41,000 product classes and 17,000 properties, which amount to 15.5 Gb of data. In this situation, extracting desired concepts requires the manual examination of a vast amount of resources as well as their re-implementation by a target DSL. Moreover, any update that is performed on eCl@ss-standard resources, may also impact the implementation in the target system and involve complex and time-consuming maintenance tasks. In an effort to counteract such limitations we apply the Extremo framework as follows.

[8] http://www.gemoc.org.

Table 3
Number of collected instances of different XSD and EMF-concepts, respectively.

XSD concept in eCl@ss-standard	Number of instances	EMF concept in SigPML	Number of instances
XSD file	30	Ecore file/EPackage	1
xs:element	960	EClass	14
xs:element IDREF attribute	110	EReference	18
xs:attribute	104	EAttribute	10
xs:restriction	84	OCL EAnnotation	0

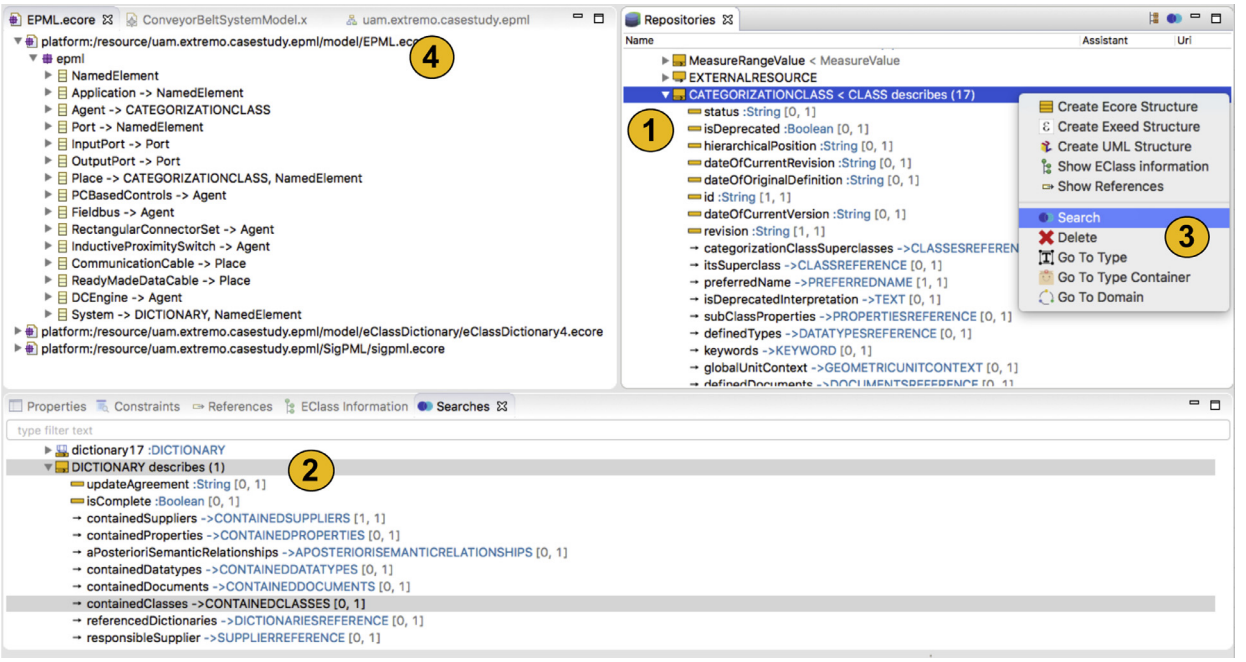


Fig. 24. Integration of EXTREMO with the Sample Reflective Ecore Model Editor (from the perspective of the case study).

Resource import. First, we import the resource collection by the application of the EcoreAssistant and the XsdAssistant. The XsdAssistant reuses functionality of the XMLINTELLEDIT framework [44]—composed of XMLTEXT [42] and INTELLEDIT [43]. The XMLTEXT framework transforms XML-artifacts, i.e., XSDs and XML instances, to corresponding MDE-artifacts, i.e., Ecore meta-models and conforming models. Then, the EcorAssistant is used to map the MDE-artifacts into the common data model.

Meta-model construction. Next, we employ the EXTREMO Eclipse perspective as well as the Sample Reflective Ecore Model Editor, integrated with EXTREMO following the approach described in Section 4.2.1. Fig. 24 shows a moment in the construction of the EPML meta-model. In particular, it shows the set of resources in the repository view (label 1) and the EXTREMO functionalities involved in the meta-model construction phase for querying (label 2), traversing (label 3), and applying desired concepts from the imported repositories (label 4). For example, available concepts that represent an electrical drive in the eCl@ss-standard are gathered by issuing an EXTREMO query for retrieving semantic nodes that are named "engine" and then used for creating corresponding concepts in the EPML meta-model. EXTREMO traversal-functionalities, such as Reveal On Repository, Go To Type, and Go To Domain, are employed for gathering respective classes, which are referenced as super-types and thus enforce conformance to the eCl@ss-standard.

The final result of the EPML meta-model construction process is depicted in Fig. 25. In detail, the EPML data flow process elements originated from the SigPML (in dark-grey) such as System, Application, Block, Connector, and Port. Moreover, EPML is composed of several eCl@ss-standard concepts (in light-grey), which include (*i*) electrical drives, (*ii*) cables, (*iii*) controls, (*iv*) binary sensors, i.e., safety-related sensors, and (*v*) connector systems. For example, the eCl@ss-standard CATEGORIZATIONCLASS represents the super-type of Block and Connector in EPML. Additionally, subtypes of Block and Connector are also instances of CATEGORIZATIONCLASS in the eCl@ss-standard. As a result of distinguishing specific instances of categorization-classes adds additional EPML-specific semantics that gather concepts found in SigPML and the eCl@ss-standard.

