1st Doctoral Symposium

of the

International Conference on Software Language Engineering (SLE)

Collected research abstracts

Eric Van Wyk and Steffen Zschaler (eds.)
October 11, 2010, Eindhoven, The Netherlands

Table of Contents

Preface

The first Doctoral Symposium to be organised by the series of International Conferences on Software Language Engineering (SLE) will be held on October 11, 2010 in Eindhoven, as part of the 3rd instance of SLE. This conference series aims to integrate the different sub-communities of the software-language engineering community to foster cross-fertilisation and strengthen research overall. The Doctoral Symposium at SLE 2010 aims to contribute towards these goals by providing a forum for both early and late-stage Ph.D. students to present their research and get detailed feedback and advice from researchers both in and out of their particular research area. Consequently, the main objectives of this event are:

- to give Ph.D. students an opportunity to write about and present their research;
- to provide Ph.D. students with constructive feedback from their peers and from established researchers in their own and in different SLE sub-communities;
- to build bridges for potential research collaboration; and
- to foster integrated thinking about SLE challenges across sub-communities.

All Ph.D. students participating in the Doctoral Symposium submitted an extended abstract describing their doctoral research. Based on a good set of submisssions we were able to accept 13 submissions for participation in the Doctoral Symposium. These proceedings present final revised versions of these accepted research abstracts. We are particularly happy to note that submissions to the Doctoral Symposium covered a wide range of SLE topics drawn from all SLE sub-communities.

In selecting submissions for the Doctoral Symposium, we were supported by the members of the Doctoral-Symposium Selection Committee (SC), representing senior researchers from all areas of the SLE community. We would like to thank them for their substantial effort, without which this Doctoral Symposium would not have been possible. Throughout, they have provided reviews that go beyond the normal format of a review being extra careful in pointing out potential areas of improvement of the research or its presentation. Hopefully, these reviews themselves will already contribute substantially towards the goals of the symposium and help students improve and advance their work. Furthermore, all submitting students were also asked to provide two reviews for other submissions. The members of the SC went out of their way to comment on the quality of these reviews helping students improve their reviewing skills.

We would also like to thank Mark van den Brand, the SLE General Chair, for asking us to organise this event and for providing invaluable assistance in doing so. Thanks are also due to the SLE Organizing Committee—Alex Anthony Cleve, Nicholas Kraft, Arjan van der Meer, and Alexander Serebrenik—for their help in publicising and organising this event. Finally, we would like to thank the Software Improvement Group for sponsoring a Best Paper award and the 250 Euro prize.

We are looking forward to a stimulating and enriching first Doctoral Symposium!

Eric Van Wyk and Steffen Zschaler
SLE Doctoral Symposium Co-chairs
Minneapolis and Lancaster, September, 2010

Doctoral Symposium Organisation

Doctoral Symposium Co-Chairs

Eric Van Wyk, Steffen Zschaler

Selection Committee

Colin Atkinson, Abraham Bernstein, Jordi Cabot, Tony Clark, Charles Consel, James Cordy, Dragan Gašević, Jeff Gray, Görel Hedin, Adrian Johnstone, Paul Klint, Dimitris Kolovos, Ivan Kurtev, Julia Lawall, Ralf Lämmel, Brian Malloy, Richard Paige, João Saraiva, Steffen Staab, Jurgen Vinju, Jos Warmer, Jon Whittle, Mark van den Brand

Practical Ambiguity Detection for Context-Free Grammars

Research Abstract

H. J. S. Basten

Centrum Wiskunde & Informatica

Abstract. The use of unconstrained context-free grammars for generalized parsing techniques has several advantages over traditional grammar classes, but comes with the danger of undiscovered ambiguities. The ambiguity problem for these grammars is undecidable in the general case, but this does not have to be a problem in practice. Our goal is to find ambiguity detection techniques that have sufficient precision and performance to make them suitable for practical use on realistic grammars. We give a short overview of related work, and propose new directions for improvement.

1 Problem Description and Motivation

Generalized parsing techniques allow the use of the entire class of context-free grammars (CFGs) for the specification of the syntax of programming languages. This has several advantages. First, it allows for modular syntax definitions, which simplifies grammar development and enables reuse. Second, it grants total freedom in structuring a grammar to best fit its intended use. Grammars do not have to be squeezed into LL, LALR or LR(k) form for instance.

Unfortunately, using unconstrained context-free grammars comes with the danger of ambiguities. A grammar is ambiguous if one or more sentences in its language have multiple parse trees. The semantics of a sentence is usually based upon the structure of its parse tree, so an ambiguous sentence can have multiple meanings. This often indicates a *grammar bug* which should be avoided. However, in some cases a grammar is intended to contain some degree of ambiguity. For instance in reverse engineering, where certain legacy languages can only be disambiguated with type checking after parsing. In both cases it is important to know the sources of ambiguity in the developed grammar, so they can be resolved or verified.

Unfortunately, detecting the (un)ambiguity of a grammar is undecidable in the general case [7, 10, 9]. However, this does not necessarily have to be a problem in practice. Several Ambiguity Detection Methods (ADMs) exist that approach the problem from different angles, all with their own strengths and weaknesses. Because of the undecidability of the problem there is a general tradeoff between precision and performance/termination. The challenge for all ADMs is to give the most precise and understandable answer in the time available. The current state of the art is not yet sufficiently advanced to be practical on realistic grammars, especially the larger ones.

2 Brief Overview of Related Work

Existing ADMs can roughly be divided into two categories: *exhaustive* methods and *approximative* ones. Methods in the first category exhaustively search the language of a grammar for ambiguous sentences. This so called *sentence generation* is applied by [11, 8, 13, 1]. These methods are 100% accurate, but a problem is that they never terminate if the grammar's language is of infinite size, which usually is the case. They do produce the most precise and useful ambiguity reports, namely ambiguous sentences and their parse trees.

Approximative methods sacrifice accuracy to be able to always finish in finite time. They search an approximation of the grammar for possible ambiguity. The methods described in [12, 6] both apply conservative approximation to never miss ambiguities. The downside of this is that when they do find ambiguities, it is hard to verify whether or not these are false positives.

In [2] we compared the practical usability of several ADMs on a set of grammars for real world programming languages. It turned out that the exhaustive sentence generator AMBER [13] was the most practical due to its exact reports and reasonable performance. However, it was still unsatisfactory to find realistic ambiguities in longer sentences. The approximative Noncanonical Unambiguity test [12] had a reasonably high accuracy, but it is only able to assess the ambiguity of a grammar as a whole. Its reports might point out sources of individual ambiguities, but these can be hard to understand.

3 Proposed Solution

The aim of this research is to increase the precision and performance of ambiguity detection to a practical level. More specifically, the idea is to increase the performance of exhaustive searching by reducing the search space using approximative techniques. For instance by identifying *harmless production rules* in a grammar. These are rules that are certainly not used in the derivation of any ambiguous string. Since these rules do not contribute to the ambiguity of the grammar they can be removed before exhaustive searching is applied, which reduces the number of sentences to generate.

In [3] we describe a first exploration in this direction. We propose an extension to the approximative Noncanonical Unambiguity test that enables it to identify harmless production rules. We implemented this filtering technique into a tool [4], which we applied on a series of real world programming language grammars in [5]. It is shown that the performance of the sentence generators AMBER and CfgAnalyzer is indeed improved by several orders of magnitude, with only a small filtering overhead.

Further research will be focussed on finding more detailed detection techniques that can identify harmless rules with more precision, as well as faster sentence generation methods. For instance by exploring opportunities for paralellisation. Furthermore, we like to extend existing techniques to include commonly used disambiguation constructs, like priorities and associativities, longest match, keyword reservation, etc.

4 Research Method

New techniques will be validated both theoretically and experimentally. Through formal specification they will be proved correct. Then, to test a technique's suitability for practical use, a prototype implementation will be tested on a series of realistic benchmark grammars. For instance grammars for real world programming languages, that will be seeded with ambiguities if needed.

5 Conclusion

This abstract proposes research into ambiguity detection for context-free grammars, to make it suitable for practical use. More specifically, it aims at combining approximative searching with exhaustive searching, to be able to find real ambiguous sentences in shorter time. First explorations in this direction show promising results.

