Harvesting Software Systems for MDA-Based Reengineering

Thijs Reus[1], Hans Geers[2], and Arie van Deursen[3]

[1] Interactive Objects, Freiburg
Thijs.Reus@interactive-objects.com
[2] Delft University of Technology, The Netherlands
H.J.A.M.Geers@ewi.tudelft.nl
[3] Delft University of Technology and CWI, The Netherlands
A.vanDeursen@ewi.tudelft.nl

Abstract. In this paper we report on a feasibility study in reengineering legacy systems towards a model-driven architecture (MDA). Steps in our approach consist of (1) parsing the source code of the legacy system according to a grammar; (2) mapping the abstract syntax trees thus obtained to a grammar model that is defined in the Meta-Object Facility (MOF); (3) using model to model (M2M) transformations to turn the grammar model into a generic meta-model, called GenericAST, in which information about software systems can be stored in a language-independent way; (4) mapping the GenericAST models, again using M2M transformations, to UML models that can be either used for code generation or for documentation purposes. The steps have been implemented in a prototype model harvesting tool that is based on ArcStyler, the MDA environment provided by Interactive Objects. Our paper presents this approach, and reports on our experiences in applying the method to a 178 KLOC production system from the insurance domain written in PL/SQL.

1 Introduction

Model Driven Architecture (MDA) provides a promising basis for keeping software maintainable by using a series of models in the development process: models are the main software assets, as opposed to source code. In this paper we explore how MDA concepts can be applied to existing software systems. The key problem here is that usually no adequate models of actual systems are available. In order to overcome this gap, we investigate to what extent reverse engineering techniques can be used to extract adequate models from source code.

It is very unlikely that a fully automatic approach will ever be able to reconstruct models that are (1) at an appropriate level of abstraction; and (2) can be used to (re)generate the full functionality of the original application. Therefore, we will aim at the interactive reconstruction of models that serve the following purposes:

– The models can be used for system understanding and software exploration in order to support a transition to a model-driven reimplementation;

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 213–225, 2006.

– The models can be used to generate code templates, that can subsequently be refined to include deeper application knowledge.

A distinctive characteristic of reengineering to an MDA context is that MDA provides a flexible environment for manipulating models, using open standards such as the Meta-Object Facility (MOF) [20], transformations using the Query/View/Transformation approach (QVT) [19] and the Unified Modeling Language UML. For that reason, we will investigate a reengineering approach which switches to the MDA "technological space" [15] as quickly as possible, after which model to model transformations are used to refine the initial raw results.

The work presented in this paper was carried out within a pilot study conducted at a major Dutch insurance company. The objective of this pilot study is to investigate the feasibility of adopting MDA techniques for their legacy systems, in order to safeguard the future maintainability of these systems. Within the study, tools were built to reverse engineer models from code —a process called "harvesting" — and these tools have been applied to the source code of a production system written in PL/SQL. Steps in our approach consist of

1. Parsing the source code of the legacy system according to a grammar;
2. Mapping the abstract syntax trees thus obtained to a grammar model that is defined in the Meta-Object Facility (MOF);
3. Using model to model (M2M) transformations to turn the grammar model into a generic meta-model, called GenericAST, in which information about software systems can be stored in a language-independent way;
4. Mapping the GenericAST models, again using M2M transformations, to UML models that can be either used for code generation or for documentation purposes.

The paper is structured as follows. We start out with a survey of related work in the area of reengineering to model-driven architectures. We then describe our approach (Section 3) as well as the prototype workbench we developed to support our approach (Section 4). The application of our approach to the PL/SQL production system is described next (Section 5) followed by a discussion of lessons learned (Section 6). We conclude by summarizing our main contributions as well as suggestions for future work.

2 Related Work

The Object Management Group is actively involved in the "reverse engineering to MDA" area, with the Architecture Driven Modernization (ADM) task force [11, 22]. A total of seven Requests for Proposal aim at standardizing an extensive reverse engineering framework towards an MDA target environment. Part of ADM are two generic intermediate models, for supporting analysis and refactoring. The generic models we use are inspired by but significantly simpler than the wide spectrum ADM models.