Result. Finally, we evaluate the capability of handling large models as they occur in the eCl@ss-standard in form of XML files, which are transformed to XMI files by the XMLINTELLEDIT framework to enable their use by EXTREMO. Further, SigPML (only) contains 13 semantic nodes at meta-model level and none at model-level and is thus neglected in

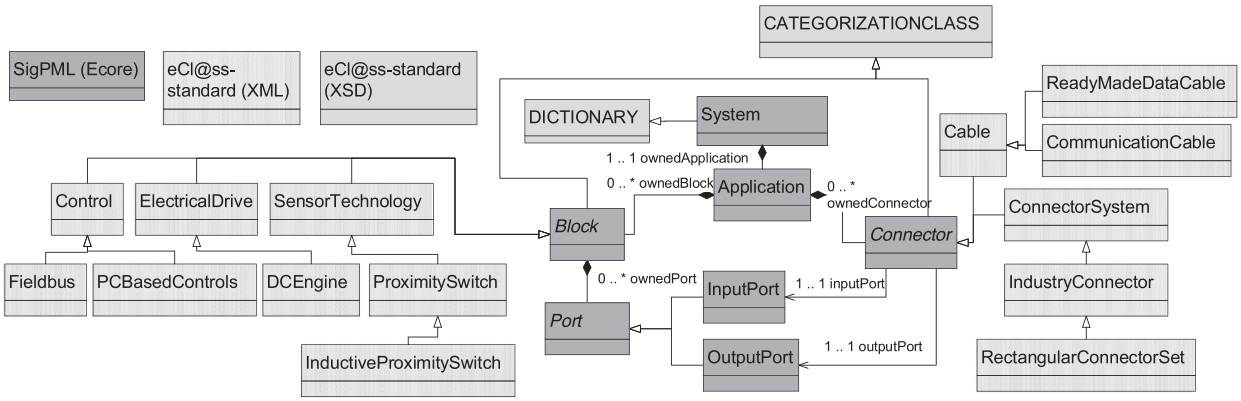


Fig. 25. Excerpt of industrial production system meta-model based on SigPML and the eCl@ss-standard.

Table 4
Instances of imported Common Data Model concepts within the industrial production system modelling case study.

	Number of instances	
Common Data Model concept	Meta-level (types)	Model-level (data)
Resource	1	1
SemanticNode	525	487,746 (4,071)
ObjectProperty	500	805,097 (23,752)
DataProperty	26	487,745
Constraint	88	820,356

Table 4. To summarize, the meta-level contains one *Resource*, i.e., "EPML.ecore", which references 18 different XSD files, that is instantiated by a single XML file, i.e., "eClass9_1_BASIC_EN_SG_27.xmi" (55.6 MB). Moreover, at the model-level there are 487,746 instances of semantic nodes (525 different kinds), 805,097 instances of object properties (500 different kinds), 487,745 instances of data properties (26 different kinds), and 820,356 instances of constraints[9] (88 different kinds).

Consequently, our results indicate that the Extremo-constructed EPML reduces the number of potential candidates for conveyor belt system components, which are available in the eCl@ss standard, by approximately 99.17% (97.05%), i.e., from 487,746 (805,097) to 4,071 (23,752) semantic nodes (object properties) that represent instances (references) of CATEGORIZATIONCLASS and thus potential candidates for instances (references) of (to) Block and Connector in SigPML.

Additional comments. EPML may be extended by either adding further eCl@ss-standard specific concepts, which represent instances of CATEGORIZATIONCLASS, to the meta-model or by expressing the concept of blocks and connectors as concrete (instead of abstract) classes. In more detail, the latter option would move the decision making-process of choosing desired eCl@ss-standard elements from meta-model level to model-level. Although Extremo supports such cases by the means of level-agnostic data handling, we choose to constrain EPML at meta-model level to limit the set of possible types, which can be instantiated at model-level, and thus fit the purpose of modelling conveyor-belt production systems.

5.2.3. Summary of the demonstration cases

The processes for immigration case study (Section 5.2.1) imports and queries data from different technical spaces, i.e., Ecore meta-models, CSV files, and OWL specifications and the industrial case study (Section 5.2.2) considers XML schemas, XML instances, and Ecore meta-models. Thus, in the first one, we consider a greater variety of technical spaces and smaller models. In contrast, in the second one we address the importing of a lower variety of technical spaces but larger models, i.e., XML instances in the size of multiple gigabytes. Then, in the first case study, we evaluate the ability of Extremo in providing assistance during the modeling of resources that are originated from a variety of technical spaces, and in the second case study, we evaluate the applicability of Extremo in assistance-scenarios that require dealing with industrially-sized resources.

Table 5 summarizes the number of meta-classes obtained from each assistant in both cases. In the first one, a total of 6 meta-classes were obtained according to the domain-specific concepts, 12 metaclasses were obtained from different ecores and the rest were added by hand. In the second one, a total of 16 meta-classes were obtained from different schemas and

[9] Note that the "number of instances" of constraints refers to the number of constraints defined at meta-level and validated at model-level.

Table 5
Evaluating the usefulness of EXTREMO: results of the experiments.

Measures	Case 1: processes for immigration	Case 2: production Systems
Size of metamodel obtained	27 metaclasses	23 metaclasses
From Ecore Assistant	13 metaclasses	7 metaclasses
From Ontologies Assistant	4 metaclasses	N/A
From CSV Assistant	2 metaclasses	N/A
From XSD/XML Assistant	N/A	16 metaclasses
Manually added	8	N/A

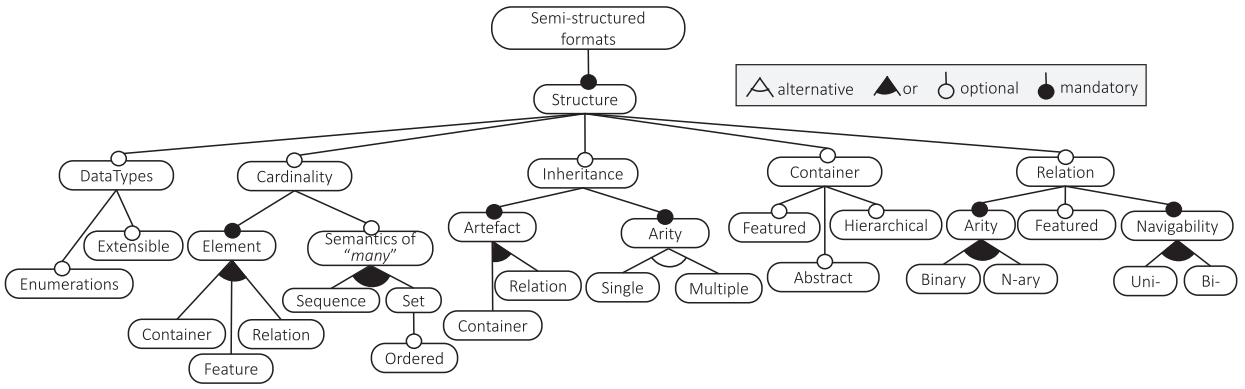


Fig. 26. Feature model of characteristics of structured formats (1/2).

