Model-driven Language Engineering: the ASMETA case study

Angelo Gargantini
Dip. di Ing. Informatica e Metodi Matematici
Università di Bergamo, Italy
angelo.gargantini@unibg.it

Elvinia Riccobene, Patrizia Scandurra
Dip. di Tecnologie dell'Informazione
Università di Milano, Italy
{riccobene,scandurra}@dti.unimi.it

Abstract

This paper reports our experience in exploiting the metamodelling approach of model-driven language engineering *to define a standard modelling language for the Abstract State Machines (ASMs) formal method, and develop a general framework (ASMETA) for a wide interoperability of ASM tools in a model-driven development context. We describe the requirements to fulfill and the design/implementation/validation/tools development steps necessary to support such a language engineering life cycle. We finally discuss the benefits/limits of a model-driven language engineering approach with respect to traditional techniques primarily used for the same goal.*

1 Introduction

The *model-based* approach to software development promotes models as first-class entities that need to be maintained, analyzed, simulated and otherwise exercised, and mapped into programs and/or other models. In the context of (software) language engineering, we refer to *model-driven language engineering* when language descriptions are first class artifacts of the model-based approach, and the abstract syntax of the language is defined in terms of a (usually object-oriented) model, called *metamodel*, which allows separating the abstract syntax and semantics of the language constructs from their different concrete notations. Indeed, the metamodel of a language describes the vocabulary of concepts provided by the language, the relationships existing among those concepts, and how they may be combined to create models. A metamodel based abstract syntax definition has the great advantage of being suitable to derive from the same metamodel (through mappings or projections) different alternative concrete notations, textual or graphical or both, for various scopes like graphical rendering, model interchange, standard encoding in programming languages, and so on, still maintaining the same semantics. Therefore, a metamodel could be intended as a standard representation of the language notation.

Here, we address the issue of applying the basic principles of model-driven language engineering to the ASM (Abstract State Machine) domain, in order to engineer a metamodel-based language for ASMs [2, 6] and its associated supporting tools.

The success of the ASMs as a system engineering method able to guide the development of complex systems, from requirements capture to their implementation, is nowadays widely acknowledged [2]. As discussed in [14], a unified notation and interchange format is of primary interest for the ASMs community, since ASM tools have been usually developed by individual research groups, are loosely coupled, and have syntaxes and internal representations of ASM models strictly depending on the implementation environment. This makes the encoding of ASM mathematical models not always natural and straightforward and makes the integration of tools hard to accomplish, so preventing ASMs from being used in an efficient and tool supported manner during the software development life-cycle.

To achieve the goals of *developing* a unified abstract notation for ASM, a notation independent from any specific implementation syntax and allowing a more direct encoding of the ASM mathematical concepts and constructs, and *tackling* the problem of ASM tool inter-operability, we exploited the *metamodelling* approach suggested by the model-driven development.

In this paper, we report our experience in engineering a standard metamodel-based modelling language for ASMs, and building a general framework suitable to the development of new ASM tools for ASMs and for the integration of existing ones. We describe the requirements, the design/implementation/validation steps that are necessary to support the (software) language engineering life cycle, and we present a set of tools for ASMs that we have developed by using the ASMs Metamodelling (ASMETA) framework [1], i.e. an instantiation of the OMG MDA framework for the ASM domain. We summarize the lessons learned by providing evidence of the benefits/limits of model-driven language engineering. Although this work is specific to the ASMs, our experience can be useful for de-

veloping metamodel-based languages based on the concepts of "abstract state" and "transitions" for which, indeed, well-established object-oriented metamodelling patterns may not always be reused.

The remainder of this paper is organized as follows. Sect. 2 introduces basic concepts underlying ASMs. Sect. 3 presents the overall process of engineering a language for ASMs by metamodelling. It has implied the development of the *Abstract State Machine Metamodel* (AsmM) as abstract syntax description of the language, and the implementation (obtained in a generative manner from the AsmM) of some derivative artifacts. Sect. 4, discusses over benefits/limits of model-driven language engineering compared to traditional techniques, like graph grammars, primarily used for the same scope. Finally, related and future work are given in Sect. 5.

