Delft University of Technology
Software Engineering Research Group
Technical Report Series

Model-Driven Software Evolution: A Research Agenda

Arie van Deursen, Eelco Visser, and Jos Warmer

Report TUD-SERG-2007-006

TUD-SERG-2007-006

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: To be presented at the CSMR 2007 Workshop on Model-Driven Software Evolution (MoDSE), Amsterdam, 20 March 2007

Model-Driven Software Evolution: A Research Agenda

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
Arie.vanDeursen@tudelft.nl

Eelco Visser
Delft University of Technology
Delft, The Netherlands
visser@acm.org

Jos Warmer
Ordina
Nieuwegein, The Netherlands
Jos.Warmer@ordina.nl

Abstract

Software systems need to evolve, and systems built using model-driven approaches are no exception. What complicates model-driven engineering is that it requires multiple dimensions of evolution. In regular evolution*, the modeling language is used to make the changes. In* meta-model evolution*, changes are required to the modeling notation. In* platform evolution*, the code generators and application framework change to reflect new requirements on the target platform. Finally, in* abstraction evolution*, new modeling languages are added to the set of (modeling) languages to reflect increased understanding of a technical or business domain. While MDE has been optimized for regular evolution, presently little or no support exists for metamodel, platform and abstraction evolution. In this paper, we analyze the problems raised by the evolution of model-based software systems and identify challenges to be addressed by research in this area.*

1 Introduction

The promise of model-driven engineering (MDE) is that the development and maintenance effort can be reduced by working at the model instead of the code level. Models define what is variable in a system, and code generators produce the functionality that is common in the application domain. The problem with model-driven engineering is that it can lead to a lock-in in the abstractions and generator technology adopted at project initiation. Software systems need to evolve, and systems built using model-driven approaches are no exception. What complicates model-driven engineering is that it requires multiple dimensions of evolution. In *regular evolution*, the modeling language is used to make the changes. In *meta-model evolution*, changes are required to the modeling notation. In *platform evolution*, the code generators and application framework change to reflect new requirements on the target platform. Finally, in *abstraction evolution*, new modeling languages are added to the set of (modeling) languages to reflect increased understanding of a technical or business domain. While MDE has been optimized for regular evolution, presently little or no support exists for metamodel, platform and abstraction evolution.

In this paper, we analyze the problems raised by the evolution of model-based software systems and identify challenges to be addressed. We then outline a research agenda for addressing these challenges. The paper is written as a first step in our *Model-Driven Software Evolution* project, in which Delft University of Technology in collaboration with a number of industrial partners will seek out to resolve the most prominent of the issues raised in the research agenda discussed in this paper.

2 Model-Driven Engineering

Software engineering can be reduced to development of new software systems on the one hand and maintenance of existing ones on the other. The aim in development is to produce a high-quality system with the least possible effort. The aim in maintenance is to apply improvements and extensions with the least possible effort. Abstractions play a key role in development and maintenance. During development of a new system, the problem is to identify the right abstractions to represent high-level requirements and designs, which are then encoded in software. During maintenance, the problem is to identify the abstractions that are encoded in the existing implementation in order to correctly make modifications. With increasing distance between design and implementation, complexity increases, and productivity and maintainability decrease.

Model-Driven Engineering *Model-driven engineering (MDE)* is the unification of initiatives that aim to improve software development by employing high-level, domain-specific, *models* in the implementation, integration, maintenance, and testing of software systems [6]. To overcome the abstraction barrier, MDE introduces models that capture designs at a higher-level of abstraction. Unlike technical

documentation which has a fragile connection to the implementation of a software system, models are an integral part of the software evolution process. Developers represent designs using models that conform to an appropriate metamodel, which are then *automatically* transformed to implementations. Thus, with an appropriate modelling language, the effort of producing a new software system decreases and maintenance is reduced to model maintenance.

Prominent among the MDE initiatives is OMG's Model-Driven Architecture (MDA) [38, 31], in which software development is envisioned as a series of model transformation steps, which starts with a high level specification using a vocabulary that is familiar to the practitioners of the domain in question, and which ends with a platform-specific model describing how, for example, the system makes use of certain J2EE features. Related industrial efforts include Microsoft's DSL framework of *software factories* [29] for building stacks of domain-specific languages, and Jetbrain's Meta Programming System for *language oriented programming* [22, 26].

