Language Engineering for Model-driven Software Development

Dagstuhl Seminar 04101
February 29th to April 5th 2004

Jean Bézivin[1] and Reiko Heckel[2]

[1] University of Nantes, France
Jean.Bezivin@sciences.univ-nantes.fr
[2] Universität Dortmund, Germany (on leave from Paderborn)
reiko@upb.de

Abstract. This paper summarizes the objectives and structure of a seminar with the same title, held from February 29th to April 5th 2004 at Schloss Dagstuhl, Germany.

1 Introduction

Model-driven approaches to software development require precise definitions and tool support for modeling languages, their syntax and semantics, their notions of consistency and refinement, as well as their mappings to the implementation level. In order to support model-driven development in a variety of contexts, we must find efficient ways of designing languages, accepting that definitions are evolving and that tools need to be delivered in a timely fashion.

In this respect, language definitions are not unlike software. Thus, a discipline of *language engineering* is required to support the design, implementation, and validation of modeling languages with the goal to deliver languages at low cost and with high quality.

An important contribution of any engineering science, besides the actual technology provided, is the meta knowledge about what are the relevant concerns to be addressed, what are the possible solutions, and what concern is best addressed in a given context by which kind of technology.

It is understood that different concerns of language engineering, like the definition of abstract syntax and well-formedness rules, operational and denotational semantics, consistency and refinement relations, and model transformations, will, in general, require technologies from different domains.

A framework for classifying, choosing, and relating different solutions domains is provided by the concept of *technological spaces* [KBA03]. A technological space is a working context with a set of associated concepts, body of knowledge, tools, acquired skills and possibilities, often associated to a given community. Well-known examples include XML, UML meta modeling, graph transformation, algebra and logic, programming languages, etc.

It has been the goal of the seminar to investigate relevant concerns and promising solution domains for language engineering, learn from specific solutions presented by the participants, and attempt a provisional classification and mapping. To illustrate problems and available solutions, a sample language engineering problem was proposed and elaborated.

After a more detailed discussion of the architectural aspect of language engineering, this summary presents this case study, discusses concerns and open issues raised by the corresponding language definition problem, and gives a more general motivation of technological spaces as solution domains for model-driven development.

2 MDA as Architecture of Processes and Languages

With its Model-Driven Architecture (MDA) initiative, the OMG has placed models into the center of their vision of software development. The claim is that, using UML models, "platform-independent application descriptions . . . can be realized using any major open or proprietary platform, including CORBA, Java, .NET, XMI/XML, and Web-based platforms" [Gro04]. Technically, the approach is based on model transformations for the automated refinement of platform-independent by platform-specific models, and code generation from the latter to the actual implementations.

As means of application integration, a model in MDA has a primarily architectural purpose (hence the name), even when it does not describe the architecture of a system. Consider, as a simple but typical example, a UML class diagram providing a platform-independent data model for an application structured according to a three-tier architecture into presentation layer, application logic, and data base access. To ensure the consistency of the data models used in the different layers, they shall all be derived from a single integrated model, i.e., the platform-independent class diagram.

In the course of development, the relevant parts of this diagram could be transformed into platform-specific models for, e.g., an XML schema, and a Java class structure, and a relational data base schema, in order to provide adequate representations for the data in different layers. From these platform-specific class diagrams the actual schemas or classes could be generated automatically.

In this case, the architectural aspect is not present in the models themselves, but reflected by the transformations of models and the consistency relations between them. In this sense, we think of the "A" in "MDA" as referring to the *architecture of the development process.*

As a consequence of the MDA goal of generating implementations from models automatically, the quality of these models (their level of precision and formality, their completeness and consistency) must be comparable with actual programs or formal specifications. However, a model can only be as good as the definition of the language to which it conforms. Thus, the first challenge consists in delivering high quality language definitions. Due to the necessary specialization of languages to different implementation platforms and application domains, this challenge manifests itself for a variety languages and dialects which, moreover, are continuously evolving. Hence, *reuse* is inevitable to reduce costs in defining and implementing languages. That means, languages (or their respective definitions) should be seen as "components" with "connectors" defined by consistency relations and transformations defined at the language level. This leads us to a view of MDA as an *architecture of languages*.