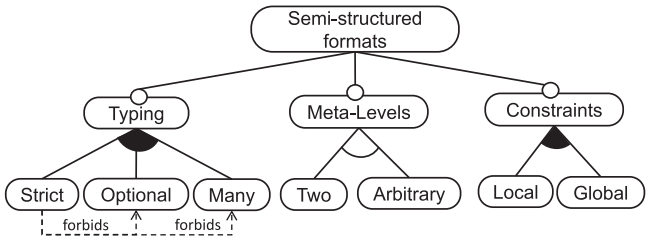


Fig. 27. Feature model of characteristics of structured formats (2/2).

descriptions using the XSD assistant and the rest from ecores. Overall, most content in both meta-models was reused from the available resources, which were taken from 4 different technical spaces.

From these demonstration cases, we can answer *RQ1: How useful is* EXTREMO*to solve practical problems in language engineering?* by stating that EXTREMO was helpful in locating elements within heterogeneous resources that helped to create the meta-models. These elements could be directly inserted in the final meta-model. For both cases, most elements in the meta-models were reused from the artefacts in the repository.

5.3. Evaluating format extensibility

In order to evaluate format extensibility, we perform an analytical evaluation of the degree of coverage of the data model of common features found in information modelling approaches [48,49]. Figs. 26 and 27 show a feature diagram (splitted in two parts for readability) displaying these features. Our aim is not to be fully exhaustive, but to cover a reasonable design space for information modelling approaches.

Fig. 26 shows features related to the space of possible supported structure primitives. Formats to represent semi-structured information are based on some kind of container for data (models, nodes, objects, classes), and on relations among them (features Container and Relation) [49,50]. Containers may support features (fields, references), may support nesting, and have ways to control their instantiability (e.g., abstractness tag). Relations have an arity, which is typicaly either binary (to model relations between exactly two containers) or n-ary (to model multi-container relations). Similar to containers, some systems may allow relations to own features. Relations may by navigable either in one or both directions [51,52].

Some systems support some form of inheritance to reuse information [51]. Inheritance relations can normally be set either between containers or relations, and be single (at most one super) or multiple (any number of super elements). Often, systems support a notion of cardinality, to specify the expected range of values of a given element can take. Typically, cardinalities can be attached to containers, relations or features. Some systems permit specifying the semantics of "many"

Table 6
Features of some information modelling technologies: DataType (E=Extensible, F=Fixed, EN=Enumerations), Cardinality (C=Container, F=Feature, R=Relation, Sem=Configurable semantics of many), Inheritance (C=Container, R=Relation, S=Single, M=Multiple), Container (F=Featured, A=Abstract, H=Hierarchical), Relation (Bin=Binary, N=N-ary, F=Featured, U=Unidirectional, B=Bidirectional), Typing (S=Strict, O=Optional, M=Many), Meta-Levels (2=Two, A=Arbitrary), Constraint (L=Local, G=Global).

System	DT	Card	Inh	Container	Relation	Typing	Levels	Const
EMF	E, EN	F, Sem	C, M	F, A, H	Bin, U, B	S	2	L, G
UML	E, EN	F, Sem	C, M	F, A, H	N, U, B, F	M	2	L, G
MetaDepth	F, EN	C, F, R, Sem	C, M	F, A, H	Bin, U, B, F	O	A	L, G
XSD	E, EN	F, Sem	C, S	F, H	Bin, U	S	2	L
OWL	E, EN	F, Sem	C, M	F, A	Bin, U, B	O, M	A	L
CSV	F	–	–	F	–	–	–	–
Extremo	F	F	C, M	F, A, H(Rsr)	Bin, U, B	O, M	A	L, G

cardinality: a set (no repetition allowed) optionally ordederd, or a sequence (repetitions allowed) [51,52]. Finally, many systems may have predefined data types (like integer, String, etc), have support for enumerations, and be extensible with new data types, provided by the user.

Fig. 27 captures additional features. Some systems organize their elements in meta-layers, typically two (e.g., models and meta-models), but others support an arbitrary number of them. If meta-levels are supported, some kind of typing is needed between entities in different layers. In some cases, this typing can be optional, or allowed to be multiple. In other cases, the typing is strict, meaning that each element of a certain level is typed exactly by one entity at the level above [53]. This precludes optional typing, multiple typings and typing relations within the same level [54]. Finally, some systems support constraints, typically defined on one meta-level, and evaluated in some meta-level below. These constraints can be local (i.e., involving features of just one entity) or global (if entities of the whole resource need to be access).

Once we have set the design space for information modelling formats, we analyse how different technologies are positioned in it, and the degree of coverage of our data model. A summary of such analysis is shown in Table 6.

In the modelling technical space, we have taken three representatives: EMF, UML and MetaDepth. It can be seen that EMF allows enumerations and extensible data types, permits cardinalities on features and fine-tuning the semantics of "many". It supports multiple inheritance on classes, classes (but not references) may have features, and can be abstract. Both packages and classes can be organized hierarchically (classes may define containment references that contain other classes). Relations are binary, and can be bidirectional (emulated through opposite references). The typing is strict and on a two meta-level architecture. Constraints can be expressed by OCL and can be both local and global. UML offers similar features, but includes N-ary, featured relations (association classes), and typing can be multiple (through overlapping generalization hierarchies). Finally, MetaDepth permits adding cardinalities on nodes, relations (edges) and features. Edges can have features, and there is optional typing on an arbitrary number of meta-levels.

In addition to the modelling technical space, we are interested in evaluating representatives from other technical spaces. First, we selected XSD which offers enumerations and extensible data types as well as cardinalities on features with different configurations for the semantics of "many". Furthermore, XSD offers for element types the following three possibilities: single inheritance, features, and nesting of element types (i.e., hierarchies). XSD only allows binary unidirectional references. Typing in XSD is considered to be strict, except open points in XSD descriptions that allow for any valid XML structure. XSD follows the classical two-level approach and allows for local constraints. For global constraints, additional format languages such as Schematron have to be used. OWL has similar features, but it allows for multiple inheritance between classes that may be also abstract classes. There is no explicit hierarchy based on nesting classes. Relations in OWL may be defined as bidirectional and addition to many other relationship types. Interestingly, OWL allows for optional typing as well as multiple types. Furthermore, arbitrary modeling levels may be defined with OWL, to be more precise, with OWL Full. Local constraints are supported, however, for global constraints, additional constraint languages such as SHACL [55] have to be used.

It can be seen that the data model of Extremo supports most features, with some limitations that can either be overcomed, or are not important for Extremo's goals, as explained next. First, Extremo's data types are currently not extensible. Instead, unsupported data types (e.g., currency) need to be mapped to an existing one (e.g., String) and the can be annotated using MetaData objects. Similarly, enumerations need to be stored as Strings. However, this is not problematic when issuing queries. Cardinality can only be placed on features, and the semantics of "many" is not configurable. However, this is not an important feature to issue queries, and can be reflected using MetaData or Constraint objects. Inheritance is on containers and can be multiple. This is in-line with most analyzed systems. Containers can have features and be abstract. However, only resources can be hierarchial. Nonetheless, hierarchy of semantic nodes can be emulated by ObjectProperties. Relations (ObjectProperties) are binary, can be bidirectional (by declaring opposites) and may have features. N-ary or featured associations can be emulated by adding an intermediate SemanticNode. This is the strategy we followed when building the assistant for MetaDepth. Extremo's typing is optional and multiple, supporting an arbitrary number of levels. Finally there is support for both local and global constraints.