Domain-Specific Languages Model-driven engineering is strongly related to the field of *domain-specific languages* (DSLs) and *generative programming* [39, 4, 18, 20, 21, 37]. Domain-specific languages fill the gap between general purpose languages and particular application domains, by providing a language with notation and concepts geared to the domain. For example, database query languages provide concise notation for extracting data from a database, and regular expressions are the standard notation to formulate text searches. These notations encapsulate domain knowledge that cannot be expressed easily and effectively by programmatic means.

The need for domain-specific languages is clearly demonstrated by their presence in domains such as language recognition (YACC, ANTLR, SDF), graphics (SVG, VRML), querying (XQuery, SQL, XPath), text processing (Perl, Sed), document transformation (XSLT), mathematics (Mathematica, Matlab, Fortran, Fortress, Magma), and enterprise query languages (EJBQL for Enterprise JavaBeans, JDOQL for Java Data Objects, ODMG's OQL, Hibernate's HQL, JMS message selectors), to mention just a few. A DSL is described by a grammar that describes valid programs, a code generator that maps DSL programs to GPL programs, and a framework or run-time system that generated code makes use of.

Models and DSLs have complementary strengths. Models tend to be represented using a graphical notation, while DSLs often make use of a textual representation. Furthermore, models tend to be designed for describing structures, while DSLs are better at describing business logic. Since large scale adoption of model-driven engineering requires business logic as well as structure description, a unification of modelling and DSLs is needed.

What distinguishes MDE from the 'fourth-generation' and domain-specific languages of the past, is the aim at a systematic approach, with supporting technologies, to the construction of models *and* modeling languages such that these activities can be undertaken by the 'average' software developer and integrated in the software development process.

One of the challenges of realizing MDE is to unify the wealth of experience from work on code generation, domain-specific languages, and other generative programming techniques with the current momentum of the MDE initiative.

From frameworks to models Enterprise software is typically designed as a multi-tier system [25], which allows physical separation of parts of the system (layers are distributed over different machines), separation of concerns, focus of expertise, and code reuse. Technical domains (horizontal) correspond to the different implementation concerns such as user-interface, services, business logic, and data storage. Business domains (vertical) intersect the technical domains, since the implementation of a software system for application in some type of business affects design and implementation choices in all layers of the system. Development productivity of similar systems is increased by reuse through frameworks. In addition to capturing the common functionality in a class of systems, frameworks define their architecture. That is, frameworks provide a default system composition that application code only needs to customize using mechanisms such as inheritance, delegation, reflection, and/or inversion of control.

Further increase of productivity can be achieved by code generation, which is already used for development of some parts, such as the generation of an object-relational mapping from a data base schema. Model-driven engineering aims at extending this practice to the rest of the software stack. Application code is replaced by DSL programs, with the DSL capturing the variability in the framework, and code generators producing the application code automatically. While a generator may still produce code for use with a framework, this is not necessarily the case. The DSL, rather than the framework, defines the architecture and composition. It may well be profitable to generate more application code that is specialized to the application at hand using a trimmed down run-time system. An immediate advantage is the possibility of targeting a generator to a new architecture, without changing the DSL programs.

3 Challenges

While MDE promises to improve productivity and maintainability, widespread adoption and scaling to large soft-

ware systems requires research into evolution of model-based systems, scope and expressivity of modelling languages, and into interaction and integration of models.

Model-Driven Software Evolution Software evolution is concerned with the complete life cycle of software systems, from initial development to maintenance, and includes introducing new features, improving old features, and repairing bugs. While introduction of model-driven engineering brings advantages, it also requires a new style of evolution. In traditional software evolution, the development platform is fixed. Migration to a new platform is a sporadic event. The problem with model-driven engineering is that it can lead to a lockin in the abstractions and generator technology adopted when the project was started. Since an MDE platform hardwires many more architectural and design decisions than a traditional development platform, platform evolution is a requirement for MDE. Thus, MDE requires ***multiple dimensions of evolution***:

1. In *regular evolution*, the modeling language is used to make the changes; the development platform, that is, the set of domain-specific and general-purpose languages, is fixed.
2. In *meta-model evolution*, changes are required to the modeling language to improve its expressivity. Such changes may require migration of models.
3. In *platform evolution*, the underlying infrastructure, such as the code generators and the application framework, is required to change, because of new requirements in the target platform. Existing models may remain unaffected by such changes, if the modeling language abstracts over the specifics of the target platform.
4. In *abstraction evolution*, new modeling languages are added to the set of (modeling) languages to reflect increased understanding of a technical or business domain. After introducing new languages, the old system should be migrated to make use of it.