2 Abstract State Machines

Abstract State Machines are an extension of FSMs, where unstructured control states are replaced by states with arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates defined on them, while the *transition relation* is specified by "rules" describing the modification of the functions from one state to the next. A complete mathematical definition of the ASM method can be found in [2, 6]. The notion of ASMs moves from a definition which formalizes simultaneous parallel actions of a single agent, either in an atomic way, *Basic ASMs*, and in a structured and recursive way, *Structured or Turbo ASMs*, to a generalization where multiple agents interact *Multi-agent ASMs*. Appropriate rule constructors also allow non-determinism and unrestricted synchronous parallelism.

3 Model-driven language engineering

According to the metamodelling approach to (software) language engineering, a language definition comprises:
– an *abstract syntax*, i.e. a metamodel representing in an abstract (and possibly visual) way concepts and constructs of the modelling language, and providing the means (usually constraints) to distinguish between valid and invalid models, i.e. *conformance*;
– one or more *concrete syntaxes*, textual or visual o mixed, derived from the metamodel as notation to be used by language users to effectively write *models* conforming to the langauge metamodel;
– a *semantics*, i.e. the abstract logical space in which models, written in the given language, find their meaning.

Therefore, the overall iterative – often we had to come back to previous steps and make corrections – process of developing a metamodel-based language for ASMs and implementing several artifacts (as concrete syntaxes, interchange formats, APIs, etc.) and supporting tools consisted of the following steps (later explained in detail):

1. Language requirements capture and analysis;
2. Choice of a metamodelling framework and supporting technologies;
3. Language design by metamodelling;
4. Implementation of *language artifacts*, i.e. metamodel derivatives, to handle ASM models:
   (a) an interchange syntax, usually XMI/XML-based, for serializing ASM models;
   (b) APIs to access and manipulate ASM models in a model repository or metamodelling framework;
   (c) one or more concrete syntaxes (textual, graphical, or mixed) for human use with their associated parsers for conformance-checking of models against the metamodel;
5. Validation of the metamodel and its derivatives;
6. Development of tools based on the chosen metamodelling framework and of SW artifacts to integrate existing tools with the new framework.

A metamodel provides a unifying framework in which to ensure and check model conformance. Steps 3. and 4., in fact, lead to an *instantiation* of the chosen metamodelling framework for a specific domain of interest, that for the ASM domain we called ASMETA (ASMs METAmodelling).

3.1 Language requirements analysis

We started collecting all material available on the ASM theory and tool support. We took [2] as reference for official documentation about the ASM theory.

Regarding the languages adopted by existing ASM tools[1], we observe that they all are ad-hoc extensions of the implementation languages (namely Gofer for AsmGofer, .Net languages for AsmL, C for XASM, ML for ASM-SL, Promela, SMV and PVS for model analysis tools), developed to encode an ASM formal model into the language of the implementation environment. Therefore, all these tool languages have syntaxes strictly depending on the implementation environment, provide different representations of ASM models, and have proprietary constructs which extend the basic mathematical concepts of the ASMs. This causes two main disadvantages for a practical and efficient use of ASMs in system development. First, a practitioner needs to map mathematical concepts, like ASM states (namely universes and functions defined on them), into types and structures provided by the target language; therefore, the process of encoding ASM models is not always straightforward and natural. Second, all these tools are loosely coupled and their integration is hard to accomplish.

[1] See http://www.eecs.umich.edu/gasm/tools.html for a list of ASM tools.

Metamodelling the ASM formal notation was the solution adopted in order to: (i) develop a unified notation, independent from any specific implementation syntax and allowing a more direct encoding of the ASM mathematical concepts and constructs, and (ii) tackle the problem of ASM tool inter-operability.

3.2 Choice of a metamodelling framework

As metamodelling framework, we initially chose the OMG MDA framework – the mainstream at the time we started [14], based on object oriented technology –, although many other platforms implementing the model-based development principles exist now, like the AMMA metamodelling platform, the Xactium XMF Mosaic initiative, the Software Factories and their Microsoft DSL Tools, the Model-integrated Computing (MIC), the Generic Modeling Environment for domain-specific modelling, and the Eclipse Modelling Framework.

We used the OMG's Meta Object Facility (MOF) as meta-language, i.e. as language to define metamodels, the model repository proposed by the MDR (Model Driven Repository) of NetBeans, and the OCL support provided by the OCLE tool.

Thanks to well-mastered model transformations, like the ATL-KM3 plugin which allows to move forward and backward in the EMF [3] and MOF modelling spaces, the choice of a specific metamodelling framework does not prevent the use of models in other different modelling spaces.