Mansurov and Campara [16] argue that a first step in the migration towards the MDA is the introduction of modeling in the software development process. They propose an approach to raise the maturity of software architectures to a level where software maintenance and evolution are driven by the architecture instead of by the code. For this they introduce so-called Container Models. They focus on the extraction of these Container Models from existing code.

A framework for language neutral representation of source code is presented by Al-Ekram and Kontogiannis [1]. A generic abstract syntax tree (AST) is part of the program representation framework. XML is used as the main language. It has the advantage of being a light weight solution, but comes with meta-model discovery problems. The meta-model must be hardwired into the programs that use it.

A view on language support for MDA also requiring a generic representation is discussed by Cepa and Mezini [4]. They propose a *generic annotated abstract syntax tree* that can be used to support domain-specific but platform-independent models. An explicit meta-representation is advocated for programs in an AST-like structure, together with the possibility for users to add their own annotations to this AST using a dedicated language.

Boronat *et al.* present a framework for automatic legacy system migration in MDA [3], using rewriting logic as their transformation engine. The results are UML models of the legacy system.

An approach aiming at incremental adoption of model-driven technologies is provided by Gannod and Carey [13], who rely on Java annotations that support the creation of models that fit in the Eclipse Modeling Framework EMF.

Harvesting MDA models from existing proprietary *models* is discussed by Doyle [9]. His starting point are models as used in a 4GL application generator also in use at Fortis. His approach involves reconstructing and normalizing models from database representations, which are subsequently transformed into MOF representations using EMF.

Reengineering from code towards the MDA involves a combination of parsing and model transformation. Kurtev et al. [15] refer to this as building bridges between *technological spaces*, in this case between the grammar-ware and MDA spaces. A very generic framework for bridging between technological spaces is discussed by Wimmer and Kramler [24]. 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.

A general introduction to reengineering and system renovation is provided in [5,7]. Reengineering generally consists of a series of *reverse engineering* steps that reconstruct system representations at a higher level of abstraction. These representations can subsequently be used for code generation. Since fully automated reengineering often is not feasible, much reverse engineering research focuses on supporting *system exploration*, i.e., helping software engineers in

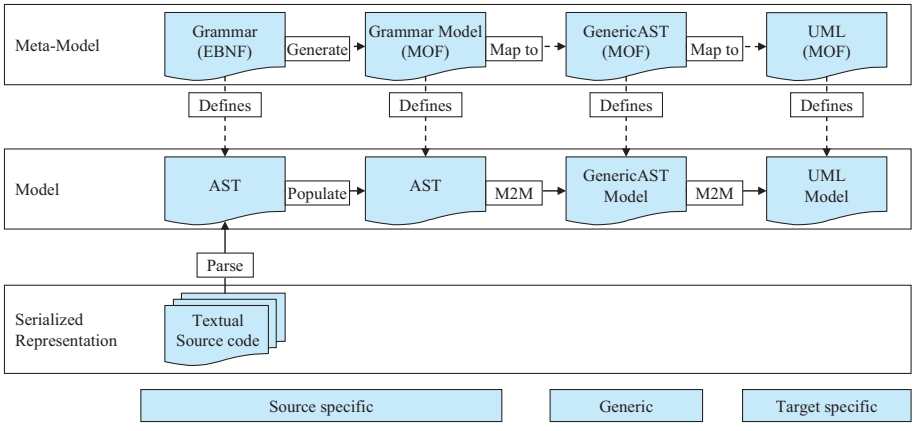

Fig. 1. Reengineering Framework

understanding the legacy system at hand [17]. A reverse engineering process tailored towards reconstructing software architectures from source code is provided in [6].

3 Harvesting Approach

The approach we used for harvesting models from the source code of an existing application is depicted in Figure 1. It consists of the following steps.

The starting point is a grammar of the legacy language, expressed in an EBNF-like formalism. We first of all use this grammar to generate a parser capable of processing the system's source code and representing it by means of abstract syntax trees (ASTs).

Existing parser generators generally produce proprietary AST representations. In order to benefit from standards and available tool support from MDA technologies, we therefore need to transfer these abstract syntax trees to a MOF-based representation. To that end, we use the EBNF grammar to generate:

1. A MOF-based *Grammar Model*, i.e., a metamodel of the Grammar defined with MOF, that offers a one-to-one mapping between EBNF-based ASTs and MOF-based ASTs; and
2. A series of transformations coded in Java mapping EBNF-based AST nodes to their Grammar Model counterparts.