Our first fundamental premise for model-driven software evolution is that evolution should be a continuous process. Software development is a continuous search for recurring patterns, which can be captured using domain-specific (modeling) languages. After developing a number of systems using a particular meta-model, new patterns may be recognized that can be captured in a higher-level or richer meta-model. Our second premise is that reengineering of legacy systems to the model-driven paradigm should be a special case of this continuous evolution, and should be done incrementally. While MDE has been optimized for regular evolution, presently little or no support exists for continuous meta-model, platform, and abstraction evolution.

Scope and Expressivity Modeling languages tend to be geared to description of structure and have limited capabilities for *expressing business logic*. The field of domain-specific languages provides a source of inspiration for design of richer modeling languages. Here it is important to find a good balance between expressivity and scope. If modeling languages become too general they loose their advantages with respect to general purpose languages.

Another issue with existing modeling languages is a lack of *modularity*. Visual modeling languages in particular have limited modularity (e.g., entire model in one diagram). In order to scale to modeling of large systems, models should be developed as independent components that can be integrated in different compositions. Furthermore, it should be possible to provide different views on models and compositions of models. For instance, an architectural view that shows the composition of a system from components, or a data-flow view that shows the data dependencies in the system.

Interaction and Integration Modeling languages are typically not designed to model an entire system at once. In order to be cost effective, a language should be usable in many different systems. Therefore, a modeling language captures aspects of a software system in some *technical* or *business* domain, and the implementation of a complete system will consist of models in many different modeling languages as well as some code in a GPL.

Thus, model-driven engineering departs from traditional software engineering, with its mostly monolithic development platform. Instead of one or a few programming languages, MDE development introduces a multitude of languages that are themselves artifacts of the development process. This has numerous implications for the integration of models in the development environment and their interaction with that environment:

- *Interaction between models and code.* Models need to interact with the code defining the rest of the application. In particular, when incrementally introducing models in a legacy system, models need to interact with legacy code. Modeling languages do not necessarily cover all corner cases. It is not uncommon in such cases to *customize generated code*, a disaster for maintainability. Instead it is necessary to be able to customize and enrich *models* with application specific code *without* resorting to modifying generated code. For the same reason, models need to provide an interface to the underlying framework.

- *Interaction between models.* Since separate models are used to define different parts of a system, their integration requires model interaction, possibly between different modeling languages. Models at different layers of the software stack need to interact, e.g. a user interface model refers to a data model. Models for business domains need to interact with models in technical domains. This requires that modeling languages define an interface through which models can be approached.

- *Interaction with the development environment.* The development and build environment needs to be aware of models and modeling languages and provide the same level of support as for regular languages. In particular, the definition of a modeling language should have the same status as other programming artifacts.

4 A Research Agenda

In this section we identify four research themes for addressing the challenges introduced above. The first theme, the construction of *model development environments* is concerned with investigating and integrating the basic techniques needed for realizing model-driven software evolution. The second theme, *from model to code*, is concerned with the investigation of modeling language designs for expressing business logic and the interaction between models and code. The third theme, *from code to models*, is concerned with deriving model abstractions from (legacy) code. Finally, the fourth theme is concerned with *evaluation* of the methods and techniques for model-driven software evolution.

4.1 Model Development Environment

An important goal is to make it possible to easily create new modeling languages as part of software development. This requires a *model development environment* with model processing techniques that build on the advanced analysis and transformation techniques produced in programming language research.