References

1. Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental SAT solver. In: Proceedings of the 35th International Colloquium on Automata, Languages, and Programming (ICALP 2008). LNCS, vol. 5126 (2008)
2. Basten, H.J.S.: The usability of ambiguity detection methods for context-free grammars. In: Johnstone, A., Vinju, J. (eds.) Proceedings of the Eigth Workshop on Language Descriptions, Tools and Applications (LDTA 2008). ENTCS, vol. 238 (2009)
3. Basten, H.J.S.: Tracking down the origins of ambiguity in context-free grammars. In: Proceedings of the Seventh International Colloquium on Theoretical Aspects of Computing (ICTAC 2010). Springer (2010), To appear, see `www.cwi.nl/~basten` for a preliminary version.
4. Basten, H.J.S., van der Storm, T.: AmbiDexter: Practical ambiguity detection, tool demonstration. In: Proceedings of the Tenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010). IEEE (2010), To appear, see `www.cwi.nl/~basten` for a preliminary version.
5. Basten, H.J.S., Vinju, J.J.: Faster ambiguity detection by grammar filtering. In: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010). ACM (2010), To appear, see `www.cwi.nl/~basten` for a preliminary version.
6. Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. Science of Computer Programming 75(3), 176–191 (2010)
7. Cantor, D.G.: On the ambiguity problem of Backus systems. Journal of the ACM 9(4), 477–479 (1962)

8. Cheung, B.S.N., Uzgalis, R.C.: Ambiguity in context-free grammars. In: Proceedings of the 1995 ACM Symposium on Applied Computing (SAC 1995). pp. 272–276. ACM Press, New York, NY, USA (1995), http://doi.acm.org/10.1145/315891.315991
9. Chomsky, N., Schützenberger, M.: The algebraic theory of context-free languages. In: Braffort, P. (ed.) Computer Programming and Formal Systems, pp. 118–161. North-Holland, Amsterdam (1963)
10. Floyd, R.W.: On ambiguity in phrase structure languages. Communications of the ACM 5(10), 526–534 (1962)
11. Gorn, S.: Detection of generative ambiguities in context-free mechanical languages. J. ACM 10(2), 196–208 (1963), http://doi.acm.org/10.1145/321160.321168
12. Schmitz, S.: Conservative ambiguity detection in context-free grammars. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP'07: 34th International Colloquium on Automata, Languages and Programming. LNCS, vol. 4596 (2007)
13. Schröer, F.W.: AMBER, an ambiguity checker for context-free grammars. Tech. rep., compilertools.net (2001), see http://accent.compilertools.net/Amber.html

Generating Semantic Editors using Reference Attribute Grammars

Emma Söderberg

Department of Computer Science, Lund University, Lund, Sweden
emma.soderberg@cs.lth.se

Abstract. In this short research plan we address the problem of generating modern semantic editors from formal specifications. We aim to tackle this problem using reference attribute grammars.

1 Problem Description and Motivation

There are a lot of programming languages around and the number keeps increasing. Most of these languages have small communities with limited resources. As a consequence, a lot of development in these languages is performed in simple text editors, in lack of better semantic tool support. At the same time, users of languages like Java with larger communities can choose from a set of high-quality semantic editors, like the Eclipse JDT, IntelliJ IDEA or NetBeans, with modern semantic services like context-sensitive name completion and refactorings [18].

Preferably, it should be simple and fast to develop semantic tools with modern services like these for languages with smaller communities or less resources. However, these tools are hand-crafted and developed over several years. One appealing approach to reduce the development time of semantic tools is to generate them from a formal specification [17]. This approach has several benefits in that it lets developers describe behavior on a higher conceptual level. Also, the specification is typically smaller than its manually implemented counterpart, which makes it easier to overview and easier to change. Another benefit is the possibility to check the specification for semantic errors, an activity which would require more cumbersome testing in a hand-coded implementation.

We can summarize the above need for semantic editors and the benefits of generation into a problem of *generating modern semantic editors from a formal semantic description*. This problem includes technical difficulties such as coping with growing languages [41] and extensibility, responsiveness and performance of a generated editor, and flexible descriptions of the views and services of an editor. The rest of this document aims to give a coarse overview of a plan for research addressing this problem.

2 Brief Overview of Related Work

There exists several formal means for specifying semantics. For example, attribute grammars (AGs) by Knuth [26], denotational semantics by Scott and Strachey [39, 42], natural semantics by Kahn [21], and algebraic semantics by Bergstra et al. [4]. We will focus on reference attribute grammars (RAGs) by Hedin [16], an extended form of AGs. A benefit of RAGs is their ability to explicitly express super-imposed graphs on top of an abstract syntax tree (AST). Super-imposed graphs like these can be used to describe for example inheritance and cross-references. RAGs have been shown useful for describing the semantics of complex languages like Java [13] and Modelica [2]. Some examples of systems supporting RAGs are JastAdd by Ekman et al. [19, 14], Silver by van Wyk et al. [44], Kiama by Sloane et al. [40], and Aster by Kats et al.[24].

Examples of earlier systems generating semantic editors from formal specifications include the PSG system by Bahlke et al. [3] using denotational semantics, the CENTAUR system by Borras et al. [6] using natural semantics, the ASF+SDF meta-environment [25] using algebraic semantics, the Synthesizer Generator by Reps et al. [35] using ordered attribute grammars (OAGs) by Kastens [22], and the Lrc system by Kuipers et al [28] using higher-order attribute grammars by Vogt et al. [46]. OAGs is a powerful subset of AGs enabling a static evaluation order.

One important property of a semantic editor is incremental updating of the semantic model. Both the Synthesizer Generator and the Lrc system support incremental updating. The statically known evaluation order of OAGs, supported by these systems, provide sufficient information for incremental updating of attribute values. RAGs have also been used in the context of editors, in the APPLAB system by Bjarnason et al. [5], but not in conjunction with incremental updating. In general, RAGs require dynamic evaluation and incremental updating of RAGs is an open problem.

In recent years, a number of tool generating systems have emerged which extend the Eclipse Platform. One example is the Eclipse Modeling Framework (EMF) by Budinsky et al. [8] which provides means for expressing structured data models (graphs). EMF can generate a basic graphical editor for these models and supports updating of a model via manual registration of model observers.

Another example is the IDE Meta-tooling Platform (IMP) by Charles et al. [10] which has a semi-automatic approach to the development of textual semantic editors. IMP semi-generates text editors using wizards and generation of code skeletons. Developers manually fill in language-specific behavior in these code skeletons. Parsing is supported by the LPG parser generator, but this is optional as shown in, for example, the Spoofax/IMP system by Kats et al. [23] which extends IMP using a different parsing technology. Spoofax provides a language workbench which uses strategic term rewriting [45] to express language semantics.

The EMF project also supports generation of textual editors via the xText project by Efftinger et al. [12]. In contrast to IMP, xText generates a more complete text editor based on a custom grammar format, using an EMF-based model and an ANTLR parser. xText uses a combination of the Object Constraint Language (OCL) [43] and dependency injection to implement semantics.

These frameworks could possibly be used as the target platform for a generated editor based on RAGs. A combination of EMF and JastAdd models have been explored by Bürger et al. in JastEMF [9]. Another example of a system supporting generation of textual editors is the MontiCore system by Krahn et al. [27]. MontiCore uses a combined grammar format for concrete and abstract syntax supporting modular language extensions. This grammar format uses UML-like associations to describe semantics.

3 Proposed Solution

In order to address the problem posed in Section 1 we need a formal yet flexible way to describe the semantics of a language, including the abstract syntax. The semantic descriptions need to be modular in order to accommodate the need for extensibility. The semantic formalism also needs to be expressive to such a degree that the semantic information needed by the editing services can be computed. Beyond the need for pure semantic descriptions, we need a framework surrounding the underlying semantic model of a program and a means to describe services and views. These descriptions of services and views should seamlessly connect to the semantic descriptions.

We have chosen to use RAGs for semantic descriptions and we aim to construct a tool JedGen – *JastAdd-based semantic editor generator* supporting the remaining parts. These remaining parts include the framework surrounding a generated editor, means for describing services and views, and the actual generation of editors. We aim to support all languages that would benefit from static semantic analysis during development. To meet the demands on semantic development tools of today we have devised a list of three areas which a generated editor should support to be on par with hand-crafted modern semantic editors:

– *Incremental update* This is a highly desirable part of an interactive tool which affects performance and hence responsiveness. Incremental updating of RAG-based models is an open problem which we plan to address. A solution can possibly be based on work by Reps [34], by Jones [20], by Hedin [15], by Boyland [7], and by Acar et al. [1]. A solution to incremental updating of RAGs would be a contribution of this thesis.

– *Multiple views* Different development tasks benefit from different views of source code artifacts. Here, we include all views, editable or non-editable. This includes textual editors. Multiple views require a general architecture with support for synchronization and updating in a multi-threaded environment. Also, a generator needs to support a general way to specify the content and visualization of these views. Some work has been done on visualizing programs using RAGs [29] which we plan to extend along with a surrounding framework.

– *Modern semantic services* Inspection and modification of source code artifacts, requiring *context-sensitive static semantic information*. Some examples of services are code smell detection, context-sensitive metrics [11], cross-referencing, renaming and name completion. Promising work by Schäfer et al. [37, 38, 36] show that RAGs can be used to support sophisticated services like refactorings, and work by Nilsson-Nyman et al. [33] show how RAGs can be used to find dead code. The potential contributions of this work are further explorations of descriptions of semantic service information and service descriptions seamlessly connecting to RAG-based semantic descriptions.