Please note that the common data model can also accommodate data formats with no explicit descriptions, like e.g., CSV. In such a case, each data row would be imported as a semantic node, and each cell as a data property.

Table 7
LOCs for integrating Extremo with other tools

	DSL-tao	EcoreEditor	UML2Editor	ExeedEditor
Ext. Point Used	Drop	Actions	Actions	Actions
Ext. Point Integration	59*	49*	49*	49*
Tree Selection Solver	–	163	165	163
SemanticNode	24	8	4	8
DataProperty	27	33	4	33
ObjectProperty	24	9	32	9

Altogether, from this analytical evaluation, we can conclude that most common features of information modelling approaches can be directly mapped to our common data model, or can be emulated. Therefore, we can answer *RQ2: How capable is Extremo to represent information coming from different technological spaces?* by stating that Extremo will be able to accommodate most commonly used information modelling approaches.

5.4. Evaluating the integration with external tools

The idea of Extremo is to be a modelling assistant easy to integrate in other (meta-)modelling tools. Hence, we have assessed to what extent this integration is feasible, by integrating Extremo with a set of tools developed by third-parties. In some cases, the original code was not accessible, while in others it was. In the first case, we used the UML model editor[10] (as shown in Fig. 17), the standard Ecore editor and Exeed, an enhanced version of the built-in EMF reflective tree-based editor that enables developers to customize the labels and icons of model elements.[11] All these solutions are based on a TreeEditor, an extensible part of the Eclipse infraestructure. Since a drag and drop integration is not possible because of restrictions to access to the original code, the solution was performed by means of the action extension point. Each of these integrations costed 234 lines of Java code (LOCs) in average, which can be considered as very light.

In the second case, we used DSL-tao, which was built by our team. In this case, the code was available, and performing a solution by means of the drag and drop extension point. costed 134 lines of Java code (LOC).

Table 7 shows details on the number of LOCs for each integration. The integration mechanisms by means of actions and drag and drop are already provided by the tool (marked with an asterisk) and do not need to be provided by the developer. Therefore, most of the code needed was related to the transformation from the instances of our data model (Fig. 2) to the classes of the modelling tool (cf. Figs. 16 and 18). In the case of the integration made by means of actions, the method execute needs to resolve the editor part that will receive the portion of the model instance and the selected elements from the views before to create the new elements of the transformation (shown in row 3). The necessary LOCs to transform nodes, data and object properties are detailed in rows 4–6 of the table.

Thus, from this study, we can answer *RQ3: How integrable is* Extremo*?* by stating that integration of Extremo is lightweight for modelling tools based on tree or GEF-based editors.

5.5. Discussion and threats to validity

As we have seen in the three preceding subsections, we were able to use Extremo to help in constructing DSLs by reusing heterogeneous artefacts (some of which had large size); we analysed the degree in which the data model of Extremo is able to accommodate possible information modelling approaches; and how easy is it to integrate Extremo with external (meta-)modelling tools. While the results are positive, there are of course also potential threats to the validity of the experiments. According to Wohlin et al. [56], there are four basic types of validity threats that can affect the validity of our study. We cover each of these in the following paragraphs.

5.5.1. Construct validity

Construct validity is concerned with the relationship between theory and what is observed. The demonstration cases in Section 5.2 focussed on evaluating the use of Extremo with assistants for different technologies, and standards (like eCl@ss) developed by third parties. However, although taking realistic requirements, the DSLs to be constructed were devised by us. Therefore, further studies would need to be performed by constructing DSLs with requirements specified by third parties.

The evaluation of the demonstration cases focussed on DSL construction. However, it used artefacts acting as descriptors (XSD) and at the model/data level (XML documents). While this shows that Extremo can be used to extract information at the model level, a further study would be needed to assess the usefulness of Extremo for domain-specific modelling. However, please note that creating a meta-model is a structural modelling activity already.

5.5.2. Conclusion validity

Conclusion validity is concerned with the relationship between the treatment and the outcome. We considered two demonstration cases from different domains, seven format languages from four technical spaces, and integrated our ap-

[10] UML2-MDT, http://www.eclipse.org/modeling/mdt.
[11] Epsilon Exeed, http://www.eclipse.org/epsilon/.

proach with four modeling editors. While these numbers may be not enough to reason about statistical relevance, they still show a clear tendency of the usefulness, applicability, and integrability of our approach.

5.5.3. Internal validity

Internal validity checks whether the treatment used in the experiment actually causes the outcome. We were the performers of both demonstration cases. While the performer of one of the case study was not involved in the development of Extremo, a user study would be needed to further assess the tool usability and the subjective usefulness of Extremo. However reporting on a user study would deserve a separate publication, and we will tackle this issue in future work. Similarly, the integration of Extremo with external tools was also performed by us. Although lightweight in terms of LOC, it could be more demanding for other developers in terms of effort.

Another aspect is that we set the class as the unit of reuse, neglecting properties. We believe this is a good indicator as the number of classes outperforms that of properties.

Having good resources available is crucial for the approach to work properly. We did not evaluate how easy is it to perform this phase of resource collection (since this phase is out of the scope of our tool), but we evaluated how large was the contect of the repository, though.

5.5.4. External validity

Regarding external validity (i.e., generalizability of the results), we did not include an explicit evaluation of query extensibility, because the extension points we have defined permit adding new queries by using arbitrary Java code. Table A.1 in the appendix lists a collection of queries we have defined by implementing the extension point and that covers a set of accepted quality criteria in conceptual modelling [24].

For RQ3 (integrability) we did not evaluate the integration of Extremo with text-based modelling tools, e.g., built with Xtext, but we have assessed to what extent this integration is feasible, by integrating Extremo with a set of tools developed by third-parties (and also developed by us). In addition, we integrated Extremo with other tools within Eclipse, but not with tools in other IDEs, like JetBrains. While Extremo is an Eclipse plugin, its Core subsystem (described in Section 4.1) is largely independent from it (in contrast to the UI subsystem 4.2). Hence, migrating the Core into JetBrains would require little effort, but the UI subsystem (dealing with visualization and interaction with resources and query results) would need to be redesigned.