3.3 Language design by metamodelling

In a model-based approach, the abstract syntax of a language is defined in terms of a *metamodel*, which describes the vocabulary of concepts provided by the language, the relationships existing among those concepts, and how they may be combined to create models. In [1], a complete metamodel for ASMs is presented.

The *AsmM* (Abstract State Machines Metamodel) results into class diagrams developed using the MOF modeling constructs (classes, packages, associations).

We developed the metamodel in a *modular* and *bottom-up* way. We started separating the ASM static part represented by the *state*, namely domains, functions and terms, from the dynamic part represented by the *transition system*, namely the ASM rules. Then, we proceeded to model Basic ASMs, so reflecting the natural classification of abstract state machines. The whole and precise architecture came later, at the end of the metamodelling process (see Fig. 1).

The complete metamodel is organized in one package called `ASMETA` containing 115 classes, 114 associations, and 150 OCL class invariants, approximatively. The `ASMETA` package is further divided into four packages as shown in Fig. 1. Each package covers different aspects of the ASMs. The dashed ovals in Fig. 1 denote the packages representing the notions of *State* and *Transition System*, respectively. The `Structure` package defines the architectural constructs (modules and machines) required to specify the backbone of an ASM model. The `Definitions` package contains all basic constructs (functions, domains, constraints, rule declarations, etc..) which characterize algebraic specifications. The `Terms` package provides all kinds of syntactic expressions which can be evaluated in a state of an ASM. The `TransitionRules` package contains all possible transition rules schemes of Basic and Turbo ASMs. All derived transition rules are contained in the `DerivedTransitionRules` package. All relations between packages are of type *uses*.

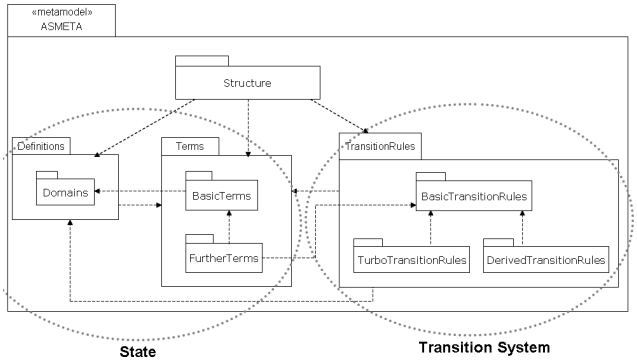

Figure 1. AsmM package structure

Each class of the metamodel is equipped with a set of relevant *constraints*, OCL (version 2.0) invariants written to fix how to meaningfully connect an instance of a construct to other instances, whenever this cannot be directly derived from the class diagrams.

AsmM is also available in the meta-languages AMMA/KM3 and in EMF/Ecore thanks to the ATL-KM3 plugin, which allows model transformations both in the EMF and MOF modelling spaces.

To define the *semantics* of the metamodel, we have established a semantic mapping from AsmM to a semantic domain where AsmM constructs take their meaning. The semantic domain is the first- order logic extended with a logic for function updates and for transition rule constructors formally defined in [2].

3.4 Language artifacts implementation

Whenever a language or formalism is specified in terms of a MOF-compliant metamodel, it is possible to automatically (or semi-automatically) generate several artifacts – here referred as *language artifacts* – from the metamodel by exploiting standard or proprietary projections from MOF to other technical spaces [11]. Fig. 2 shows, in particular, three projections that we used in order to generate:
– an XMI (XML Metadata Interchange) interchange format for ASMs; The main purpose of XMI is to provide an easy interchange of data and metadata between modeling tools.

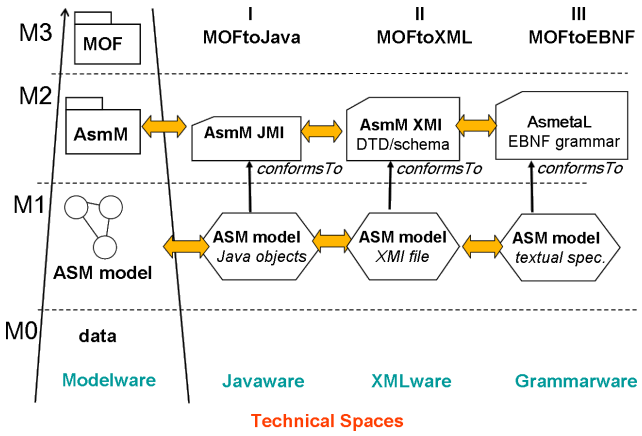

Figure 2. MOF Projections

– JMI (Java Metadata Interfaces) APIs for the creation, storage, access and manipulation of ASM models in a MOF-based instance repository; They can be used by tool developers to speed up the creation of tools supporting ASMs and by researchers to experiment algorithms over ASMs.
– a concrete textual notation, called AsmetaL (ASMETA Language), and its parser to effectively edit ASM models conforming to the AsmM metamodel.