4 Research Method

Our research method is constructive and experimental. We base our research on the hypothesis that "*RAGs can be used to generate modern semantic editors*", which we aim to demonstrate using a prototype. The development of a RAG-based generator prototype makes our research constructive.

The plan for the development of JedGen includes two phases – a *prototype framework* and a *prototype generator*. The purpose of the first phase is to build the general framework needed around a generated editor. This work can be separated into three subparts – the development of incremental updating of RAG-based ASTs, the development of mechanisms for access and updating of the ASTs in a general way and specification of views and services.

During the first phase the goal is to stepwise develop non-generated editor extensions to existing RAG-based compilers as a means for evaluation of the framework. Currently, we are working with editor extensions for Java and Modelica. We aim to evaluate these prototypes experimentally with regard to *behavior* (e.g., with regard to correctness), *coverage* (e.g., the range of errors that a generated editor can locate), *efficiency* (e.g., the performance of semantic analysis), and *effort* (e.g., line of code of an editor specification) The purpose of the second phase is to develop a prototype generator based on experiences gained in the previous phase. This includes a general description format for definition of editors based on an abstract syntax.

The JedGen tool is still in its first phase, but an alpha version of the tool supporting a semantic editing model has been used by Schäfer et al. in their exploration of refactorings [37, 38]. JedGen has also been used by several undergraduate students, as a part of their thesis work [31, 30, 32], and in a graduate course on RAGs.

5 Acknowledgements

A big thanks to all anonymous reviewers for valuable comments on an early version of this abstract.

References

1. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006.
2. Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Development of a Modelica compiler using JastAdd. *Science of Computer Programming*, 75:21–38, January 2010.
3. Rolf Bahlke and Gregor Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Langanguages and Systems (TOPLAS)*, 8(4):547–576, 1986.
4. Jan A. Bergstra. *Algebraic specification*. ACM, New York, NY, USA, 1989.
5. Elizabeth Bjarnason, Görel Hedin, and Klas Nilsson. Interactive language development for embedded systems. *Nordic Journal of Computing*, 6(1):36–54, 1999.
6. Patrick Borras, Dominique Clément, Th. Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and V. Pascual. CENTAUR: The system. In *Software Development Environments (SDE)*, pages 14–24, 1988.

7. John Tang Boyland. Incremental evaluators for remote attribute grammars. In *Proceedings of the Second Workshop on Language Descriptions, Tools and Applications (LDTA 2002)*, volume 65 of *Electronic Notes in Theoretical Computer Science*, pages 9–29. Elsevier B.V., July 2002.
8. Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.
9. Christoff Bürger and Sven Karol. JastEMF, 2010. http://code.google.com/p/jastemf [Access September 2010].
10. Philippe Charles, Robert M. Fuhrer, Stanley M. Sutton, Jr., Evelyn Duesterwald, and Jurgen Vinju. Accelerating the creation of customized, language-specific ides in eclipse. *SIGPLAN Notices*, 44(10):191–206, 2009.
11. Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
12. Sven Efftinge and Markus Völter. oAW xText: a framework for textual DSLs. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, Esslingen, Germany, October 2006.
13. Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM.
14. Torbjörn Ekman and Görel Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14–26, December 2007.
15. Görel Hedin. *Incremental semantic analysis*. PhD thesis, 1992.
16. Görel Hedin. An overview of door attribute grammars. In Peter Fritzson, editor, *CC*, volume 786 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 1994.
17. Jan Heering and Paul Klint. Semantics of programming languages: a tool-oriented approach. *SIGPLAN Notices*, 35(3):39–48, 2000.
18. Daqing Hou and Yuejiao Wang. An empirical analysis of the evolution of user-visible features in an integrated development environment. In *CASCON '09: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 122–135, New York, NY, USA, 2009. ACM.
19. The JastAdd Team. jastadd.org. http://jastadd.org/ [Access May 2010.
20. Larry G. Jones. Efficient evaluation of circular attribute grammars. *ACM Trans. Program. Lang. Syst.*, 12(3):429–462, 1990.
21. Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.
22. Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, March 1980.
23. Lennart C. L. Kats, Karl T. Kalleberg, and Eelco Visser. Domain specific languages for composable editor plugins. In Torbjörn Ekman and Jurgen Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier B. V., 2009.
24. Lennart C. L. Kats, Anthony M. Sloane, and Eelco Visser. Decorated attribute grammars: Attribute evaluation meets strategic programming. In Oege de Moor and Michael I. Schwartzbach, editors, *CC*, volume 5501 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2009.
25. Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):176–201, April 1993.
26. Donald E. Knuth. Semantics of context-free languages. *Journal Theory of Computing Systems*, 2(2):127–145, June 1968.
27. Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular development of textual domain specific languages. In Richard F. Paige and Bertrand Meyer, editors,

TOOLS (46), volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer, 2008.

28. Matthijs F. Kuiper and João Saraiva. Lrc - a generator for incremental language-oriented tools. In Kai Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*, pages 298–301. Springer, 1998.
29. Eva Magnusson and Görel Hedin. Program Visualization using Reference Attributed Grammars. volume 7, pages 67–86. Publishing Association Nordic Journal of Computing, 2000.
30. Jesper Mattsson. The JModelica IDE: Developing an IDE reusing a JastAdd compiler. Master's thesis, Lund University, Lund, Sweden, August 2009.
31. Erik Mossberg. Inspector – tool for interactive language development. Master's thesis, Lund University, Lund, Sweden, October 2009.
32. Philip Nilsson. Semantic editing compiler extensions using JastAdd. Master's thesis, Lund University, Lund, Sweden, June 2010. To be presented.
33. Emma Nilsson-Nyman, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. Declarative intraprocedural flow analysis of Java source code. In *Proceedings of the Eight Workshop on Language Description, Tools and Applications (LDTA 2008)*, Electronic Notes in Theoretical Computer Science. Elsevier B.V., 2008.
34. Thomas Reps. *Generating Language-Based Environments*. PhD thesis, 1984.
35. Thomas Reps and Tim Teitelbaum. The Synthesizer Generator. In Peter B. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19(5) of *SIGSOFT Software Engineering Notes*, pages 42–48, Pittsburgh, Pennsylvania, USA, May 1984. ACM.
36. Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct Refactoring of Concurrent Java Code. In Theo D'Hondt, editor, *24th European Conference on Object-Oriented Programming (ECOOP '10)*, 2010.
37. Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for Java. In Gail E. Harris, editor, *OOPSLA*, pages 277–294. ACM, 2008.
38. Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon. In Sophia Drossopoulou, editor, *ECOOP*, volume 5653 of *Lecture Notes in Computer Science*, pages 369–393. Springer, 2009.
39. Dana Scott. Mathematical concepts in programming language semantics. In *AFIPS '72 (Spring): Proceedings of the May 16-18, 1972, spring joint computer conference*, pages 225–234, New York, NY, USA, 1972. ACM.
40. Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. A pure object-oriented embedding of attribute grammars. In T. Ekman and J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier B. V., 2009.
41. Guy L. Steele Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, October 1999.
42. Christopher Strachey. Towards a formal semantics. pages 198–216, 1966.
43. The Object Management Group (OMG). The Object Constraints Language (OCL), 2010. http://www.omg.org/technology/documents/formal/ocl.htm [Accessed May 2010].
44. Eric van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. In *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, Electronic Notes in Theoretical Computer Science. Elsevier B. V., 2007.
45. Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, pages 357–362, London, UK, 2001. Springer-Verlag.
46. Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher-order attribute grammars. In *PLDI*, pages 131–145, 1989.

Zipper-based Embedding of Modern Attribute Grammar Extensions[⋆]

Pedro Martins
Universidade do Minho, Portugal

Abstract. This research abstract describes the research plan for a Ph.D project. We plan to define a powerful and elegant embedding of modern extensions to attribute grammars. Attribute grammars are a suitable formalism to express complex, multiple traversal algorithms. In recent years there has been a lot of work in attribute grammars, namely by defining new extensions to the formalism (forwarding and reference attribute grammars, etc), by proposing new attribute evaluation models (lazy and circular evaluators, etc) and by embedding attribute grammars (like first class attribute grammars). We will study how to design such extensions through a zipper-based embedding and we will study efficient evaluation models for this embedding. Finally, we will express several attribute grammars in our setting and we will analyse the performance of our implementation.

1 Problem Description and Motivation

Attribute grammars (AGs) [1] are a convenient formalism not only for specifying the semantic analysis phase of a compiler but also to model complex multiple traversal algorithms. Traditional AG systems tailor their own syntax for the definition of AGs. Attribute grammars in concrete syntaxes are then automatically transformed into efficient implementations in general purpose programming languages such as Haskell, OCaml and C. LRC [2], UUAG [3] or Silver [4] are among the AG systems that are designed in this way.