We did not present a formal evaluation of scalability or performance, which are left for future work. Regarding the former, the XML artefacts considered in the second demonstration case reached a size of 55 Mb. Moreover, resources are imported and persisted using NeoEMF, a model persistence solution designed to store models in NoSQL datastores, which is able to handle very large models efficiently (e.g., models of more than 40.000.000 elements were reported to be created in [57]).

Regarding performance, our experience and preliminary evaluations indicate that resource import time is linear in the size of the resource. Typically, it takes a few seconds for resources of sizes in the order of hundreds of elements. While we plan to optimize this performance, this is a one-time operation, and once a resource is imported, it can be handled through NeoEMF. Regarding query performance, those that need to traverse the whole resource are in the order of one second for sizes up to thousands of elements. However, they may become a bottleneck for larger resources. To alleviate this issue, we cache both the input parameters for predicate-based searches (init operation shown in Fig. 6) and the query results, while NeoEMF lazy-loading mechanisms that transparently brings into memory model elements only when they are accessed. Further optimizations to speed-up queries, e.g., based on the creation of derived features and indexes for the resources [58], are left for future work.

6. Related work

The increasing complexity of software development has prompted the need for code recommenders for example, for API usage, or program quick fix. However, although code recommenders are increasingly used in programming IDEs [7,8], there is lack of such assistive tools for (meta-)modelling in MDE.

The closest work to our proposal is [59–61], where a generic architecture for model recommenders is proposed. The architecture is extensible, in the sense that different recommender strategies can be plugged-in. In contrast, the extensibility of our approach is in the supported data source, while we specifically focus on the extraction of knowledge from these sources. In addition, our approach supports out-of-the-box visualization and extensible query facilities.

Other approaches to model recommendation focus on proposing suitable ways to complete a model with respect to a meta-model [62]. Hence, using constraint solving techniques, the system proposed ways to complete a model so that it becomes a valid instance of a meta-model. In [63] the authors use ontologies in combination with domain-specific modelling, and hence can use ontology reasoners to provide reasoning services like model validation, inconsistency explanation, and services to help in the model construction phase.

Some approaches propose a common architecture to index and represent models, with the purpose of reuse. For example, in [64] the authors transform SysML models into a “universal” representation model called RHSP, which can be queried to discover reusable models. They support queries based on partial models (and model similarity search) and natural language (similar to our synonym searchs). In our case the queries are extensible, and our data model provides richer support for representing model features, including constraints.

Table 8
Summary comparison of Extremo and closest related works

Work	Assistance	Heterogeneous sources	Common model	Queries
Dyck et al. [59–61]	✓	✗	✗	✗
Sen et al. [62]	✓	✗	✗	✗
Walter et al. [63]	✓	∼ (OWL)	✗	✗
Mendieta et al. [64]	✗	∼ (SysML)	✓	Not extensible
Kern, Dimitrieski et al. [67,68]	✗	✓	✗	✗
Ontology-based DSL development [70,71]	✗	∼ (OWL)	✗	✗
Moogle [34]	✗	∼ (EMF)	✗	Not extensible
Extremo	✓	✓	✓	Extensible

Instead of using a common data model, an alternative design would have been to use model adapters, in the style of the Epsilon model management languages [65]. In this approach, the languages do not access the particular information technology (EMF, XML) directly, but through a model connectivity layer. This is an interface that can be implemented to enable uniform access to different technologies. We opted for a common data model, where the different heterogeneous elements are reified uniformly, and stored using NeoEMF, hence providing scalability and performance.

Storing artefacts in a database, to enable their flexible query has also been applied to source code [15,66]. In our case, the artefacts come from different heterogeneous sources, and hence we need to transform them into the common data model. Our query approach is extensible, based on extension points.

Other works have considered the exchange of models/data between different meta-modelling tools [67] or technical spaces [68]. In [67] the authors propose a solution that creates transformation between different meta-modelling technologies by means of declarative mappings. Our approach differs from these works by mapping the technical spaces into a common data model instead of establishing mappings between individual technical spaces. As a result, our approach is independent of a single technical space and thus enables the import, persistence, and querying of interdependent concepts that are originated from distinctive technical spaces and may be lost within single mappings. In [68] the maintenance of intra-space transformations is improved by automating the discovery and reuse of mappings between schema elements. In contrast, Extremo provides assistance during the import of artifacts from different technical spaces and the creation of new languages and models that are based on existing technical spaces, such as Ecore, regardless of their originating technical space. Although Extremo requires to specify assistants for different technical spaces, existing EMF-based work that bridges technical spaces, such as XMLText [42] for XML schema, can be reused and (only) requires the specification of a mapping within the same technical space, i.e., Ecore in case of Extremo and XMLText.

Some researchers have exploited ontologies for creating DSLs [69]. For example, in [70] the authors advocate the use of (OWL) ontologies in the domain analysis phase of DSL construction. As they target textual DSLs, they propose a tool for the automated generation of a textual grammar for the DSL. In a similar vein, in [71], the authors generate meta-model design templates from OWL ontologies, which are later manually refined into domain meta-models. In our approach, we assume that not all the required information to create a meta-model is present in one ontology, but typically such information is scattered in informational resources of different kinds, like ontologies, RDF data, or meta-models.

Combining modeling approaches from MDE with ontologies has been studied in the last decade [72]. There are several approaches to transform Ecore-based models to OWL and back, e.g., cf. [73,74]. In addition, there exist approaches that allow for the definition of ontologies in software modeling languages such as UML by using dedicated profiles [75]. Moreover, there are approaches that combine the benefits of models and ontologies such as done in [76,77] for reasoning tasks. Not only the purely structural part of UML is considered, but some works also target the translations of constraints between these two technical spaces by using an intermediate format [78]. For the data import, we may build on these mentioned approaches, but we focus on recommendation services exploiting the imported data from different technical spaces to build domain-specific modeling languages.

Finally, there are some approaches directed to search relevant models within a repository. Their aim is slightly different from our goal, which is looking for relevant information within a repository. Moogle [34] is based on textual, “Google-like” queries, similar to ours. As they focus on EMF model-level queries, they use the meta-model information for filtering, like we do as well. However, our queries are extensible, and hence new types of queries can be defined. Moreover, their results are shown in textual format and we parse and aggregate the results as well as offer graphical visualization. EMF query is directed to search EMF models [79], using OCL queries or text-based search. The latter may include regular expressions, but does not look for relevant synonyms as we do. Moreover, our extensible approach supports technologies like Ecore, OWL and RDF. Furthermore, there are dedicated approaches offering search capabilities tailored for a specific modelling domain such as [80,81]. Although these approaches allow to reason on behavioral similarity aspects, we aim for general model search support independently of the modelling domain and technical space.