Following the terminology of Kurtev *et al.*, we thus switch from the grammarware technological space to the MDA technological space [10, 15].

Our next step consists of mapping the source language abstract syntax trees to a generic, domain-independent model, which we have dubbed *GenericAST*. From this generic model, we subsequently generate models that can be used either for documentation or for code generation purposes. In some cases these

models will be based on UML, whereas in other situations these models will be domain specific. The main reason for introducing such a generic layer is that it increases opportunities for reuse when, for example, extending the framework with additional source languages.

The GenericAST meta-model is based on UML, especially for representing structural information such as containers, entities, features, constraints, or types. Regarding behavioral constructs, various meta-classes have been defined for representing common programming language constructs, such as a conditional-statement (similar to an if-statement or switch-statement) and a loop-statement (similar to a for-loop or while-loop). Note that these statements can be represented independent of a specific concrete syntax, which is abstracted in the transformations to a GenericAST model. UML support for representing behavioral constructs is limited (we used version 1.4), which is another reason for using a GenericAST as intermediate model in the transformation process, instead of transforming directly to UML. The GenericAST meta-model furthermore includes facilities for storing custom information in model elements by using tagged values, and for including references to the original source code.

4 Harvesting Workbench Developed and Used

We created a prototype tool set implementing the approach described that allowed us to harvest models from PL/SQL applications. This tool set makes use of the following components.

- The basis for our tool set is Interactive Objects' MDA environment ArcStyler.[1] ArcStyler is an extensible platform for MDA-based software architecting and engineering. It integrates a UML modeling environment with a collection of model transformations with can generate models or textual output base on UML models. It provides an open, flexible environment for tasks relating models, such as visualization, transformation (both model to model, and model to text), and manipulation. We mostly used the model to model transformation and visualization facilities, and to a lesser extend the model to text transformation facilities offered.
- We used the Grammatica[2] parser generator to obtain a PL/SQL parser. We have been able to reuse an existing PL/SQL grammar, which we tuned for our purposes.
- The mapping from grammars to MOF was set up according to the method proposed by [2], with some extensions in order to turn Grammatica parse trees into true abstract syntax trees.
- The MOF repositories were generated and manipulated using Interactive Objects' MDA-environment ArcStyler. In particular, ArcStyler's Carat.MOF functionality was used to generate a Java repository implementation, conforming to the Java Metadata Interface JMI [14], from a repository model represented in the UML profile for MOF.