For many applications, it is desirable to design and implement a special purpose language that is tailored to the characteristics of the problem domain. Perhaps the most well-known example of such a language is provided by the compiler compiler system *yacc*. In general, however, the design and implementation of a new programming language from scratch can be costly. For that reason, the designers of Simula-67 proposed that new languages are implemented through library interfaces, so the new *embedded* language inherits all the benefits of its *host* language, and the implementation effort is much reduced.

In the context of attribute grammars, this idea has already been explored [5–7]. Indeed, each of this embeddings benefits from the particular characteristics of the host language in order to achieve elegant attribute grammar solutions.

⋆ This work has been partially supported by FCT (Portuguese Science Foundation) project AMADEUS, under grant (POCTI, PTDC/EIA/70271/2006)

Recently, an embedding for classic AGs has been developed in a functional language [8]. This embedding, of around 100 lines of code, relies on an extremely simple mechanism based on the notion of functional zippers [9].

With our work we intend to explore this new approach in a systematic way. Our goal is to fully develop a mature system with advanced AG constructions embedded in a zipper-based setting, namely circular attributes (fix-point computation), aspects (aspected oriented programming), higher-order and forwarding (functional programming), references (imperative programming), multiple inheritance (object oriented programming), strategies (strategic programming) and incremental attribute evaluation (incremental computation).

Once we define how such an embedding can be modeled in a functional setting, we will study different models of execution for the AGs. Finally, we will conduct a series of experiments in order to benchmark our system against other well-established ones.

2 Overview and Related Work

Attribute grammars have proven to be a suitable formalism to the design and implementation of both domain specific and general purpose languages, with powerful systems based on attribute grammars [10–12, 2] been constructed. While, in the beginning, AG systems were used mainly to specify and derive efficient (batch) compilers for formal languages, nowadays, AG-based systems are powerful tools that not only specify compilers, but also syntax editors [10], programming environments [2], visual languages [13], complex pretty printing algorithms [14], program animations [15], etc. More recently, new extensions/features have been defined for attribute grammars, like forwarding attribute grammars [16], higher-order attribute grammars [17, 15], reference attribute grammars [18], multiple inheritance [19], aspect oriented attribute grammars [20].

While these extensions can be considered standard in traditional systems, they have not yet been studied in the context of modern functional AG embeddings, such as [5, 7]. Given the simple mechanism of [8], however, we believe that some extensions should be simple to obtain in that setting, while others, although probably requiring improvements on the embedding mechanism itself, still can be achieved elegantly.

With our work, we also propose to make incremental computation available within our embedded AG system. Reps was the first to use incremental attribute evaluation in the structured editors produced by the Synthesizer Generator System [10]. The LRC system [2] is a programming environment generator that uses a strict, purely functional attribute evaluators and incrementality is obtained via function memoisation [21]. The Eli [12] AG systems produce visual programming environments but it does not support incremental evaluation, yet. The ASF+SDF is a programming environment generator based on the paradigm of term re-writing [22]. The action semantics environment was developed with ASF+SDF [23]. None of these systems use incremental evaluation.

In the context of functional programming, John Hughes was the first to work on the incremental evaluation of lazy programs [24]. Acar *et al* [25] presented a general technique for the incrementalisation of functional programs. Magnus Carlsson modeled this work as a Haskell library [26]. These techniques, however, do not handle lazy evaluation. In [8], the authors show that within their embedding, incremental computation can be obtained via function memoization. Their techniques, however, do not require lazy evaluation. This means that if we rely on lazyness in any of our extensions, this approach to incrementality may break down and further studies are necessary. Even if this does not occur, the impact of memoization in the authors' approach still needs to be evaluated.

3 Research Method

The aim of this project is to embed advanced features of the AG paradigm into a general purpose, lazy, and purely functional language. This goal builds upon the definition attribute grammars as a domain-specific embedded language in Haskell by [8]. That is, attribute grammars become a Haskell library of higher-order and lazy functions, and we want to model in this library new language concepts like circular attributes, aspects, higher-ordeness, forwarding, references, multiple inheritance, strategies, and, finally, incremental evaluation.

We also intend to conduct a systematic performance study of traditional attribute grammar evaluators *versus* their implementation as a library in Haskell. We want to verify with realistic examples if a highly optimized functional implementation of AGs (lazy, strict and deforested evaluators [21]) is really faster than the simple Haskell library. The outcome of the performance experiments, will allow us to to recommend improvements both to existing compilers for Haskell, attribute grammar based systems, and incremental programming environments.

The phases described below constitute the main work areas for our work:

3.1 Design: For many applications, it is desirable to design and implement a special purpose language that is tailored to the characteristics of the problem domain. However, the design and implementation of a new programming language from scratch is usually costy. The idea of embedded languages has been enthusiastically embraced by the functional programming community [14, 7, 8].

The first goal to achieve in this project is to enhance the zipper-based embedding of [8] with advanced AG features such as aspects, higher-order and circular attributes, references, multiple inheritance and incremental attribute evaluation. In order to achieve this, we will design and propose different implementations for each feature, and each proposal will possibly rely on a different advanced feature of the host language, HASKELL. Then, we will conduct several experiments in order to realize which proposal achieves our goal as elegantly as possible.

3.2 Implementation: One of the uses of AGs today is in the creation of programming environments (also called language-based editing environments), that report on syntactic and semantic errors as the program is being constructed. In such an environment, it is important that the attribute values are computed *incrementally* after each edit action, re-using results of previous computations where possible. Reps and Teitelbaum were the first to demonstrate the feasibility of the idea [10]. It is partly because of this emphasis on incremental computation that the embedding of attribute grammars into functional programming was ignored: it was not clear at all how incremental computation could be achieved.

An important step was made by João Saraiva, who showed how the known attribute evaluation techniques could all be implemented in a strict, purely functional setting [21, 27]. He was however not able to apply these techniques as part of an implementation of attribute grammars as a software library in Haskell: a quite complex preprocessor was still required.

Acar *et al.* [25] presented a new technique for incremental computation of functional programs, which is completely general as it can be used to make any program incremental. Furthermore, it is implemented as a library of functions in ML. A remarkable property of Acar's work is that it maintains a dynamic dependency graph, as opposed to the static dependency graph used in all previous work on attribute grammars. Due to this, it potentially requires few recomputations after the input has been changed. This theoretical advantage may however be outweighed in practice by the additional book-keeping required.

With our work we will study different execution models for our extensions. In particular, we will work on the development of incremental models of execution.

3.3 Benchmarks: We will apply the library developed in the previous phases to realistic examples (Java grammar, Pretty-printing optimal algorithm [28], etc). Then, we will compare the performance of the obtained implementations with equivalent implementations obtained by other AG systems. Firstly, we will compare the performance of our library against other functional AG embeddings, whenever such a comparison is possible (recall that most of the features we want to embed in our system are not available in functional embeddings such as [5, 7]). Secondly, we will compare our implementations against the ones derived by standard AG systems from AG expressed in special purpose languages [2–4].

Research Questions: The following research questions have to be answered in the project: How can we correctly and elegantly embed advanced attribute grammar features,*e.g.*, reference, forwarding, higher-orderness, in the embedding of [8]? What is the impact of memoization in the embedding of [8] and in the extensions we define for it? If lazyness is necessary for any of our extensions, how can we restore incremental computation? In other words, how can we combine lazyness and incremental computation? How does the perfomance of our implementations compare to well-established others?

4 Conclusions

We propose to develop a mature attribute grammar system, embedded in a modern functional programming language, and with all advanced AG features incorporated. Later, a systematic analysis on this system will be conducted.

The beneficiaries of this research are implementors of functional programming languages [29, 30], attribute grammar-based systems [12, 14, 2] and programming environments [10, 22, 23, 31], because we provide experimental evidence to guide further work. The software artifacts we produce in the process will be of use to a wide audience of functional programmers.