Table 8 presents a feature-based summary of Extremo and the closest related works. In summary, our approach is novel in that it provides an assistant system that profits from the integration and querying of heterogeneous information sources. Although some approaches have focussed on using specific technologies, such as Ontologies [63,69–71], to build (meta-)models, our approach is more general as a result of supporting different technologies. Moreover, there exist approaches

that establish bridges between technical spaces [67,68], our contribution differs by providing a common data model to store, query, and establish assistance for information from different technical spaces. Further, in contrast to other existing approaches, which have devised query mechanisms to search for relevant models in a repository [34], our querying mechanism is extensible. Finally, some approaches to provide model assistance are based on model completion (w.r.t. a metamodel) [62] or provide a generic mechanism to plug-in assistants [59–61]. Contrarily, we contribute a specific architecture to support assistance that is based on querying heterogeneous information sources.

7. Conclusions and future work

In this paper, we have presented Extremo, an extensible assistant for modelling and meta-modelling. The system is able to gather information from different technological spaces (like ontologies, RDF, XML or EMF), by representing this information under a common data scheme. This enables their uniform querying and visualization. Extremo is independent of the particular modelling tool, but easily integrated with them due to its modular architecture based on extension points. We have shown its integration with DSL-tao and several other tools, and used it for the construction of DSLs in the e-Government and production systems domain. We have performed an evaluation of several aspects, showing good results.

In the future, we plan to connect Extremo with meta-model repositories, such as MDEForge [82]. Extremo currently supports a re-active integration mode, where the assistant is explicitly invoked.

Similar to [60], we would also like to explore pro-active modes for assistance. For this purpose, we plan to use recommendation techniques based on rich contextual models, which take into account not only the current model state, but also the user interaction with the IDE [83]. We are currently considering a user study, made of two parts. First, we will evaluate the perceived usefulness of Extremo by engineers in order to perform different modelling tasks (e.g., construct or modify a model). Second, we will compare the quality of the resulting models, and the effectivenes of the modelling task, with respect to not using assistance. Finally, we plan to improve query efficiency by using model indexes [58].

Acknowledgements

We would like to thank to the reviewers for their valuable comments. This work was supported by the Ministry of Education of Spain (FPU grant FPU13/02698); the Spanish MINECO (TIN2014-52129-R); the R&D programme of the Madrid Region (S2013/ICE-3006); the Austrian agency for international mobility and cooperation in education, science and research (OeAD) by funds from the Austrian Federal Ministry of Science, Research and Economy - BMWFW (ICM-2016-04969); and by the Christian Doppler Forschungsgesellschaft, the Federal Ministry of Economy, Family and Youth and the National Foundation for Research, Technology and Development, Austria.

Appendix A. List of queries

Table A.1 shows the list of pre-defined queries of Extremo, which were defined by implementing the provided extension points. The user may provide additional queries by implementing the extension point (in Java). Some of these queries come from catalogues of accepted quality criteria in conceptual modelling [24]. We divide them into predicate and custom, and depict the element they inspect (filterBy).

Table A.1
List of simple search configurations (queries).

SearchConfiguration	filterBy	Description
Predicate Based Search		
A NamedElement has a name	NamedElement	Checks if a NamedElement object has a name that matches with a value. Options: name: PrimitiveTypeParam typed as string
All instances of a NamedElement	NamedElement	Returns all the NamedElements that are instances of another one. Options: type: ModelTypeParam and recursive: PrimitiveTypeParam typed as boolean
A node with a property with value X	SemanticNode	Checks if a SemanticNode object has a data property that has a concrete value. Options: value: PrimitiveTypeParam typed as string
A node has more than a number of parents	SemanticNode	Checks if a SemanticNode object has more than a number of supers instances. Options: parents: PrimitiveTypeParam typed as int
A node has more than a number of children	SemanticNode	Checks if a SemanticNode object has more than a number of subs instances. Options: children: PrimitiveTypeParam typed as int
Attributes are overloaded	SemanticNode	Checks if a SemanticNode object contains more than a number of DataProperties. Options: maxattrs: PrimitiveTypeParam typed as int
References are overloaded	SemanticNode	Checks if a n: SemanticNode object contains more than a number of ObjectProperties. Options: maxrefs: PrimitiveTypeParam typed as int
An abstract node without children	SemanticNode	Checks if a SemanticNode object is abstract and there are no supers instances.

(continued on next page)

Table A.1 (*continued*)

SearchConfiguration	filterBy	Description
An abstract node with an unique child	SemanticNode	Checks if a SemanticNode object is abstract and it has only a supers instance.
Data Properties Value	DataProperty	Checks if a DataProperty object has a concrete value. Options: valuefield: PrimitiveTypeParam typed as string
Data Properties Value Range	DataProperty	Checks if the integer value of a DataProperty object ranges between a minimum and a maximum. Options: minvaluefield: PrimitiveTypeParam typed as int and maxvaluefield: PrimitiveTypeParam typed as int
Custom Search		
Nodes without descriptions	Resource	For a resource, split the nodes in two groups. The first one refers to the nodes with descriptions are the second one refers to the nodes without descriptions. Options: resource: ModelTypeParam typed as Resource
Isolated nodes	Resource	Checks if a resource contains nodes that are isolated. Options: resource: ModelTypeParam typed as Resource
A hierarchy is too deep	Resource	Checks if a node is too deep on a level of hierarchy. Options: maxdepth: PrimitiveTypeParam typed as int
No node is referred from too many others	Resource	In a resource, checks if there is a node that is referred from too many others. Options: maxrefs: PrimitiveTypeParam typed as int and resource: ModelTypeParam typed as Resource
Hierarchy Splitter	Resource	For a resource, split the nodes in groups. In every group a inheritance hierarchy is left. Options: resource: ModelTypeParam typed as Resource