Connecting model representations and meta-models. Infrastructure for defining the syntax and structure of modeling languages has a central role in a model development environment. The basic infrastructure for meta-modelling is widely available, but requires integration and connection of existing formalisms such as MOF, EBNF, SDF, and XML Schema, by mappings at the meta-model and model level. Here inspiration can be drawn from previous work in syntax definition, exchange formats, pretty-printing and generic mappings between languages, e.g. [9, 41, 8, 30, 7, 10]. The next objective will be to address grammars and meta-models in the context of multiple languages that refer to each other or are combined into new languages. Important steps in this direction are provided by the work on modular parametric syntax definition [41], embedding of domain-specific languages [15], and recent experiments with grammar mixins [**?**].

Unifying model and code transformation. Model-driven software evolution requires many types of transformations. Transformations from model to code are used to implement modeling languages. Transformations from code to models are used to extract models from (legacy) code. Transformations from models to models are used to refactor models, to migrate models to a new modeling language, or to map higher-level models to lower-level models. Preferably all these transformations should be expressed using the same transformation language. Concepts from high-level program transformation languages such as strategies and dynamic rules in Stratego [45, 12, 13] can provide inspiration for the design of a high-level transformation language that can be used to transform models as well as code.

Integrating modeling language definitions in the development environment. MDE essentially proposes modeling languages as new units of abstraction. Thus, rather than providing a fixed set of meta-models and corresponding generators, introducing a new modeling language should be just as easy as, say, introducing a new class in Java. The programming environment should support the definition of new languages consisting of a grammar (meta-model), a transformation mapping models to implementations, possibly a framework for use by target code, and declaration of an interface for connecting models to other models and code. Furthermore, a language definition may come with transformations for extraction of models from code, or for refactorings. After defining a new language, the programming environment supports the creation of models in the language.

4.2 From Model to Code (Generation)

The objective of this theme is to arrive at methods and tools that support the integration between traditional development and generative development. The main question is what are good design criteria for modeling languages and their mapping to implementations; the main issues are expressivity and interaction.

Modeling business logic What are good language designs for modeling business logic? The full generality of a general purpose language provides great expressivity, but provides little guidance in or knowledge of the application

domain. On the other hand, a more restricted and to the point language may provide excellent expressivity in the domain it supports, but is usually not very flexible as soon as something slight out of the ordinary is needed. How do we find a good balance between generality and domain-specificity? How can existing (proprietary) DSL-based solutions be mapped to the (standards-based) MDE domain [23, 24]?

Model interaction In order to achieve modularity we need separation of concerns into different models, which implies dependencies between models. Furthermore, a system may only be partially defined by models, with the rest defined with GPL code, which implies dependencies between models and code. Rather than depending on conventions that may easily get out of synch, these dependencies should be formalized and checkable *before* code generation [27]. Thus, a model should publish an *interface* that custom code and other models can use to interact with the model. Such an interface may be declared explicitly as part of the model, or it may be derived from the model specification automatically. These requirements also hold for models that define different *views* on the same (aspects of a) system.

Modeling composition Modeling languages usually model a particular, frequently occurring aspect of sofware systems. This promotes their reuse. Composition of complete systems requires modeling lanuages that define compositions of models.

4.3 From Code to Model (Evolution)

The objective of this theme is to arrive at methods and tools that support the evolution of MDE-infrastructures, which may affect the underlying meta-models, code generators, and application frameworks, as well as the existing models that need to be migrated to the new infrastructure. A special case is the migration of legacy applications to an MDE approach, which we will consider as a sequence of small MDE evolution steps.

Incremental model introduction Methods and techniques are needed that make it possible to move certain concerns to the model-based paradigm, leaving the rest of the code in its current state. Our approach will be based on our earlier experience in mixing domain-specific languages (DSLs) with existing languages [15, 11], and our earlier work in migrating concerns implemented using coding idioms to an aspect-oriented DSL solution [16]. A particular issue to be resolved is how to deal with (data) dependencies between concerns.

Model reconstruction To migrate legacy systems to model-based systems, methods are needed to reconstruct or *harvest* models from existing code [40]. We distinguish two directions. In *agnostic* reconstruction, we search for recurring patterns in the source code, and investigate if these can be obtained through generation from a more concise model specification. As an example, in our earlier work we have investigated how clone detection or metrics can be used to identify certain crosscutting concerns such as logging and error handling [17, 36]. A second direction investigates how known model types can be reconstructed. The models can be generic, such as state machines, but also domain-specific, requiring domain-specific reconstruction techniques [28].