[1] `http://www.arcstyler.com`

[2] `http://grammatica.percederberg.net`

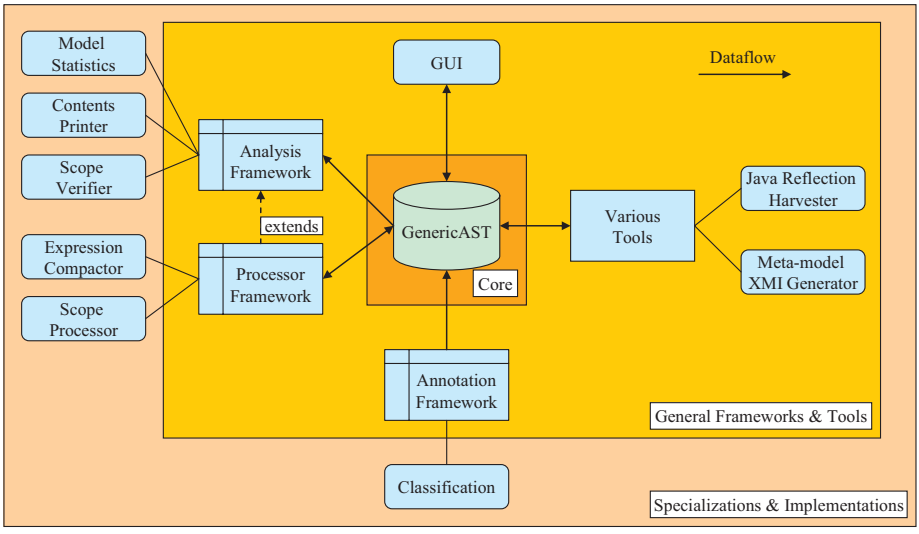

Fig. 2. GenericAST Framework and Tools

- The language independent analysis and transformation facilities offered at the GenericAST level are illustrated in Figure 2. They include tree traversals, transformations, a user interface for manipulating models, and an annotation framework.
- Model to model transformations (M2M) (e.g. from GenericAST to UML) were implemented in ArcStyler's prototype M2M-transformation engine called AIM – Atomistic Information Mapping. AIM provides a graphical user interface for defining transformations, which can be expressed in the Jython[3] scripting language.

5 Case Study

We have applied the harvester tools to HiBob, a 178 KLOC production system at De Amersfoortse Verzekeringen, a major insurance company based in The Netherlands. The system has been developed in Oracle's PL/SQL[4] and consisted of a data model with business logic that calculates insurance offers. The size of the application is shown in Table 1, both in KLOC PL/SQL and in the number of items.

For each main construct a grammar has been developed that describes part of the PL/SQL language, to generate parsers and meta-models that can process the input. The GenericAST has been used as an intermediate model, to reuse previously developed analyses and M2M-transformations to UML. ArcStyler has been

[3] `http://www.jython.org`
[4] `http://www.oracle.com/technology/tech/pl_sql`

Table 1. System size per main construct

Main Construct	Size (KLOC)	Item count
Tables with fields	70	163 with 4052
Triggers	28	468
Stored procedures (global)	7	46
Packages with procedures	73	23 with 538

used as MDA-environment for executing M2M-transformations and presenting the generated UML models.

Before commencing harvesting, the anticipated architecture of HiBob was determined. This first of all gives suggestions for the specific harvesting approach. Secondly, it will help to determine where there are mismatches between the current and the target architecture.

Only a small part of the harvesters is specific for HiBob, which is implemented in the model to model transformations from a grammar model to a GenericAST model. Using HiBob specific information, we were able to automatically modularize the application, based on known naming conventions. The modularization was implemented in one transformation rule, whereas all other transformation rules (more than 100) can be reused for harvesting other PL/SQL applications. The grammars can be completely reused for harvesting other PL/SQL applications. A grammar only regards syntax, which is not application specific.

Although there is no conceptual restriction on what target models are generated, in the case study only UML models were generated, including class diagrams, state-chart diagrams and collaboration diagrams. Class diagrams (such as in Figure 3) were used to gain insight in the data structure of the application (tables, fields, relations, triggers and constraints) and the structure of the behavior (stored procedures, packages with stored procedures, direct call dependencies). The generated model can be used for both documentation and code generation purposes.

Collaboration diagrams (such as in Figure 4) were derived to obtain insight in all required methods for executing a certain initial method, which is derived from direct call dependencies. The example diagrams shown here are relatively simple, there are for example collaboration diagrams with over 10 objects and more than 1000 method invocations required for more complex calculations.

Table 2 shows several performance measurements from the case study, performed on a Pentium4 3.0 GHz computer with 1 GB RAM. The measurements give an indication on the performance of parsing and populating a generated MOF repository. The table also shows that the cost of compressing and saving models by means of XMI are substantial. No explicit measurements have been done for the M2M-transformations, because we used a prototype M2M-transformation engine with known scalability issues. M2M-transformations ran for hours before completing which is due to the current implementation of the engine. A custom (Java) transformation may run much faster, but it is harder to keep a good overview of the transformation implementation.