References

1. Knuth, D.E.: Semantics of Context-free Languages. Mathematical Systems Theory **2**(2) (1968) 127–145 *Correction: Math. Systems Theory* 5, 1, pp. 95-96 (1971).
2. Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In Koskimies, K., ed.: 7th International Conference on Compiler Construction. Volume 1383 of LNCS., Springer-Verlag (1998) 298–301
3. Swierstra, D., Baars, A., Löh, A.: The UU-AG attribute grammar system (2004)
4. Wyk, E.V., Krishnan, L., Bodin, D., Johnson, E., Schwerdfeger, A., Russell, P.: Tool Demonstration: Silver Extensible Compiler Frameworks and Modular Language Extensions for Java and C. In: SCAM. (2006) 161
5. de Moor, O., Backhouse, K., Swierstra, D.: First-Class Attribute Grammars. In Parigot, D., Mernik, M., eds.: Third Workshop on Attribute Grammars and their Applications, WAGA'99, Ponte de Lima, Portugal, INRIA Rocquencourt (2000)
6. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. In Ekman, T., Vinju, J., eds.: Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009). Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers (2009)
7. Viera, M., Swierstra, D., Swierstra, W.: Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell. In: Procs. of the 14th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'09). (2009) 245–256
8. Fernandes, J., Sloane, A., Saraiva, J., Cunha, J.: A lightweight functional embedding of attribute grammars (in preparation). (2010)
9. Huet, G.: The zipper. Journal of Functional Programming **7**(5) (1997) 549–554
10. Reps, T., Teitelbaum, T.: The Synthesizer Generator. Springer (1989)
11. Jourdan, M., Parigot, D., Julié, C., Durin, O., Bellec, C.L.: Design, implementation and evaluation of the fnc-2 attribute grammar system. In: PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, New York, NY, USA, ACM (1990) 209–222
12. Kastens, U., Pfahler, P., Jung, M.T.: The Eli System. In: CC '98: Procs. of the 7th Int. Conf. on Compiler Construction, London, UK, Springer-Verlag (1998) 294–297
13. Kastens, U., Schmidt, C.: Vl-eli: A generator for visual languages (2002)
14. Swierstra, D., Azero, P., Saraiva, J.: Designing and Implementing Combinator Languages. In Swierstra, D., Henriques, P., Oliveira, J., eds.: 3rd Summer School on Adv. Funct. Programming. Volume 1608 of LNCS Tutorial. (1999) 150–206
15. Saraiva, J.: Component-based programming for higher-order attribute grammars. In: GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering, London, UK, Springer-Verlag (2002) 268–282
16. Wyk, E.V., Moor, O.d., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, London, UK, Springer-Verlag (2002) 128–142
17. Swierstra, D., Vogt, H.: Higher order attribute grammars. In Alblas, H., Melichar, B., eds.: International Summer School on Attribute Grammars, Applications and Systems. Volume 545 of LNCS., Springer-Verlag (1991) 48–113
18. Hedin, G.: Reference attributed grammars. In Parigot, D., Mernik, M., eds.: 2nd Workshop on Attribute Grammars and their Applications. (1999) 153–172
19. Mernik, M., Lenic, M., Avdicausevic, E., Zumer, V.: Multiple attribute grammar inheritance. In: Informatica. (2000) 319–328

20. de Moor, O., Peyton Jones, S., van Wyk, E.: Aspect-oriented compilers. In: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99). LNCS (1999)
21. Saraiva, J.: Purely Functional Implementation of Attribute Grammars. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands (1999)
22. van den Brand, M., Klint, P., Olivier, P.: Compilation and Memory Management for ASF+SDF. In Stefan Jähnichen, ed.: 8th International Conference on Compiler Construction. Volume 1575 of LNCS., Springer-Verlag (1999) 198–213
23. van den Brand, M., Iversen, J., Mosses, P.D.: An action environment. Sci. Comput. Program. **61**(3) (2006) 245–264
24. Hughes, J.: Lazy memo-functions. In Jouannaud, J.P., ed.: Functional Programming Languages and Computer Architecture. Volume 201 of LNCS., Springer-Verlag (1985) 129–146
25. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. In: POPL'02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, ACM (2002) 247–259
26. Carlsson, M.: Monads for incremental computing. In: ICFP'02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM (2002) 26–35
27. Saraiva, J., Swierstra, D., Kuiper, M.: Functional Incremental Attribute Evaluation. In David Watt, ed.: 9th International Conference on Compiler Construction, CC/ETAPS'2000. Volume 1781 of LNCS., Springer-Verlag (2000) 279–294
28. Swierstra, D., Chitil, O.: Linear, bounded, functional pretty-printing. Journal of Functional Programming **19**(01) (2009) 1–16
29. Peyton Jones, S., Hughes, J., Augustsson, L., et al.: Report on the programming language Haskell 98. Technical report (1999)
30. Leroy, X.: The Objective Caml System - Documentation and User's Manual (1997)
31. Michiel, M.: Proxima : a presentation-oriented editor for structured documents. PhD thesis, Utrecht University, The Netherlands (2004)

Automata Based Method for Domain-specific Languages Definition

Ulyana Tikhonova,

Supervisor: Fedor Novikov
St. Petersburg State Polytechnical University, Applied Mathematics Dept.,
Politekhnicheskaya 29., 195251 St. Petersburg, Russia
ulyana.tihonova@gmail.com, fedornovikov@rambler.ru

Abstract. We outline a research proposal which goal is to contribute to methods of new Domain-Specific Languages (DSLs) definition and implementation. We propose the automata based method for DSLs definition that allows specifying new languages with various notations in such a way that the language definition can be treated as a ready-to-use language implementation already.
The automata based method allows defining language by three components: language metamodel (which includes an abstract syntax), concrete syntax and operational semantics. We use Unified Modeling Language (UML) as description formalism for all three components. Namely, language metamodel is defined using class diagrams. Concrete syntax is defined as parser using state machine diagrams. Semantics is defined as metamodel interpreter using state machine diagrams as well.

Keywords: DSL, metamodel, concrete syntax, operational semantics, UML.

1 Motivation

Domain-specific Languages (DSLs) are considered to be very effective in software engineering. They raise the level of abstraction, provide common domain notation, and improve development process therefore. Per se, DSLs allow describing a problem solution in terms of the field of the problem, rather than in terms of computer. The most critical part of the whole software development process in the context of language oriented programming paradigm is definition and implementation of a new DSL [14]. All approaches to solve this problem could be divided into those, which use traditional grammars, and those, which are based on modeling in context of Model Driven Engineering (MDE) [4]. The former approach allows defining programming language as combination of its structure and textual syntax. The latter implies definition of language metamodel mostly in terms of MOF and subsequent usage of the model for code generation, model transformation, etc. In this case, concrete syntax is usually graphical or even undefined.

However, these two styles of language definition do not differ in essence. As was noticed in [5] and [3] the program in any DSL is an abstract structure, and for its editing, storage and execution various representations might be used: text, diagrams, tables, formulas, sounds, etc. Therefore, a single method for definition of different notations is desirable.

Another issue is specification of language semantics. The brief overview of methods of semantics definition is given in [2]. They vary from complicated formulas of axiomatic semantics to rewriting rules of translation semantics. We consider that usage of the same formalism for concrete syntax definition and for semantics definition would simplify the process of new DSL creation, which is rather complicated now.

At last, specification of new DSL should be sufficient for receiving its implementation automatically.

2 Brief Overview of Related Work

A number of language workbenches, which support development of DSLs with not only textual notation, were worked out recently. One of them is MetaEdit+ tool [8] that allows creation of graphical DSLs with facility to specify generation of various sorts of target data from DSL diagrams. MetaEdit+ uses its own model of DSL abstract syntax – the metamodeling language GOPPRR (Graph-Object-Property-Port-Role-Relationship).

Another one, AMMA [1], is a model based framework that supports DSL development with metamodel definition language KM3 [11], language for specifying textual concrete syntaxes TCS and model transformation language ATL. This project is based on MOF formalism and supports graphical syntax through class diagrams of models. The common approach is implemented in MOFLON [10] project. In addition the latter uses Story Driven Modeling (SDM) paradigm for definition of the dynamic semantics of a DSL.

We appeal for possibility to define various notations using single technique and for possibility to define both concrete syntax and semantics using the same specification (meta)language.

3 Proposed Solution

In this work, we propose just another DSL definition method based on model driven architecture (MDA) and executable UML approach [6]. Definition of DSL using MDA approach instead of traditional formal grammars advances software engineering unification. This approach requires separation of language definition levels, which are abstract syntax, concrete syntax and semantics. Therefore, the proposed method consists in correlated definitions of DSL metamodel (which includes abstract syntax), concrete syntax and operational semantics (Fig. 1). We use UML [12] as description formalism and investigate definition methods that differ from those listed in part 2 for all three steps of DSL specification process.

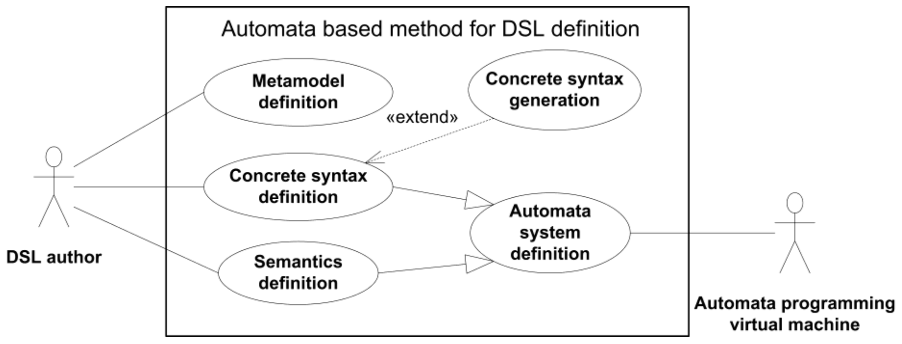

Automata based method for DSL definition
Metamodel definition
Concrete syntax generation
«extend»
Concrete syntax definition
Automata system definition
Semantics definition
DSL author
Automata programming virtual machine

Fig. 1. Use case diagram of automata based method for DSL definition

We extend formalism used for language abstract syntax definition to UML class diagram in comparison with widely spread MOF, as UML class diagram can express more copious structures. We propose description both of concrete syntax and of operational semantics as algorithms through UML state machine diagrams. Namely, concrete syntax is defined as a parsing algorithm, which analyzes source program representation and constructs abstract program. Operational semantics is defined as an algorithm of abstract program interpretation.