Model-based testing before and after reengineering Replacing a concern by a model-based concern and introducing code generation for that concern, requires validation to ensure that the application still works as before. We will investigate what testing techniques can be used to ensure that the system's functionality has not altered, and that faults likely to have been introduced in the reengineering are found in the most effective manner. An issue is how we can express the adequacy of a test suite in terms of the legacy concern implementation as well as in the target model representation.

4.4 Evaluation

While MDE holds much promise in reducing the costs of maintenance, there is presently no empirical evidence that this is indeed the case. Thus, besides research into technological solutions, we need to analyze under what circumstances incremental model-based software evolution is a viable idea, determine factors for success and counter-indicators, and demonstrate benefits in practice.

While MDE offers a vision on how software engineering should be done, there is little guidance on the *decision making process* to be followed when adopting model-driven techniques for existing systems. It would be useful to develop, based on experience from case studies, a set of guidelines for use in decision making, comprising indicators for adopting MDE for a given system, metrics for monitoring progress, guidelines for choosing technologies, analysis of the impact of MDE on lifecycle management, and methods for ensuring that the system's behavior is not altered when we improve its structure using MDE technologies.

5 Related Work

Work in some of the areas as suggested by our research agenda is already taking place. We briefly discuss a selection of the most relevant recent developments.

Technological Spaces Kurtev et al. [33] coin the term "technological space" for a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. MDA and "grammarware" are two example technological spaces they mention. The different characteristics of technological spaces can motivate the move from one technological space to another, which requires "bridges" between these spaces. Such bridges will be, e.g., needed in (incremental) platform and abstraction evolution, when new code generator technologies and new modeling notations are introduced.

A very generic framework for bridging between technological spaces is discussed by Wimmer and Kramler [48]. In this framework a compiler-compiler is used that, based on an attributed grammar mapping EBNF to MOF, generates a so-called grammar-parser. This grammar-parser not only transforms EBNF grammars into MOF-based metamodels, but also generates a tool to transform programs associated with that EBNF grammar into models associated with the generated metamodel. Model-driven software evolution requires a similar approach to establish a bridge between some of useful and earlier developed tools that are grammar-based and MOF-based meta-models such as the UML.

Language Workbenches and Metamodeling Infrastructures The *Model Development Environment* we intend to develop is strongly related to the ongoing *language oriented programming* [22], *language workbenches* [26], and *software factories* [29] initiatives. Jetbrain's prototype MetaProgramming System (MPS) for *language oriented programming* [22] allows developers to extend the base programming languages with new, domain-specific extensions and a mapping to the base language.

The *software factories* [29] of Microsoft add to the development environment *DSL Tools* that support the construction of *visual* domain-specific languages. This consists of support for creating a visual editor from a language definition, and definition templates for code generation.

Neither of these frameworks provide support for transformation and reverse engineering. To facilitate this, infrastructures such the ASF+SDF MetaEnvironment [8], TXL [19], DMS [3], and Stratego/XT [42, 30, 44, 12] supporting the modular definition of domain-specific and general-purpose languages and transformations on those languages will be necessary as well. A challenge to address is to apply the extensive experience obtained in the grammarware domain to the realm of models.

Language Combination and Interaction Whereas currently languages interact implicitly at the level of the generated code, model-driven software evolution requires explicit language interactions at the right level of abstraction.

This calls for a combination of languages, with which we have already experimented in the context of embedding of domain-specific languages [15, 11] and concrete syntax for code generator templates [43, 14]. A particular challenge is finding appropriate mechanisms for extensibility of program transformations for language extensions [46].

Reengineering to MDA The OMG is currently working on a set of standards related to the modernization of existing systems, referred to as Architecture-Driven Modernization (ADM) [1]. These standards effectively define a set of metamodels that allow for modeling the information required for modernization efforts. One of these standards is based on Mansurov and Campara's Container Models that they propose as a way to guide software maintenance [35]. Where possible we will reuse (or perhaps even contribute to) relevants elements of the ADM meta-models.

The OMG discusses a number of scenarios in which ADM could be applied [2]. One of those scenarios involves the migration towards the MDA. For this scenario it is assumed that the starting point is a situation were software development is not model-driven. In practice, a hybrid setting in which the starting point is partly model-driven but using proprietary or obsolete technology that fails to meet the demands of modern (web based) applications are likely to occur as well. Initial results in addressing such situations have been presented, in which model transformations themselves are used to automate parts of the migration [24, 40].