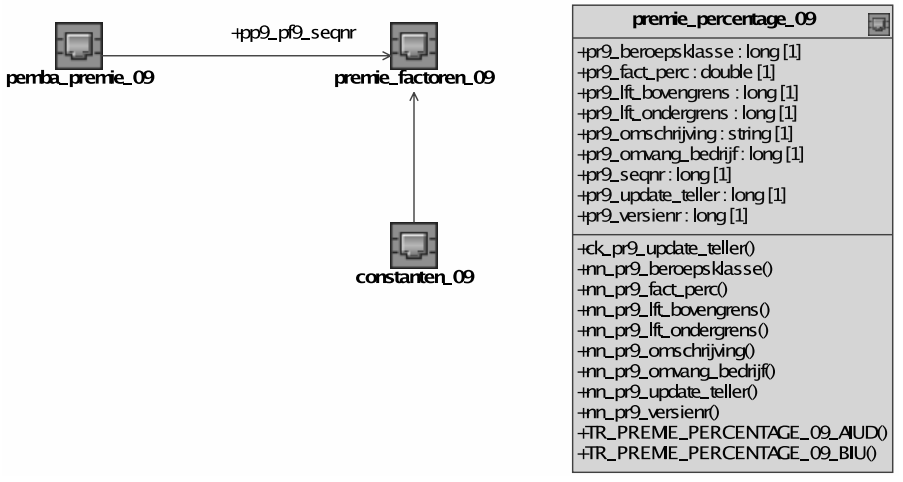

Fig. 3. Harvested Data Structures

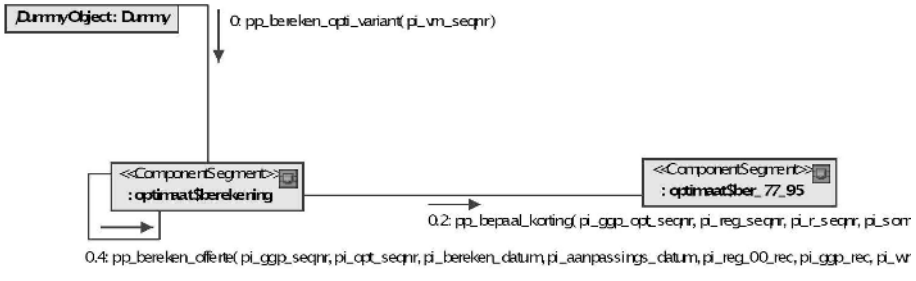

Fig. 4. Harvested Collaboration Diagram showing Required methods for executing method `pp_bereken_opti_variant`

Table 2. Performance measurements from Text to MOF-based model

Property	Unit	Tables	Triggers	Stored Procedures	Packages
Grammar size	LOC	190	240	250	250
Input size	KLOC	70	28	7	73
Parse time	Sec	6	13	3	33
Population time	Sec	9	42	7	144
Compressed XMI file size	KB	2580	15600	1800	41000
XMI save time	Sec	117	827	103	2660

The case study has shown the feasibility of harvesting existing source code (in this case 178 KLOC PL/SQL) to UML models, although the current prototype implementation suffers from scalability issues regarding M2M-transformations. Using the GenericAST has successfully enabled transformation reuse as intended,

although being an extra step in the extraction process. The extracted models have successfully been used for documentation and forward engineering purposes. Although currently no full migration has been done, experiments have been conducted in generating J2EE code from the harvested UML models, using standard model to text transformations shipped with ArcStyler. For data structures (tables with fields) complete J2EE code has been generated, whereas for the business logic (e.g. trigger implementations, stored procedures, and PL/SQL packages) only structural code has been generated (e.g. classes with methods, with an empty body).

Further details of the case study conducted can be found in [21].

6 Lessons Learned

Have a clear picture of what to harvest. It is important to have a specific question to answer or problem to solve before harvesting, and to know how to find the answer or solution: if you don't know where you're going, any road will take you there. In other words, constructs of interest that appear in the input must be specified, such that a harvester can recognize them. For example, if for a database system a question is 'what is the data structure?' then constructs of interest are table definitions, field definitions and relations between the tables.

Know the anticipated target models in detail. Having good knowledge on the target models that need to be harvested increases usability of the harvested models. For example, if the harvested models will be used for code generation and the code generator requires that associations have names then association should be given names during harvesting, which might not be necessary when generating models solely for documentation. This results in models that can be used directly for their purpose.

Make use of coding guidelines and naming conventions. The more system-specific information is used during harvesting, the better quality the initial models have. For example, if a table name contains a number that indicates the module it is part of, it can be used to automatically relocate the table to the right module. It improves the usability of the initial models.

Keep grammars small and focused. Smaller grammars are easier to maintain than bigger grammars. When a complete grammar is not available for a harvesting project, a minimal grammar should be developed. A minimal grammar has exactly the right information to describe the anticipated input, but not a complete language. This approach, which is based on the notion of *island grammar* [8,18] has been taken while harvesting HiBob.