From MOF to EBNF: derivation of a textual concrete syntax. A metamodel provides an abstract syntax for a language with the advantage of deriving (through well-established mappings or projections) different alternative concrete notations, textual or visual or mixed, to be used for different goals, all sharing the same language semantics.

From the AsmM metamodel we derived a concrete syntax, called *AsmetaL*, as a textual notation to write ASM models conforming to the AsmM. To this end, initially, we investigated the use of tools like HUTN (Human Usable Textual Notation) [9] or Anti- Yacc [7] which are capable of generating text grammars from specific MOF-based repositories. Nevertheless, we decided not to use them since they do not permit a detailed customization of the generated language and they provide concrete notations strongly reflecting the object-oriented nature of the MOF meta-language, while ASM is not an object-oriented formalism (even though it can model OO concepts). There are better MOF-to-grammar tools now, like xText of OpenArchitectureWare or TCS of AMMA, which we may consider to adopt in the future.

In [5], we defined general rules on how to derive a context-free EBNF (Extended Backus-Naur Form) grammar from a MOF-compliant metamodel, and we use these mapping rules to derive an EBNF grammar from the AsmM. The AsmetaL textual notation is the resulting language. It is completely independent from any specific platform and allows a natural and straightforward encoding of ASM models according to the AsmM metamodel (the abstract syntax).

AsmetaL consists of four parts reflecting the AsmM packages: the *structural language* providing the constructs describing the structure of an ASM, the *definitional language* providing a notation for basic ASM elements such as functions, domains, rules, and axioms, the *language of terms* providing syntactic expressions to be evaluated in an ASM state, and the *language of rules* providing a notation for transition rule schemes of an ASM.

We do not present here details of AsmetaL. Its complete EBNF grammar can be found in [1]. In [5], we also provided guidance on how to automatically assemble a script file and give it as input to the JavaCC parser generator to generate a parser for the EBNF grammar of the AsmetaL notation. This parser is more than a grammar checker: it is able to process ASM models written in AsmetaL, to check for their well-formedness with respect to the OCL constraints of the AsmM metamodel, and to create instances of the AsmM metamodel in a MDR MOF repository through the use of the AsmM-JMIs.

3.5 Language validation

The validation of a metamodel-based language requires the validation of the metamodel defining the abstract syntax of the language and of the metamodel derivatives (concrete syntaxes, APIs, etc.) representing the language artifacts.

The AsmetaL notation is the concrete syntax counterpart of the AsmM metamodel, therefore one way to validate the metamodel consists in validating the expressive power of AsmetaL w.r.t. ASM mathematical models, namely to test if AsmetaL is suitable to encode not trivial ASM specifications and if the encoding process of mathematical models is natural and straightforward. Furthermore, we validate the capability of AsmetaL to support the great variety of ASM specifications written in different ASM dialects. To this purpose, we encoded a great number of specifications (up to now we have about 140 ASM specification files in [1]), and we also asked to a non ASM expert to port in AsmetaL some specifications from [2] and other written in AsmGofer ans ASML. The task was completed within approximately three man months. Moreover, we evaluated the coverage of the metamodel obtained by our examples by instrumenting the parser of AsmetaL with EMMA, a free Java code coverage tool, and by parsing all the examples. We have checked that all the metamodel constructs are covered at least once.