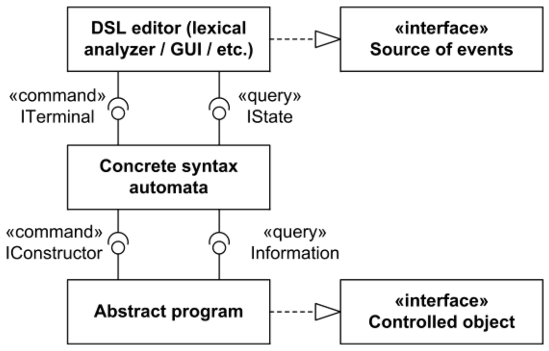


Fig. 2. Component diagram of concrete syntax automata, abstract program and DSL editor

One of the key features of the proposed method is unified view on different language notations. Any abstract program representation is considered as chain of events, which are processed by automata system defined in concrete syntax specification. Each terminal representation is considered as an event sent by some source of events. For example, events might be typographical characters in text, geometrical figures in diagram, controls in dialogue window, cells in spreadsheet, or sounds of spoken commands. Accordingly, any source of events is acceptable: a lexical analyzer of text, a graphical editor of diagrams, an editor of formulas or a dialogue window (Fig. 2).

To achieve automatic receiving of DSL's implementation we use automata based programming paradigm [7, 13]. According to this paradigm, every algorithm described through UML state machine diagrams can be executed by an automata programming virtual machine (Fig. 1).

4 Research Method

We have developed an initial candidate DSL meta-metamodel – the abstract syntax of the proposed method (Fig. 3). This meta-metamodel accumulates expressiveness both of grammar formalism and of UML class diagram. We are going to investigate it and compare it with formal grammars to find out the kind of languages that could be defined as instances of this meta-metamodel. Moreover, mapping between DSL meta-metamodel and formal grammars could be useful for the reuse of already defined languages. This mapping could be also useful for development of the algorithm of generation of concrete syntax automata system from DSL metamodel.

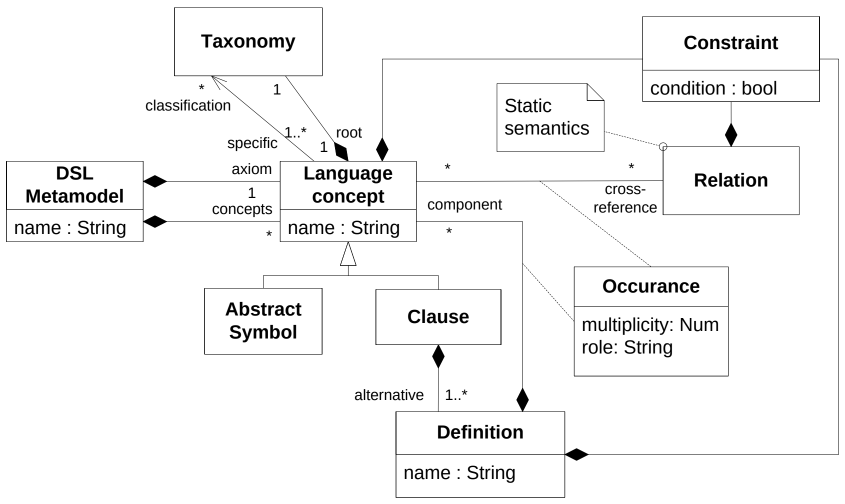


Fig. 3. Class diagram of DSL meta-metamodel

We have devised the automata model being used as formalism for definition of both concrete syntax and semantics. Two little DSLs have been specified using this model: nonlinear and graphical language of the chess position and the mini language for manipulations with sets. Further, the automata programming virtual machine should be developed to execute these specifications of DSLs. Definition of the automata programming virtual machine with the means of the proposed method would be the best use case. In other words, we are going to apply the idea of bootstrapping.

The proposed method for DSL definition allows defining different languages with various concrete representations. DSL definition can be used as its software implementation in terms of the automata based virtual machine.

References

1. Bézivin, J., Jouault, F., Kurtev, I., Valduriez, P.: Model-Based DSL Frameworks. In: 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pp. 602-616. ACM, New York (2006)
2. Combemale, B., Crégut, X., Garoche, P.-L., Thirioux, X.: Essay on Semantics Definition in MDE - An Instrumented Approach for Model. In: Journal of Software, Vol 4, No 9, pp. 943-958 (2009)
3. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm, http://www.onboard.jetbrains.com/is1/articles/04/10/lop/index.html (2005)
4. Estublier, J., Vega, G., Ionita, A.D.: Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications. In: Briand, L., Williams, C. (Eds.) International Conference on Model Driven Engineering Languages and Systems (MoDELS). LNCS vol. 3713, pp. 69-83. Springer-Verlag, Berlin Heidelberg (2005)
5. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages?, http://www.martinfowler.com/articles/languageWorkbench.html (2005)
6. Frankel, D.S.: Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley Publishing, Inc., Indianapolis, Indiana (2003).
7. Gurov, V. S., Mazin, M. A., Narvsky, A. S., Shalyto, A. A.: Tools for Support of Automata-Based Programming. J. Programming and Computer Software, Vol. 33, No. 6, pp. 343–355 (2007), http://is.ifmo.ru/articles_en/_ProCom6_07GurovLO.pdf
8. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling. IEEE Computer Society Publications, New Jersey (2008)
9. Meta Programming System, http://www.jetbrains.com/mps/
10. MOFLON project, http://www.moflon.org/
11. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Briand, L., Williams, C. (Eds.) International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS vol. 3713, pp. 264–278. Springer-Verlag, Berlin Heidelberg (2005)
12. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.2 (2009), http://www.uml.org
13. Paraschenko, D., Shalyto, A., Tsarev, F.: Modeling Technology for One Class of Multi-Agent Systems with Automata Based Programming. In: IEEE International Conference on Computational Intelligence for Measurement Systems and Applications, pp. 15-20. IEEE Xplore, La Coruna (2006)
14. Ward, M.: Language Oriented Programming. In: Software - Concepts and Tools, Springer Berlin / Heidelberg, Vol.15, No.4, pp. 147-161 (1994)

Test Case Generation for Programming Language Metamodels

Abstract for Software Language Engineering 2010 Doctoral Symposium

Hao Wu[⋆]
Supervisors: Rosemary Monahan and James F. Power

Department of Computer Science, National University of Ireland, Maynooth
{haowu,rosemary,jpower}@cs.nuim.ie

1 Problem Description and Motivation

One of the central themes in software language engineering is the specification of programming languages, and domain-specific languages, using a *metamodel*. This metamodel provides a greater degree of abstraction than a context-free grammar, since it can ignore syntactic details. However, its main benefit is in providing for the specification of the abstract syntax graph of a language, and this can then be used for the construction or generation of language processing tools.

One problem associated with the use of programming language metamodels, and metamodels in general, is determining whether or not they are correct. Of course, one approach is to forward engineer code and test this, but it should also be possible to test the metamodel directly. In this context, the question addressed by our research is: *given a programming language metamodel, how can we generate an appropriate test suite to show that it is valid?*

Being able to generate such a test suite would have two main benefits. First, examining the automatically-generated test cases would help to develop the modeller's understanding of the metamodel, and help to increase confidence in its validity. Second, since the metamodel specifies a programming language, the generated test cases should be valid programs from that language, and these can be used as test inputs for tools that process the language.

2 Related work

Testing a programming language specifications, at least at the syntactic level, has a long history, dating back at least to the work of Purdom on generating test cases from grammars [1]. However, a naive application of Purdom's approach to a programming language grammar produces programs that may not be syntactically correct (since the grammar may under-specify), and is certainly unlikely to produce semantically valid (let alone meaningful) programs [2].

⋆ This work is supported by a John & Pat Hume Scholarship from NUI Maynooth.

Incorporating at least a language's static semantics into test suite generation must go beyond simple context-free grammars. One possible approach is to use attribute grammars, and to seek to exploit the attribute equations to constrain or even direct the generation of test cases [3, 4]. Despite this research, it is still not a trivial task to directly extend this work to a metamodelling environment that uses, for example, OCL-like constraints for describing the static semantics.