References

[1] Brambilla M, Cabot J, Wimmer M. Model-Driven software engineering in practice. Morgan & Claypool; 2017.
[2] Schmidt DC. Guest editor's introduction: Model-driven engineering. Computer 2006;39(2):25–31.
[3] Silva ARd. Model-driven engineering: A survey supported by the unified conceptual model. Comput Lang Syst Struct 2015;43(Supplement C):139–55. doi:10.1016/j.cl.2015.06.001.
[4] UML 2.5 OMG specification. http://www.omg.org/spec/UML/2.5/;
[5] Kelly S, Tolvanen J-P. Domain-specific modeling - enabling full code generation. Wiley; 2008.
[6] Hutchinson JE, Whittle J, Rouncefield M. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. Sci Comput Program 2014;89:144–61.
[7] Eclipse code recommenders. http://www.eclipse.org/recommenders;
[8] Robillard MP, Walker RJ, Zimmermann T. Recommendation systems for software engineering. IEEE Softw 2010;27(4):80–6.
[9] Bézivin J. Model driven engineering: An emerging technical space. In: Generative and transformational techniques in software engineering, international summer school, GTTSE. In: Lecture Notes in Computer Science, vol. 4143. Springer; 2005. p. 36–64.
[10] Kurtev I, Bézivin J, Aksit M. Technological spaces: an initial appraisal. International symposium on distributed objects and applications, DOA 2002; 2002. http://doc.utwente.nl/55814/.
[11] Segura AM, Pescador A, de Lara J, Wimmer M. An extensible meta-modelling assistant. In: Proceedings of the IEEE EDOC. IEEE Computer Society; 2016. p. 1–10.
[12] de Lara J, Guerra E. Deep meta-modelling with metadepth. In: Proceedings of the TOOLS. In: Lecture Notes in Computer Science, vol. 6141. Springer; 2010. p. 1–20.
[13] eCl@ss Standard 9.0. http://wiki.eclass.eu/.
[14] Ménard PA, Ratté S. Concept extraction from business documents for software engineering projects. Autom Softw Eng 2016;23(4):649–86.
[15] Linstead E, Bajracharya SK, Ngo TC, Rigor P, Lopes CV, Baldi P. Sourcerer: mining and searching internet-scale software repositories. Data Min Knowl Discov 2009;18(2):300–36.
[16] Subramanian S, Inozemtseva L, Holmes R. Live API documentation. In: Proceedings of the ICSE '14. ACM; 2014. p. 643–52.
[17] Treude C, Robillard MP. Augmenting API documentation with insights from stack overflow. In: Proceedings of the 38th international conference on software engineering. ICSE '16. New York, NY, USA: ACM; 2016. p. 392–403. ISBN 978-1-4503-3900-1.
[18] Basciani F, Rocco JD, Ruscio DD, Iovino L, Pierantonio A. Automated clustering of metamodel repositories. In: Proceedings of the CAiSE. In: Lecture Notes in Computer Science, vol. 9694. Springer; 2016. p. 342–58.
[19] Pescador A, Garmendia A, Guerra E, Cuadrado JS, de Lara J. Pattern-based development of domain-specific modelling languages. In: Proceedings of the MoDELS; 2015. p. 166–75.
[20] Czarnecki K, Antkiewicz M. Mapping features to models: A template approach based on superimposed variants. In: Proceedings of the GPCE. Berlin, Heidelberg: Springer-Verlag; 2005. p. 422–37.
[21] Polzer A, Merschen D, Botterweck G, Pleuss A, Thomas J, Hedenetz B, Kowalewski S. Managing complexity and variability of a model-based embedded software product line. Innov Syst Softw Eng 2012;8(1):35–49.
[22] Cuadrado JS, Guerra E, de Lara J. A component model for model transformations. IEEE Trans Softw Eng 2014;40(11):1042–60.
[23] Moha N, Guéhéneuc Y-G, Duchien L, Meur A-FL. DECOR: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 2010;36(1):20–36.
[24] Aguilera D, Gómez C, Olivé A. Enforcement of conceptual schema quality issues in current integrated development environments. In: Proceedings of the CAiSE. In: Lecture Notes in Computer Science, vol. 7908. Springer; 2013. p. 626–40.
[25] Kurtev I, Aksit M, Bézivin J. Technical spaces: an initial appraisal. Proceedings of the CoopIS; 2002.
[26] Atkinson C, Kühne T. Reducing accidental complexity in domain models. Softw Syst Model 2008;7(3):345–59.
[27] Atkinson C, Kennel B, Goß B. The level-agnostic modeling language. Berlin, Heidelberg: Springer Berlin Heidelberg; 2011. p. 266–75.
[28] de Lara J, Guerra E. *A Posteriori* typing for model-driven engineering: Concepts, analysis, and applications. ACM Trans Softw Eng Methodol 2017;25(4) 31:1–31:60.
[29] OMG. SMOF 1.0. http://www.omg.org/spec/SMOF/1.0/; 2013.
[30] Diskin Z, Kokaly S, Maibaum T. Mapping-aware megamodeling: design patterns and laws. In: Proceedings of the SLE. In: Lecture Notes in Computer Science, vol. 8225. Springer; 2013. p. 322–43.
[31] de Lara J, Guerra E, Cobos R, Moreno-Llorena J. Extending deep meta-modelling for practical model-driven engineering. Comput J 2014;57(1):36–58.
[32] Steinberg D, Budinsky F, Paternostro M, Merks E. EMF: eclipse modeling framework. Addison-Wesley; 2008.
[33] OMG. MOF 2.5.1. http://www.omg.org/spec/MOF/2.5.1/; 2016.
[34] Lucrédio D, de Mattos Fortes RP, Whittle J. MOOGLE: a metamodel-based model search engine. Softw. Syst. Model. 2012;11(2):183–208.