Continuing the architectural metaphor, a model used in a concrete development project, conforming to its language definition, can be seen in analogy to a concrete component (instance) in the configuration of a system, instantiating a generic component (type).

In the following section, we present an example of the first perspective, the architecture of the development process, by means of a sample approach to the model-based development of Web service processes. Section 4 is devoted to the second view of MDA as an architecture of languages.

3 A Sample Language Engineering Problem

We sketch a model-based approach to the development of Web service processes and discuss the concerns and issues raised by its definition and tool support.

3.1 Approach

A *Web service* is a software component that can be dynamically discovered, linked, and invoked by its clients via XML-based protocols. This software-oriented definition of the term can be contrasted with a business-oriented view, considering a Web service as a *business process*, implemented by the composition (and coordination) of simpler services provided by other businesses.

The composition of services provided by different independent parties, at both development time or runtime, requires a high degree of standardization and flexibility. Therefore, rather than hard-coding business processes in platform-specific programming languages which depend on certain compilers and runtime environments, platform-independent XML-based languages like the *Business Process Execution Language for Web Services (BPEL4WS)* [ACD$^+$03] are advocated. Such processes in XML representation can, at least in theory, be adapted at runtime, exchanged between different services, and executed on different standardized interpreters.

To support the development of BPEL processes in a model-based approach, we require

1. an intuitive and adequate *modelling notation* to allow precise specifications of processes at the conceptual level;
2. an *automatic transformation of process models to their XML-based encoding* to avoid the costly and error-prone task of deriving the implementation manually;
3. *techniques to analyze processes at the model level for syntactic and semantic properties* to avoid "debugging" the XML code.

These problems and requirements are prototypical for a wide variety of languages and platforms in the Web services domain and elsewhere. Therefore, instead of defining and implementing languages, transformations, and analysis tools for every single problem, reusable solutions are required.

In [HV04] we have presented an approach based on the combination of three such solutions: the *Unified Modeling Language (UML)* [Obj03] as standard notation for modelling software, *graph transformation* [Roz97] as meta language for defining model transformations, and a semantic interpretation of process models in terms of *Communicating Sequential Processes (CSP)* [Hoa78] which offers a language to express semantic consistency properties and tool support for analysis.

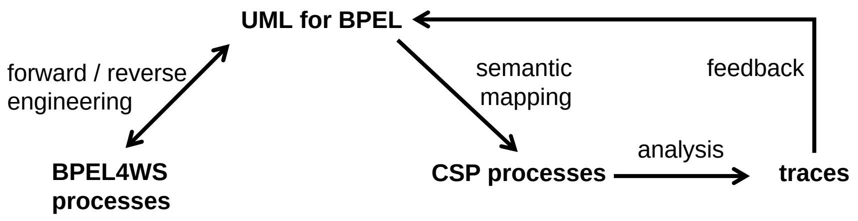

Fig. 1. Outline of the approach: languages and transformations

An outline of the approach is given by the diagram in Fig. 1, whose vertices are the languages by which processes may be represented, and whose edges represent uni- or bi-directional transformations between these representations.

3.2 Concerns and Open Issues

The example raises several concerns that are prototypical to model-driven approaches.

Syntactic and semantic extensions. The UML, as a general-purpose modelling language, provides a rich set of concepts to model all kinds a software system. However, to address the more specific concerns of a particular application domain or implementation platform the language needs to be specialized and extended. For this purpose, the standard [Obj03] foresees the extension mechanism of *profiles*, a compromise between desirable flexibility of the language and necessary compatibility with existing tools. We have to use profiles to tailor, in particular, the syntax of UML activity diagrams to the specification of BPEL4WS processes. However, this tailoring should not stop at the level of syntax, but continue to provide semantics to the new and specialized language constructs. To define semantics in an incremental, extensible way represents an problem that is not even completely solved for classical programming languages, but yet more relevant for the UML extension mechanism.