Modularize grammars. Modular grammars improve reusability of commonly used grammar parts, such as statement and expression definitions. In the case study, four different grammars were composed from several grammar parts. Not

only does this save work while developing grammars, it also allows reuse of several M2M-transformation parts, which correspond with the grammar parts. This has successfully done in the case study.

Design grammar towards a target meta-model. Grammars should be developed with a certain target meta-model in mind, including how to map the grammar to the target meta-model. This improves reusability of M2M-transformation parts. For example, if in a target meta-model statements are all contained by a certain container, this should be reflected in the grammar. This could be done by having a production `statement_container` which contains other statements and acts the entry-point for statements in the grammar. This creates an extra node in the AST that can be mapped to the statement container model element in the target meta-model.

Avoid usage of XMI to store model contents. Persistence of harvesting results using XMI should be avoided, because it is a slow mechanism. Instead of saving and loading every time to and from XMI, as much as possible should be done without XMI. In the case study, parsing and populating a repository took in the order of seconds, while streaming to XMI took in the order of minutes. Therefore, it is more efficient to parse the input every time the models are needed for M2M-transformations. Because generation of a GenericAST model takes more time than saving and loading XMI, it should be used to save a generated GenericAST model.

Optimize M2M-transformations. M2M-transformations should be optimized everywhere possible. In the case study, M2M-transformations were the longest operations, which took in the order of days to complete. Simple optimizations could improve the performance, for example by doing calculations once, pass on the results to child rules where filtering takes place. In the case study, calculations where done at the same time as filtering, requiring each calculation to be executed multiple times instead of once.

Genericity Mismatch. The genericity (or expressiveness) of the GenericAST could be inadequate for a given harvesting project. It means that the GenericAST cannot represent constructs that appear in the project's source code. This risk is hard to identify and predict; it will show up during individual harvesting projects.

Possible ways of minimizing or dealing with the consequences are:

- Provide extension points in meta-model: Currently each element can be extended by tagged-values, which is a light-weight, pragmatic extension mechanism. This will not be sufficient for all situations, but it is a start.
- Extend GenericAST meta-model: Evolving the meta-model requires extensive testing in both the meta-model and tools, because it is a heavy-weight extension mechanism. It has impact on compatibility with existing models and transformations.

The genericity (or expressiveness) of the GenericAST could be too large. It happens when a semantic construct can be represented in more than one way. If this is true it is harder to create generic transformations and analyses on GenericAST models, because there could be two semantically equal models that do not result in the same target model and/or analysis result. This risk is hard to identify and predict.

Possible ways of minimizing the consequences are:

- Thorough meta-model review: For each meta-model element make sure why it exists and what can be represented with it. It should not be able to represent the same construct with any other element.
- Definition of well-formed model guidelines: Identified ambiguities must be resolved by making one option 'preferred' and the other options 'illegal', which can be done with a guideline. Any model that violates a guideline is not a valid GenericAST model. It may be enforced by providing a model-checker which detects and reports violations.

7 Concluding Remarks

Reverse engineering to an MDA target context requires a flexible, automated process that uses open standards as propagated by the OMG. Our reverse engineering framework provides an abstract process with minimal transformations to generate UML models from textual source code. The process consists of several transformations: textual source code is parsed into an AST, which is populated into a MOF-based repository with a meta-model conforming to the grammar that describes the structure of the textual source code. The contents of the MOF-based repository are transformed to an initial target model, which can be a UML model. The benefits of MDA can now be used to their full potential: generated models can be used for documentation, or even for MDA-based forward engineering.

To automate the process our prototype implementation uses generators. The source code structure is described in a grammar and from the grammar a specialized harvester and repository are generated. An improved mapping from an EBNF grammar to a MOF meta-model results in concise and usable repositories.

The generic intermediate model allows us to reuse M2M-transformations and analyses. For specific harvesters a transformation can be developed to generate a generic model. Available transformations and analyses can then be applied to the generic model to get the desired target models.