Validating the language artifacts means to check if the AsmM-specific XMI and JMIs provide the desired global infrastructure for inter-operability and integration of ASM tools, as well as the suitable support in terms of specification language, abstract storage (i.e. the MOF-based model repository), APIs, interchange format, etc., to develop new ASM tools. To test the feasibility of tool inter-operability within the ASMETA framework and to evaluate the effort

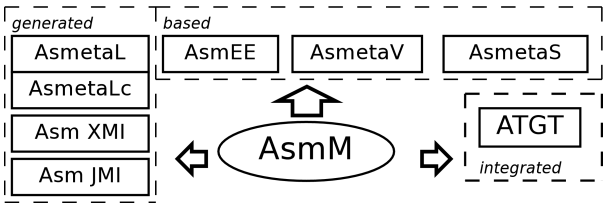

Figure 3. The ASMETA tool set

required to modify existing code, we made ATGT, an existing tool supporting test case generation for ASMs (written using the AsmGofer syntax), AsmM-compliant. To test the advantages offered by the ASMETA metamodelling framework to build new tools, we have developed a simulator for ASM models, called AsmetaS. It operates directly on instances of ASM models in the ASMETA repository, which is a metadata repository based on the MDR Netbeans libraries. AsmetaS reads ASM models in terms of JMI objects and, at every step, builds the update set according to the theoretical definitions given in [2] to construct the run of the model under simulation. Therefore, the AsmetaS development did not require to implement a parser, a type checker, and an internal representation of the model to simulate. The specification in the repository can be loaded from textual AsmetaL files by using the AsmetaL parser, but AsmetaS works regardless the way models are loaded in the repository.

3.6 Development and integration of tools

We take advantage of the metamodelling approach to develop a set of tools for ASMs – the ASMETA tool set available in [1] – based on the ASMETA framework.

The ASMETA tool set components (see Fig. 3) includes, among other things, a textual notation, *AsmetaL*, to write ASM models (conforming to the AsmM) in a textual and human-comprehensible form; a text-to-model compiler, *AsmetaLc*, to parse AsmetaL models and check for their consistency w.r.t. the AsmM OCL constraints; a simulator, *AsmetaS*, to execute ASM models; the *Avalla* language for scenario-based validation of ASM models, with its supporting tool, the *AsmetaV* validator; the *ATGT* tool that is an ASM-based test case generator based upon the SPIN model checker; a graphical front-end called *ASMEE* (ASM Eclipse Environment) which acts as IDE and it is an eclipse plug-in.

All the above artifacts/tools are classified in: *generated*, *based*, and *integrated*. Generated artifacts/tools are derivatives obtained (semi-)automatically by applying appropriate MOF projections to the technical spaces Javaware, XMLware, and grammarware. Based artifacts/tools are those developed exploiting the ASMETA environment and related derivatives; an example of such a tool is the simulator AsmetaS). Integrated artifacts/tools are external and existing tools that are connected to the ASMETA environment.

Our overall goal in developing the ASMETA tool set is to: (a) provide an intuitive modelling notation having rigourous syntax and semantics, possibly supporting a graphical view of the model; (b) allow modelling techniques which facilitate the use of the ASMs in many stages of the development process, and analysis techniques that combine validation (by simulation and testing) and verification (by model checking or theorem proving) methods at any desired level of detail; and (c) support an open and flexible architecture to make easier the development of new tools and integration with other existing tools.

4 Lesson learned

In general, we can identify some significant benefits we get in exploiting the metamodelling approach for language engineering. First, a metamodel could serve as *standard interlingua* establishing a common terminology to discriminate pertinent elements to be discussed during language design, and therefore, helps to communicate understandings, especially if – as in the case of the ASMs – the underlying language is a still evolving formal method and the community is too much heterogeneous to easily come to an agreement on the further development of the method itself.

Second, the metamodel-based approach has the advantage of being suitable to derive from the same metamodel (through mappings or projections) several artifacts (concrete syntaxes, interchange formats, APIs, etc.) – here also called language artifacts – which are useful to create, access, transform, manage and interchange models in a model-driven development context, settling, therefore, also a flexible object-oriented infrastructure for tools development and inter-operability. This framework allowed us to develop new tools like AsmetaS and AsmetaV with a very limited effort.

Third advantage, especially important for formal notations, is that people often claim that formal methods are too difficult to put in practice due to their mathematical-based foundation. In this direction, an abstract and (possibly) visual representation, like the one provided by a MOF-compliant metamodel, delivers a more readable view of the modeling primitives offered by a formal method, especially for people, like students, who do not deal well with mathematics but are familiar with the standard MOF/UML.

Furthermore, we like to remark that, although the task of defining a metamodel for a language is not trivial and its complexity closely matches that of the language being considered, the effort of developing from scratch a new EBNF grammar for a complex formalism, like the ASMs, would not be less than the effort of defining a MOF-compliant metamodel for the ASMs, and then deriving a EBNF grammar from it. Moreover, one has to consider the great possibility of being able to derive, from the same metamodel, different alternative concrete notations, textual or visual or

both, for various goals like graphical rendering, model interchange, standard encoding in programming languages, etc.