Roundtrip Engineering We are *not* considering roundtrip engineering as in Fujaba's 'from models to code and back' [32]. Roundtrip engineering assumes editing of generated code, which poses restrictions on code generation (generated code should be invertible) and run a great risk of divergence between model and code.

Instead we assert that generated code should *never* be modified, and that models should be customized 'from the outside', and that customization should either be considered by the generator or be plugged into the generated code.

A challenge that must be addressed that such customizations should not depend on knowledge of the generated code, in order to enable retargetability. Furthermore, our transformations *from code to model* are "apply-once" model extraction transformations that may become much more complicated than one-to-one mappings common in rountrip engineering.

Unifying Code Generation, Reverse Engineering, and Model Transformation The key innovation that model-driven software evolution requires is a unification of reverse engineering, program transformation, code generation, and model transformation techniques. The vision is

that programmers using the model development environment can develop code generators, as well as transformations for model-to-model migrations, model harvesting, and code-to-model extraction. Currrently transformation and analysis frameworks tend to focus on one or the other application.

Frameworks such as Fujaba, ATL and the proposed QVT are biased to model transformations, and do not effectively support program transformations as well. Likewise, many reverse engineering activities, such as Rigi [49], Crocopat [5], Bauhaus[1], or CodeCrawler [34] (to name a few), are effective in recovering lost structures, but have little support for transformations.

There is a lot to be gained from combining and unifying these various initiatives. One of our routes will be to take our earlier program transformation and reverse engineering work as starting points. We believe that the unifying approach to program transformation developed in the Stratego/XT project [47, 44, 13] consisting of rewriting with programmable rewriting strategies, generic traversals, separation of rules and strategies for reusability and separation of concerns in transformations, and dynamic rules for context-sensitive transformation is a valuable contribution to the field of model transformation, and provides solutions for many of the requirements in the QVT proposal.

6 Perspective

In the late 80s and early 90s, significant investments in 4th generation languages and code generators were made by many organizations. 4GL was considered an important way to speed up time to market, and to reduce the costs of software development. Unfortunately, 4GL could not deliver the value that was promised: in many cases, the use of a 4GL meant a lockin in proprietary code generation technology, and in abstractions and modeling languages that that were less and less suitable for the new demands imposed by the environment in which the application had to run.

To date, model-driven architectures are generating a similar amount of interest as 4GL, again in order to reduce development effort and increase software quality. The strong backing of large companies and the fact that many of the proposed techniques are based on open standards, help to avoid the problem of being restricted to proprietary technology. Nonetheless, the problem remains that for systems with a long life span, it will be essential to be able to alter the underlying code generators, frameworks or the modeling notation used. Developing methods, tools and techniques for exactly this problem will allow the modernization of existing systems and the incremental adoption of model-driven techniques.

[1] www.bauhaus-stuttgart.de

In this paper, we have analyzed the underlying issues and challenges, leading to a research agenda for model-driven software evolution.

Acknowledgments Partial support has been received from NWO Jacquard, project 638.001.610 “Model-Driven Software Evolution” (MoDSE). We would like to thank the MoDSE partners and Martin Bravenboer for fruitful discussions on the MoDSE project plan, which resulted in this research agenda.

References

[1] Architecture-Driven Modernization Task Force. Architecture-Driven Modernization (ADM). http://adm.omg.org/, 2006.

[2] Architecture-Driven Modernization Task Force. Architecture-Driven Modernization scenarios. Technical report, OMG, 2006. http://adm.omg.org/ADMTF_Scenario_White_Paper(pdf).pdf.

[3] I. D. Baxter, C. Pidgeon, and M. Mehlich. Dms: Program transformation for practical scalable software evolution. In *Int. Conf. Software Engineering (ICSE)*, pages 625–634. IEEE, 2004.

[4] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[5] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Software Eng.*, 31(2):137–149, 2005.

[6] J. Bézivin. On the unification power of models. *Software and Systems Modelling*, 4(2):171 – 188, May 2005.

[7] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.