The case study has shown that the prototype implementation of the reverse engineering framework is able to extract models from a production system of 178 KLOC: UML models have been generated that give insight in structure and behavior. These models can be used for documentation purposes as well as for (partial) forward engineering: in our case Java classes have been generated from the harvested models that represent the application's structure.

Future work firstly regards solving scalability issues for our prototype implementation with the current M2M-transformation engine. An improved version

of such an engine is required to support larger M2M-transformations without running into performance problems. A functional extension of the prototype implementation is replacing the parser generator with a more powerful one, such as a Generalized LR parser generator [23] or an expression grammar parser generator [12]. A more powerful parser generator enables usage of island grammars [18], which is a powerful technique to quickly develop grammars for complex structured source code. In an island grammar constructs of interest can be specified in detail while the parser is told to ignore any other construct encountered in the input.

Secondly, it should be investigated whether information in GenericAST models is sufficient to generate target source code, such that method implementations can be generated as much as possible (at least for several constructs this expected to be feasible). A particular challenge is how to represent business logic. It is an open issue to what extend UML Action Semantics can help for the case at hand. To fully evaluate the current GenericAST prototype implementation should be tested on more source languages and evolved accordingly.

Last but not least, a full system migration should be attempted to assess feasibility of using reverse engineered models in an MDA-based forward engineering track. This might require transformation to specific UML profiles, domain specific languages and/or abstraction of platform dependent constructs to platform independent constructs.

References

1. R. Al-Ekram and K. Kontogiannis. An XML-based framework for language neutral program representation and generic analysis. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 42–51, Washington, DC, USA, 2005. IEEE Computer Society.
2. M. Alanen and I. Porres. A relation between context-free grammars and meta object facility meta-models. Technical Report 606, TUCS Turku Center for Computer Science, 2003.
3. A. Boronat, J. A. Carsi, and I. Ramos. Automatic reengineering in MDA using rewriting logic as transformation engine. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 228–231, Washington, DC, USA, 2005. IEEE Computer Society.
4. V. Cepa and M. Mezini. Language support for model-driven software development. *Science of Computer Programming*, 2006. Special issue on model-driven architectures; to appear.
5. E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, pages 13–17, January 1990.
6. A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 122–134. IEEE Computer Society Press, 2004.
7. A. van Deursen, P. Klint, and C. Verhoef. Research issues in software renovation. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE '99)*, Lecture Notes in Computer Science, pages 1–21. Springer-Verlag, 1999.

8. A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society, 1999.
9. D. Doyle. Transforming proprietary domain-specific modeling languages to model-driven architectures. Master's thesis, Delft University of Technology, 2005. URL: `swerl.tudelft.nl`.
10. J.-M. Favre and T. Nguyen. Towards a megamodel to model software evolution through transformations. *Electr. Notes Theor. Comput. Sci.*, 127(3):59–74, 2005.
11. ADM Task Force. Architecture-driven modernization roadmap. Technical report, OMG, 2006. Draft #1 dated 1/12/2006, `adm.omg.org`.
12. B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 111–122. ACM, 2004.
13. G. Gannod and M. Carey. Evolution of java programs to a model-driven environment using EMF. In *Proceedings EDOC Workshop on Model-Driven Evolution of Legacy Systems (MELS)*. IEEE Computer Society Digital Library, 2004.
14. Java Specification Requests. *JSR 040: Java Metadata Interface (JMI) Specification Version 1.0*, 2002.
15. I. Kurtev, J. Bézevin, and M. Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA 2002 Federated Conferences*. Springer-Verlag, 2002. Industrial Track.
16. 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.
17. L. Moonen. *Exploring Software Systems*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, December 2002.
18. L. Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society Press, June 2002.
19. OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, 2002. Final Adopted Specification, ptc/05-11-01.
20. OMG. *Meta Object Facility (MOF) Specification Version 1.4*, 2002.
21. T. Reus. Harvesting existing software systems for MDA-based reengineering. Master's thesis, Delft University of Technology, 2006. URL: `swerl.tudelft.nl`.
22. W. Ulrich. A status on OMG architecture-driven modernization task force. In *Proceedings EDOC Workshop on Model-Driven Evolution of Legacy Systems (MELS)*. IEEE Computer Society Digital Library, 2004.
23. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
24. 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.