[35] OCL 2.4. specification. http://www.omg.org/spec/OCL/.
[36] Porter MF. An algorithm for suffix stripping. Program 2006;40(3):211–18.
[37] Lesk M. Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone. In: Proceedings of the SIGDOC. ACM; 1986. p. 24–6.
[38] Miller GA. Wordnet: a lexical database for english. Comm ACM 1995;38(11):39–41.
[39] Kharlamov E, Grau BC, Jiménez-Ruiz E, Lamparter S, Mehdi G, Ringsquandl M, Nenov Y, Grimm S, Roshchin M, Horrocks I. Capturing industrial information models with ontologies and constraints. In: Proceedings of the ISWC, Part II. In: Lecture Notes in Computer Science, vol. 9982; 2016. p. 325–43.
[40] Motik B, Horrocks I, Sattler U. Adding integrity constraints to OWL. Proceedings of the OWLED. CEUR Workshop Proceedings, vol. 258. CEUR-WS.org; 2007. http://ceur-ws.org/Vol-258.
[41] Benelallam A, Gómez A, Sunyé G, Tisi M, Launay D. Neo4emf, a scalable persistence layer for EMF models. In: Proceedings of the ECMFA. In: Lecture Notes in Computer Science, vol. 8569. Springer; 2014. p. 230–41.
[42] Neubauer P, Bergmayr A, Mayerhofer T, Troya J, Wimmer M. XMLText: from XML schema to Xtext. In: Proceedings of the SLE; 2015. p. 71–6.
[43] Neubauer P, Bill R, Mayerhofer T, Wimmer M. Automated generation of consistency-achieving model editors. In: Proceedings of the IEEE SANER; 2017. p. 127–37.
[44] Neubauer P, Bill R, Wimmer M. Modernizing domain-specific languages with XMLText and intelledit. In: Proceedings of the IEEE SANER; 2017. p. 565–6.
[45] Eclipse graphical editing framework. https://eclipse.org/gef/.
[46] Runeson P, Host M, Rainer A, Regnell B. Case study research in software engineering: guidelines and examples. 1st. Wiley Publishing; 2012. ISBN: 1118104358, 9781118104354.
[47] OMG business process model and notation. http://www.bpmn.org/.
[48] Mylopoulos J. Characterizing information modeling techniques. Berlin, Heidelberg: Springer Berlin Heidelberg; 1998. p. 17–57.
[49] Kern H, Hummel A, Kühne S. Towards a comparative analysis of meta-metamodels. In: Proceedings of the compilation of the co-located workshops, DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, and VMIL'11. Portland, OR, USA; 2011. p. 7–12.
[50] Buneman P. Semistructured data. In: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems. ACM Press; 1997. p. 117–21.
[51] Olivé A. Conceptual modeling of information systems. Springer; 2007. doi:10.1007/978-3-540-39390-0.
[52] Abiteboul S, Buneman P, Suciu D. Data on the web: from relations to semistructured data and XML. Morgan Kaufmann; 1999. ISBN 1-55860-622-X.
[53] Atkinson C, Kühne T. Profiles in a strict metamodeling framework. Sci Comput Program 2002;44(1):5–22.
[54] Kühne T. Matters of (meta-)modeling. Softw Syst Model 2006;5(4):369–85.
[55] Shapes constraint language (SHACL). https://w3c.github.io/data-shapes/shacl/.
[56] Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B. Experimentation in software engineering. Springer; 2012.
[57] Daniel G, Sunyé G, Benelallam A, Tisi M. Improving memory efficiency for processing large-scale models. BigMDE. York, UK, United Kingdom: University of York; 2014. https://hal.inria.fr/hal-01033188.
[58] Barmpis K, Kolovos DS. Towards scalable querying of large-scale models. In: Proceedings of the ECMFA. In: Lecture Notes in Computer Science, vol. 8569. Springer; 2014. p. 35–50.
[59] Dyck A, Ganser A, Lichter H. Enabling model recommenders for command-enabled editors. In: Proceedings of the MDEBE; 2013. p. 12–21.
[60] Dyck A, Ganser A, Lichter H. A framework for model recommenders - requirements, architecture and tool support. In: Proceedings of the MODELSWARD; 2014. p. 282–90.
[61] Dyck A, Ganser A, Lichter H. On designing recommenders for graphical domain modeling environments. In: Proceedings of the MODELSWARD; 2014. p. 291–9.
[62] Sagar S, Baudry B, Vangheluwe H. Towards domain-specific model editors with automatic model completion. Simulation 2010;86(2):109–26.
[63] Walter T, Parreiras FS, Staab S. An ontology-based framework for domain-specific modeling. Softw Syst Model 2014;13(1):83–108.
[64] Mendieta R, de la Vara JL, Llorens J, Álvarez-Rodríguez J. Towards effective sysML model reuse. In: Proceedings of the MODELSWARD. SCITEPRESS; 2017. p. 536–41.
[65] Paige RF, Kolovos DS, Rose LM, Drivalos N, Polack FAC. The design of a conceptual framework and technical infrastructure for model management language engineering. In: Proceedings of the ICECCS. IEEE Computer Society; 2009. p. 162–71.
[66] Hajiyev E, Verbaere M, de Moor O, De Volder K. Codequest: querying source code with datalog. In: Proceedings of the OOPSLA. ACM; 2005. p. 102–3.
[67] Kern H, Stefan F, Dimitrieski V, Čeliković M. Mapping-based exchange of models between meta-modeling tools. In: Proceedings of the DSM. New York, NY, USA: ACM; 2014. p. 29–34. ISBN 978-1-4503-2156-3.
[68] Dimitrieski V, Čeliković M, Igić N, Kern H, Stefan F. Reuse of rules in a mapping-based integration tool. In: Proceedings of the SoMet. Cham: Springer International Publishing; 2015. p. 269–81. ISBN 978-3-319-22689-7.
[69] Tairas R, Mernik M, Gray J. Using ontologies in the domain analysis of domain-specific languages. In: Proceedings of the TWOMDE; 2008. p. 20–31.
[70] Ceh I, Crepinsek M, Kosar T, Mernik M. Ontology driven development of domain-specific languages. Comput Sci Inf Syst 2011;8(2):317–42.
[71] Ojamaa A, Haav H-M, Penjam J. Semi-automated generation of DSL meta models from formal domain ontologies. In: Proceedings of the MEDI; 2015. p. 3–15.
[72] Gasevic D, Djuric D, Devedzic V. Model driven engineering and ontology development. Springer; 2009.
[73] Walter T, Parreiras FS, Gröner G, Wende C. OWLizing: transforming software models to ontologies. Proceedings of the ODiSE; 2010. 7:1–7:6.
[74] Kappel G, Kapsammer E, Kargl H, Kramler G, Reiter T, Retschitzegger W, Schwinger W, Wimmer M. Lifting metamodels to ontologies: a step to the semantic integration of modeling languages. In: Proceedings of the MoDELS; 2006. p. 528–42.
[75] Milanovic M, Gasevic D, Giurca A, Wagner G, Devedzic V. Towards sharing rules between OWL/SWRL and UML/OCL. ECEASST 2006;5:2–19.
[76] Parreiras FS, Staab S, Winter A. On marrying ontological and metamodeling technical spaces. In: Proceedings of the FSE; 2007. p. 439–48.
[77] Parreiras FS, Staab S. Using ontologies with UML class-based modeling: the TwoUse approach. DKE 2010;69(11):1194–207.
[78] Djuric D, Gasevic D, Devedzic V, Damjanovic V. A UML profile for OWL ontologies. In: Proceedings of the MDAFA; 2004. p. 204–19.
[79] EMF Query. https://projects.eclipse.org/projects/modeling.emf.query.
[80] Bislimovska B, Bozzon A, Brambilla M, Fraternali P. Textual and content-based search in repositories of web application models. TWEB 2014;8(2):1–11.
[81] Dijkman RM, Dumas M, van Dongen BF, Käärik R, Mendling J. Similarity of business process models: Metrics and evaluation. Inf Syst 2011;36(2):498–516.
[82] Rocco JD, Ruscio DD, Iovino L, Pierantonio A. Collaborative repositories in model-driven engineering. IEEE Softw 2015;32(3):28–34.
[83] Gasparic M, Murphy GC, Ricci F. A context model for IDE-based recommendation systems. J Syst Softw 2017;128:200–19.