[8] M. G. J. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-Environment: a component-based language laboratory. In R. Wilhelm, editor, *Compiler Construction (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–368, Genova, Italy, April 2001. Springer-Verlag.

[9] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996.

[10] M. Bravenboer. Connecting XML processing and term rewriting with tree grammars. Master's thesis, Utrecht University, Utrecht, The Netherlands, November 2003.

[11] M. Bravenboer, R. de Groot, and E. Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In *Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, Braga, Portugal, July 2005.

[12] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.16. components for transformation systems. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, pages 95–99, Charleston, South Carolina, January 2006. ACM SIGPLAN.

[13] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69:123–178, 2006.

[14] M. Bravenboer, R. Vermaas, J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In R. Glück and M. Lowry, editors, *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *Lecture Notes in Computer Science*, pages 157–172, Tallinn, Estonia, September 2005. Springer.

[15] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383. ACM Press, October 2004.

[16] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating idiomatic crosscutting concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, pages 37–46. IEEE Computer Society, 2005.

[17] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying cross cutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[18] J. C. Cleaveland. Building application generators. *IEEE Software*, pages 25–33, July 1988.

[19] J. Cordy. Source transformation, analysis and generation in TXL. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, pages 1–11, Charleston, South Carolina, January 2006. ACM SIGPLAN.

[20] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 1999.

[21] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[22] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains 'onBoard*, November 2004. `http://www.onboard.jetbrains.com/is1/articles/04/10/lop/`.

[23] D. Doyle. Transforming proprietary domain-specific modeling languages to model-driven architectures. Master's thesis, Delft University of Technology, 2005. URL: `swerl.tudelft.nl`.

[24] D. Doyle, H. Geers, B. Graaf, and A. van Deursen. Migrating a domain-specific modeling language to MDA technology. In J. M. Favre, D. Gasevic, R. Lammel, and A. Winter, editors, *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ateM 2006)*, pages 47–54. Johannes Gutenberg-Universitat Mainz, 2006.

[25] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[26] M. Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* ThoughtWorks, June 2005. `http://www.martinfowler.com/articles/languageWorkbench.html`.

[27] B. Graaf and A. v. Deursen. Model-driven consistency checking of behavioural specifications. In *Proceedings Fourth International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*. IEEE Computer Society, 2007.

[28] B. Graaf, S. Weber, and A. van Deursen. Model-driven migration of supervisory machine control architectures. *Journal of Systems and Software*, 2007.

[29] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.

[30] M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Perigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2001.

[31] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley, 2003.

[32] T. Klen, U. A. Nickel, J. Niere, and A. Zndorf. From uml to java and back again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, September 1999.

[33] I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: an initial appraisal. In *Confederated International Conferences CoopIS, DOA, and ODBASE 2002*. Springer-Verlag, 2002. Industrial Track.

[34] M. Lanza and M. Radu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[35] N. Mansurov and D. Campara. Managed architecture of existing code as a practical transition towards mda. In *UML Modeling Languages and Applications: ≪UML≫ 2004 Satellite Activities*, volume 3297 of *Lecture Notes in Computer Science*, pages 219–233. Springer-Verlag, 2005.

[36] M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 2007.

[37] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[38] J. Miller and J. Mukerji, editors. *MDA Guide Version 1.0.1*. OMG, 2003.

[39] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–74, Sept. 1984.

[40] T. Reus, H. Geers, and A. van Deursen. Harvesting software systems for MDA-based reengineering. In *European Conference on Model Driven Architectures: Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 213–225, 2006.

[41] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[42] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques*

and Applications (RTA'01), volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[43] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[44] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[45] E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

[46] E. Visser. Transformations for abstractions. In J. Krinke and G. Antoniol, editors, *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 3–12, Budapest, Hungary, October 2005. IEEE Computer Society Press. (Keynote paper).

[47] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[48] M. Wimmer and G. Kramler. Bridging grammarware and modelware. In *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops*, volume 3844 of *Lecture Notes in Computer Science*, pages 159–168. Springer-Verlag, 2006.

[49] K. Wong, S. Tilley, H. Müller, and M.-A. Storey. Structural redocumentation: a case study. *IEEE Software*, 12(1):46–54, 1995.

TUD-SERG-2007-006
ISSN 1872-5392