It is possible to borrow some ideas from software modelling, and there is a great deal of existing research dealing with model-based testing strategies [5, 6]. However, much of this work focuses on the behavioural elements of software models (such as state machines), rather than the structural aspects which might be more relevant to metamodelling.

In the context of testing structural models, two tools strike us as being particularly noteworthy:

– **USE** (a UML-Based Specification Environment) which allows the user to specify a UML (Unified Modeling Language) class diagram with OCL constraints [7]. From these we can manually generate corresponding object diagrams, and the USE environment will check that these are valid instances, satisfying the relevant class invariants.
– **Alloy** which has its own logical specification language, corresponding roughly to the elements found in a class diagram, and automatically generates object diagrams that correspond to this specification [8]. One of the useful aspects of the Alloy tool is that it is designed to work with a number of different SAT solvers in order to test constraints and generate counter-examples.

As part of an earlier project, we have previously exploited the close relationship between the Alloy notation and UML class diagrams to generate instances of a metamodel for software metrics [9]. One of the drawbacks of this earlier approach is that it did not control the strategy used to generate instances. Faced with the impossibility of manually validating several hundred thousand instances, we exploited test-suite reduction techniques to narrow the set of test cases.

3 Proposed Solution

It is clearly inefficient to generate test cases that are not used, and an ideal solution would be to generate an appropriate set of test cases in the first place. Of course, this immediately raises two questions: what do we mean by an *appropriate* set of test cases, and how do we generate these test cases?

One way of measuring the adequacy of a set of test cases for a piece of software is to use coverage criteria. Typically, a test suite can be judged in terms of the percentage of statements, decisions, branches etc. in the source code that are executed when the software is run. It seems natural therefore to attempt to seek to assemble a test suite for a metamodel along similar lines, i.e. to form a set of models that cover the features of the metamodel. In terms of programming language metamodels, this would involve creating a set of programs that exercise the features of the language.

Since most metamodels, including programming language metamodels, are described using UML, it is possible to use UML techniques to compute their coverage. Many coverage criteria for the various UML diagrams have been proposed [10–13]. However, we restrict our attention to UML structural diagrams, since metamodels are often presented in the form of UML class diagrams.

Our recent work has focused on coverage criteria for UML class diagrams initially proposed by Andrews et al. [14], specifically:

- Generalisation coverage which describes how to measure inheritance relationships.
- Association-end multiplicity coverage which measures association relationships defined between classes.
- Class attribute coverage which measures the set of representative attribute value combinations in each instance of class.

It is unlikely that these alone will provide sufficient granularity to determine the adequacy of a test suite, and previous work has already determined that there is a poor correlation between coverage of syntactic and semantic features [15]. However, our immediate work has centred on constructing an extensible, modular system that can at least measure these levels of coverage, and which can be used as a basis for further studies.

4 Research Method

Our research can be broken down into three phases:
Phase I: calculating coverage measures for programming language metamodels,
Phase II: generating valid models that satisfy coverage criteria,
Phase III: generating models satisfying criteria *not* based on coverage.

In our work to date we have constructed a tool-chain which, when given a UML class diagram and a set of UML object diagrams, will calculate the three coverage measures described above [16]. The tool chain uses the USE tool as a parser and validator for the class and object diagrams, and uses the Eclipse Modeling Framework (EMF) to represent these as instances of the UML metamodel. To represent the output we have built a coverage metamodel in EMF which is essentially an extension of an existing metrics metamodel [17]. Finally, we have written a transformation to calculate the coverage measures using ATL (ATLAS Transformation Language).

In order to complete the first phase we intend to extend the coverage measures for class diagrams to deal with the associated OCL constraints. We intend to use (OCL) decision coverage as our initial measure, and to extend our tool chain to implement this coverage.

In the next phase, we hope to generate valid models that satisfy coverage criteria for at least one programming language metamodel. We are currently studying Alloy's approach as a model of exploiting third-party SAT solvers to control model generation.

At the final phase of our research we hope to expand this approach to other language-based criteria. To do this, we intend to use OCL-based queries across the metamodel to specify the kind of model to be generated. Ideally, this would allow a user to specify the kind of programs that would be generated in the test suite for a given language metamodel.

References

1. Purdom, P.: A sentence generator for testing parsers. BIT **12**(3) (1972) 366–375
2. Malloy, B.A., Power, J.F.: An interpretation of Purdom's algorithm for automatic generation of test cases. In: 1st Annual International Conference on Computer and Information Science, Orlando, Florida, USA (October 3-5 2001)
3. Lämmel, R.: Grammar testing. In: Fundamental Approaches to Software Engineering. Volume 2029 of Lecture Notes in Computer Science., Springer Verlag (2001) 201–216
4. Lämmel, R., Schulte, W.: Controllable combinatorial coverage in grammar-based testing. In: 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems, New York, NY (May 2006) 19–38
5. Pilskalnsa, O., Andrews, A., Knight, A., Ghosh, S., France, R.: Testing UML designs. Information and Software Technology **49**(8) (August 2007) 892–912
6. Utting, M., Legeard, B., eds.: Practical Model-Based Testing. Elsevier (2007)
7. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comp. Prog. **69**(1-3) (2007) 27–34
8. Jackson, D.: Software Abstractions. MIT Press (2006)
9. McQuillan, J.A., Power, J.F.: A metamodel for the measurement of object-oriented systems: An analysis using Alloy. In: IEEE International Conference on Software Testing Verification and Validation, Lillehammer, Norway (April 9-11 2008) 288–297
10. McQuillan, J., Power, J.: A survey of UML-based coverage criteria for software testing. Technical Report NUIM-CS-TR-2005-08, NUI Maynooth (2005)
11. Dinh-Trong, T.T., Ghosh, S., France, R.B.: A systematic approach to generate inputs to test UML design models. In: 17th International Symposium on Software Reliability Engineering, Raleigh, NC (2006) 95–104
12. Mahdian, A., Andrews, A.A., Pilskalns, O.: Regression testing with UML software designs: a survey. J. of Software Maintenance and Evolution: Research and Practice **21**(4) (July/August 2009) 253–286
13. Briand, L., Labiche, Y., Lin, Q.: Improving the coverage criteria of UML state machines using data flow analysis. Soft. Test. Verif. & Reliability **20** (2010)
14. Andrews, A., France, R., Ghosh, S., Craig, G.: Test adequacy criteria for UML design models. Soft. Test. Verif. & Reliability **13**(2) (April/June 2003) 95–127
15. Hennessy, M., Power, J.F.: Analysing the effectiveness of rule-coverage as a reduction criterion for test suites of grammar-based software. Empirical Software Engineering **13**(4) (August 2008) 343–368
16. Wu, H., Monahan, R., Power, J.F.: Using ATL in a tool-chain to calculate coverage data for UML class diagrams. In: 2nd International Workshop on Model Transformation with ATL, Malaga, Spain (June 2010)
17. Vépa, E.: ATL transformation example: UML2 to Measure. Available on-line as http://www.eclipse.org/m2m/atl/atlTransformations/#UML22Measure (August 30 2007)

Lenses for View Synchronization in Metamodel-Based Multi-View Modeling

Arif Wider

Humboldt-Universität zu Berlin
Unter den Linden 6, D-10099 Berlin, Germany
`wider@informatik.hu-berlin.de`

Abstract. When using multiple views to describe a system, the underlying models of these views have to be kept consistent, which is called *model synchronization.* Manually implemented model synchronizations that are not simple bijections are hard to maintain and to reason about. Special languages for expressing *bidirectional transformations* can help in this respect, but existing languages applicable in *model-driven engineering* are often restricted to bijections or complex to use. I adapt *lenses*, a promising term-rewriting-based approach to bidirectional transformations, to model synchronization. This allows for flexible view synchronization that can be integrated with existing metamodel-based technologies.

Introduction

Modeling a system using multiple views is a common means nowadays to break down the complexity of the system description. A prominent example for that is the architecture of the UML, providing multiple diagram types which serve as aspect-specific views on a system. Using multiple views to describe a system imposes the problem of *inter-view consistency.*

In *model-driven engineering* (MDE) different views on a system can be implemented as different *domain-specific languages* (DSLs). This way, the system description consists of an ensemble of models described using these languages. This is called *multi-view modeling* or *domain-specific multimodeling* [11, 10]. In multimodeling inter-view consistency is achieved by synchronizing these models.

Naively implemented model synchronizations, i.e., pairs of forward and backward transformations described in a general-purpose language, can be hard to maintain and to reason about because consistency of forward and backward transformations has to be ensured and transformations can be arbitrarily complex. Special languages for describing *bidirectional transformations* provide notations to describe consistency relations between models. From this notation a forward and a backward transformation can be automatically inferred so that the consistency of these transformations is ensured by construction. This is easy if the relation is a bijection but gets hard if it is neither surjective nor injective. Unfortunately, as the idea of a view is to hide information which is not aspect-specific, bijections hardly occur in a multi-view setting.