Model Transformations. In our example, model transformations occur in several places: the transformation of activity diagrams into BPEL4WS, the implementation language and into CSP, the language for behavioral analysis, as well as the transformation of traces, the result of CSP model checking, back into models. In many situations, two-way transformations are required, e.g., to support a *round-trip engineering* approach, where not only models are transformed into implementations (forward engineering), but also vice versa (reverse engineering), thus allowing incremental changes at both levels.

Moreover, for a transformation specification to be manageable and reusable, a modular approach is important which is structured in terms of the fundamental concepts of the domain. In this case, whenever a concept is added or modified, the corresponding transformation rules can be exchanged, hopefully without affecting the rest of the mapping specification. For example, in the domain of executable business processes, or workflow models, a corresponding concept analysis has produced an established list of *workflow patterns* [vdAtHKB03], a subset of which is supported by UML activity diagrams. In [HV04], we have given a mapping specification based on graph grammars which uses workflow patterns to organize the set of rules. However, this mapping is restricted to well-structured (essentially hierarchic) activity diagrams. It is open if a similar modularization can be supported in general.

Model-based Analysis. The final building block of our approach is the analysis of processes. Depending on the representation on which the analysis is performed, we distinguish between *syntactic* and *semantic* analysis. The former is often restricted to the evaluation of well-formedness constraints on (the abstract syntax of) the model which reveals inconsistencies in structural dependencies and typing.

Analysis of behavioral properties, instead, can hardly be done at the syntactic level, but requires a mapping of models into a semantic domain providing (1) a representation of the behavior to be analyzed, (2) means to express the desired properties, and (3) techniques and tools to check if these properties hold [EKGH01]. We have chosen the semantic domain of CSP [Hoa78] for this purpose, whose refinement relations are the basis for expressing properties over processes while tool support is provided by the FDR2 model checker [Ros97].

However, both syntactic and semantic analysis, should they be automated, are limited to those parts of the model that are completely formalized. Inscriptions in natural language, for example, can only be checked manually by a review process. Still, semi-formal models have their advantages over formal ones if they are used primarily by humans. The application of formal analysis techniques to incomplete and semi-formal models is an open problem.

Summarizing, the model-driven development problem for Web service processes as discussed in items $1-3$ above could be realized through the technological spaces supporting UML models, XML documents, and CSP processes and analysis. The mapping of MDA problems to existing solution domains is discussed more generally in the next section.

4 Mapping Problems to Solutions

In order to understand such mappings for the emerging field of model-driven software development, the main characteristics of both the problem and the solution domain need to be understood.

On the problem side, one of the main goals is the separation of business-oriented and platform-related parts. This shall allow to deal more easily with rapidly changing platforms, using generative techniques to map business-neutral descriptions onto specific execution platforms. But mappings may also apply in the reverse direction, extracting business models from legacy systems.

This description covers only part of the problem, focussing on a single aspects of the development process. The separation and subsequent integration of business and platform is part of the more general task of aspect separation and aspect weaving. For example, the separate expression and merging of functional and non-functional aspects are essential parts of the general problem space, too. For each aspect we need a domain specific language to express it in a user-oriented and non-ambiguous way, as well as a corresponding integration strategy.

On the solution side, there are different alternatives, too. For example, aspect separation and weaving may be done on a code-centric basis which leads to an AOP-like paradigm. Instead, it is also possible to use model-based techniques to handle the various aspects. Since a model captures only a specific view of the overall system, it is very natural to build a dedicated model representing a given aspect.

An understanding of this relation of *representation* between a system and its model is the basis of model-driven development. But the model itself is written in a given language which, in the MDA space, is defined by a metamodel. The relation between a model and its metamodel is a *conformance* relation, the precise specification of which is the task of the language designer. Similar organizations can be found in various technological spaces.

Systems are becoming more complex because of the increase in complexity (of code, data and aspects), in evolutivity and heterogeneity. In order to build these systems, many technologies are offered. Usually one will need to use several of these to solve a given problem. Unfortunately there is not and there will not be any uniformly superior technology because each one has its weak and strong points. In order to propose an agile method to problem solving in the domain of software system development and maintenance, the main characteristics of various technological spaces need to be identified and compared.