Finally, metamodelling allows to settle a "global framework" to enable otherwise dissimilar languages (of possibly different domains) to be used in an interoperable manner in different *technical spaces* [11], namely working contexts with a set of associated concepts, knowledge, tools, skills, and possibilities. Indeed, it allows to establishment precise *bridges* (or *projections*) among different domain-specific languages to automatically execute model transformations.

5 Related work and future directions

Concerning the metamodelling technique for language engineering, we can mention the official metamodels supported by the OMG for MOF itself, for UML, for OCL, etc. Academic communities like the Graph Transformation community [8, 15] and the Petri Net community [13], have also started to settle their tools on general metamodels and XML-based formats. A metamodel for the ITU language SDL-2000 has been also developed [4]. Recently, a metamodel for the AsmL language is available as part of a zoo of metamodels defined by using the KM3 meta-language [10]. However, this metamodel is not appropriately documented or described elsewhere, so this prevented us to evaluate it for our purposes.

Developing a grammar for the ASMs from the metamodel was challenging and led us to the definition of a bridge between grammars and metamodels as explained in [5]. This part of the process required at least six man month. Although we did not automatize these rules, since we wanted to derive only one grammar for AsmetaL, the rules could be easily reused for other formalisms.

Future work will include the integration of more existing tools and the development of new ones in the ASMEE IDE to better support *model evolution activities* [12] such as model refinement, model refactoring, model inconsistency management, etc. Today, only limited support is available in model-based development tools for these activities, but a lot of research is being carried out in this particular field, especially for language engineering, to establish synergies between model-based approaches and many other areas of software engineering including software reverse and re-engineering, generative techniques, grammarware, aspect-oriented programming, etc.

The definition of a means for specifying languages semantics rigorously and natively within their metamodels is currently an open and crucial issue in the model-driven context. What is required is a metamodelling environment sufficiently rich to express all syntactic and semantic aspects of a language. We believe this goal can be achieved by integrating metamodelling techniques with formal methods providing the requested and lacked rigour and preciseness. We think that the ASM formal method is a good candidate for this goal, and currently we have been working on defining a formal *semantic framework* to express the dynamic (possibly executable) operational semantics of metamodel-based languages in a multi-domain environment, by providing different techniques showing how the ASM formal method can be integrated, and in some cases promoted as metalanguage, with metamodel engineering environments.

References

[1] The Abstract State Machine Metamodel website. `http://asmeta.sf.net/`, 2006.

[2] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[3] Eclipse Modeling Framework. `http://www.eclipse.org/emf/`, 2008.

[4] J. Fischer, M. Piefel, and M. Scheidgen. A Metamodel for SDL-2000 in the Context of Metamodelling ULF. In *Fourth SDL And MSC Workshop (SAM'04)*, pages 208–223, 2004.

[5] A. Gargantini, E. Riccobene, and P. Scandurra. Deriving a textual notation from a metamodel: an experience on bridging Modelware and Grammarware. In *3M4MDA'06 workshop at the European Conference on MDA*, 2006.

[6] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[7] D. Hearnden, K. Raymond, and J. Steel. Anti-Yacc: MOF-to-text. In *Proc. of EDOC*, pages 200–211, 2002.

[8] R. Holt, A. Schürr, S. E. Sim, and A. Winter. Graph eXchange Language. `http://www.gupro.de/GXL/index.html`.

[9] OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08-01. `http://www.uml.org/`.

[10] F. Jouault and J. Bézivin. KM3: a DSL for metamodel specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, Bologna, Italy, 2006.

[11] I. Kurtev, J. Bézivin, and M. Aksit. Technical Spaces: An Initial Appraisal. In *CoopIS, DOA'2002, Federated Conferences, Industrial track*, Irvine, USA, 2002.

[12] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *International Workshop on Principles of Software Evolution (IWPSE'05)*, 2005.

[13] Petri Net Markup Laguage (PNML). `http://www.informatik.hu-berlin.de/top/pnml`.

[14] E. Riccobene and P. Scandurra. Towards an Interchange Language for ASMs. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines. Advances in Theory and Practice*, LNCS 3052, pages 111 – 126. Springer, 2004.

[15] G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In *J. Padberg (Ed.), UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques, satellite workshop of ETAPS*, 2001.