Some technological spaces offer better support for separation of concerns, for executability, for transformations, for modularity, etc. Among the classical spaces, we may mention model-driven engineering, XML document management, ontology engineering, programming languages and abstract syntaxes, graph theory and graph transformation, relational data base management systems, etc.

Each technological space is based on a central representation system (text files, trees, graphs, hypergraphs, etc.). This representation system may be explicitly defined, e.g., by a representation ontology or by a meta-metamodel (like the OMG MOF). This often gives rise to a three-level organization. In MDA the levels are called M3 for the MOF meta-metamodel, M2 for the metamodels, and M1 for the models. In programming languages, a similar organization holds with, e.g., EBNF at level M3, the syntax definitions of specific languages at level M2, and programs at level M1. In ontology engineering we have a corresponding structure with meta-level, intention, and extension ontologies. And in the XML technological space, this same layering could be exhibited again. We thus see that many technological spaces are similarly organized. This facilitates the construction of bridges between them. For example a brigde between MDA and XML is called XMI; a bridge between MDA and Java is called JMI; etc.

A Java program may also be considered as an XML document based on a given DTD (JavaML for example) or as an MDA model based on a Java metamodel, etc. These three representations in different technological spaces have each their advantages and drawbacks. When solving a given problem, the engineer should have a flexible and agile attitude towards which technology or mix of technologies to apply in order to solve this problem. The sample language engineering problem described in the previous section, for example, could be solved in specific instances and combinations of the programming language space, the XML document space, and the MDA space.

The study of language engineering problems from the perspective of various technological spaces is of conceptual and practical interest. At the heart of this study lies the investigation of how each space implements the basic structure required (the two relations of representation and conformance) as well as the investigation of mappings between spaces preserving that structure.

5 The Seminar

The program of the seminar was composed of contributions of about 40 participating researchers and practitioners. They were representative of the different technologies which contribute to a discipline of language engineering, i.e. UML and Metamodelling, Graph Transformation and Graph Grammars, CASE Tools, Aspect-oriented Software Development, and Programming Language Semantics.

In order to address the classical language engineering issues, discussion groups on *Syntax and Semantics*, *Model Transformation and Consistency*, *Tools*, and *Pragmatics* have been formed.

Reports on the results of discussions, abstracts of presentations, as well as work-in-progress papers describing new ideas developed at the seminar are published in these proceedings. A preliminary version of this summary has been published in the 7th edition newsletter of the European Association on Software Science and Technology (EASST).

References

ACD$^+$03. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana.

Business Process Execution Language for Web Services, Version 1.1, May 2003. `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/`.

EKGH01. G. Engels, J.M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In V. Gruhn, editor, *Proc. European Software Engineering Conference (ESEC/FSE 01), Vienna, Austria*, volume 1301 of *LNCS*, pages 327–343. Springer Verlag, 2001.

Gro04. Object Management Group. The architecture of choice for a changing world, MDA executive overview. `http://www.omg.org/mda/executive_overview.htm`, 2004.

Hoa78. C. Hoare. Communicating sequential processes. *Communicat. Associat. Comput. Mach.*, 21(8):666–677, 1978.

HV04. R. Heckel and H. Voigt. Model-based development of executable business processes for web services. In W. Reisig and G. Rozenberg, editors, *Proc. Advanced Course on Petri Nets, Eichstätt, Germany*, LNCS. Springer-Verlag, 2004. to appear.

KBA03. I. Kurtev, J. Bézivin, and M Aksit. Technological spaces: an initial appraisal. `http://www.sciences.univ-nantes.fr/lina/atl/publications/PositionPaperKurtev.pdf`, 2003.

Obj03. Object Management Group. Unified modelling language(UML) 2.0, 2003. `http://www.omg.org/uml`.

Ros97. A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

Roz97. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations.* World Scientific, 1997.

vdAtHKB03. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. *Distributed and Parallel Databases.* 2003.