Bidirectional transformations are researched for a long time, e.g, in the graph transformation community using *Triple Graph Grammars* (TGGs) [18, 6]. With

QVT Relational[1], there is even a standard by the OMG for describing bidirectional transformations of metamodel-based models. Nevertheless, languages for describing bidirectional model transformations are still not widely used in MDE. Concerning QVT, Stevens points out semantic issues that could be one reason for this limited acceptance [19]. Another practical issue could be the weak situation regarding tool support for QVT Relations: Although QVT specification was completed in 2008, there are only few implementations and even those are not maintained regularly.

Lenses [8, 7] is a *combinator-based* approach to bidirectional transformations: Foster et al. provide small, well-unterstood bidirectional transformations (called lenses) and a set of combinators that allow more complex transformations to be composed from those smaller ones. This greatly improves extensibility and comprehensibility. Furthermore, a type system guarantees that composed lenses preserve certain properties of their sub-lenses. This combinator-based approach is possible because lenses are restricted to the asymmetric case where one of the two models to be synchronized is an abstraction of the other, i.e., the relation is at least surjective. This way, the problem of model synchronization resembles the *view update problem* that has been studied in the database community for decades [4].

In contrast to less restricted symmetric approaches like TGGs and QVT that were designed for transformations of graphs and models, respectively, lenses were designed for synchronization of tree-like data and were mainly implemented for string transformations [3], e.g., synchronization of XML-data. This poses some conceptual challenges, when attempting to use lenses for model synchronization in a metamodel-based setting.

Related Work

There are several approaches using bidirectional transformations in the context of MDE, but most of them use symmetric bidirectional transformations and therefore lack the combinator-based nature that can be achieved using an asymmetric approach (e.g., the *AToM3* Framework [1]). Among them are also some, that integrate with existing metamodel-based technologies, e.g., the *Tefkat* transformation engine [16] that is closely connected with QVT and integrates with the *Eclipse Modeling Framework* (EMF)[2]. Recently, Hettel et al. presented an asymmetric approach for using the SQL-like Tefkat language for round-trip engineering [12].

An approach quite similar to lenses which is also used for view synchronization is the work of Hu et al. [14, 17]. Somehow similar to my approach, Garcia proposed to use their work in a metamodel-based context [9]. However, in their approach, changes in a model have to be explicitly marked to be synchronized. This prevents agnostic integration with existing metamodel-based technologies.

[1] http://www.omg.org/spec/QVT/1.0/
[2] http://www.eclipse.org/modeling/emf/

Furthermore, there are some approaches to bidirectional transformations that are heavily inspired by lenses or extend lenses, but are not used for view synchronization: Hidaka combines lenses with a query language but not in a metamodel-based context [13].

Probably closest to my work is the work of Xiong [22, 21, 5], who integrated concepts of lenses into his work on bidirectional transformations of metamodel-based models, but he proposes an update-based approach in contrast to the state-based approach of lenses and his work does not focus on integration with existing metamodel-based technologies.

Approach

My approach is to use lenses for view synchronization in metamodel-based multi-view modeling environments. In order to achieve this, I want to show that

1. the advantages of lenses, especially their composability, can be leveraged when describing bidirectional transformations of metamodel-based models,
2. that in conjunction with a synchronization architecture that incorporates a common model, lenses can be beneficially used for view synchronization and
3. that this approach allows for straightforward integration with existing meta-model-based technologies.

Lenses for Bidirectional Model Transformations In order to use lenses for model synchronization, the concepts of lenses have to be bridged from the grammarware technological space that lenses originate from to the modelware technological space [20]. For this, the following challenges are to be solved:

– **Typing:** In the original lens framework typing is mainly used for ensuring that certain lens properties are preserved when composing lenses. In MDE, transformations usually transform models conforming to one metamodel so that they conform to another metamodel. Therefore, typing of input and output data of lenses is highly desirable for model transformations.
– **Ordered data:** Originally, lenses work either on unordered tree-like data or certain keys have to be defined to be able to synchronize changes regarding order. With metamodel-based models, i.e., in an object-oriented setting, there is the object identity as an implicit key. This can be used to propagate changes in order and other complicated changes back to the original model.
– **References:** Models in general are graphs because they can contain references, whereas lenses were designed for synchronizing tree-like data. A pragmatic solution could be to make use of the containment hierarchy that is provided by many metamodeling frameworks anyway.

A Lens-Based Model Synchronization Architecture While the restriction to asymmetric synchronization does not seem to be flexible enough for the general MDE setting, it fits well to view synchronization: Lenses can be used to asymmetrically synchronize view-models with a common model. This common model can be a shared abstraction, i.e., it only contains those information that is represented in more than one view. In the database community, this approach to view synchronization was already proposed by Atzeni & Torlone in 1996 [2]. Another approach is to synchronize views with a shared complete model of the system containing the information of all models to be synchronized. In both cases the changes made in one view-model are propagated to the other view-models through the common model.

Technological Integration As stated before, it is my goal to provide a solution that can be integrated with existing metamodel-based technologies, in particular, with the *Eclipse Modeling Framework* (EMF). As EMF is a Java-based framework, I decided to implement lenses for model transformations as an internal DSL in the *Scala*[3] programming language. Scala code compiles to JVM bytecode and Scala provides great interoperability with Java-based frameworks. Moreover, Scala combines functional and object-oriented concepts, which fits to the task of adapting lenses that come from a functional background to be used in a metamodel-based, i.e., object-oriented setting. Finally, Scala has static typing and it is my goal to achieve compile-time type checking for transformations in as many situations as possible. Therefore, I make use of *heterogeneously typed lists* [15], that were originally developed for the Haskell programming language. My solution will be deliverable as a Scala library. As a consequence, no further tools than the Scala compiler and a Scala IDE plug-in should be needed to integrate the solution into existing projects and tool chains. Hopefully, this results in higher user acceptance compared to solutions like QVT that always depend on up-to-date tool support.

Evaluation and Expected Contributions

The evaluation of my approach is tightly coupled with the ongoing development of a *domain-specific workbench* for the development of optical nanostructures, which is subject of a cooperation with a group of physicists. This workbench provides different DSLs for describing different aspects of experiments in nanostructure development and is being implemented with EMF-based technologies. This project serves as a comprehensive case study for my solution. In particular, it has to be evaluated if a reasonably sized set of basic lenses and lens combinators can be provided that enable to accomplish common view synchronization tasks in a concise way.

As a result of my work, the following contributions can be expected:

– An extended formal lens framework, adapted for an object-oriented setting.
– A language for bidirectional model transformations implemented as an internal DSL in Scala, deliverable as a Scala library.
– A lens-based view synchronization architecture that integrates with EMF-based technologies and can be used in multi-view modeling environments.

[3] http://www.scala-lang.org

References

1. Francisco Pérez Andrés, Juan de Lara, and Esther Guerra. Domain specific languages with graphical and textual views. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *AGTIVE*, volume 5088 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2007.
2. Paolo Atzeni and Riccardo Torlone. Management of multiple models in an extensible database design tool. In *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 1996.
3. Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, CA*, pages 407–419, January 2008.
4. U. Dayal and P.A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems (TODS)*, 7(3):381–416, 1982.
5. Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From state- to delta-based bidirectional model transformations. In *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*, volume 6142 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2010.
6. H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information preserving bidirectional model transformations. In *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007*, volume 4422, page 72. Springer, 2007.
7. J.N. Foster. *Bidirectional Programming Languages*. PhD thesis, University of Pennsylvania, 2009.
8. J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):17, 2007.
9. Miguel Garcia. Bidirectional synchronization of multiple views of software models. In Dirk Fahland, Daniel A. Sadilek, Markus Scheidgen, and Stephan Weißleder, editors, *Proceedings of the Workshop on Domain-Specific Modeling Languages (DSML-2008)*, volume 324 of *CEUR-WS*, pages 7–19, 2008.
10. Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wasowski. Guided development with multiple domain-specific languages. In *MoDELS*, pages 46–60, 2007.
11. Anders Hesselund. *Domain-specific Multimodeling*. PhD thesis, IT University of Copenhagen, 2009.
12. T. Hettel, M. Lawley, and K. Raymond. Towards model round-trip engineering: an abductive approach. *Theory and Practice of Model Transformations*, pages 100–115, 2009.
13. Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. A compositional approach to bidirectional model transformation. In *ICSE Companion*, pages 235–238, 2009.
14. Z. Hu, S.C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1):89–118, 2008.

15. Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.
16. M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In *Satellite Events at the MoDELS 2005 Conference*, pages 139–150. Springer, 2006.
17. K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, page 58. ACM, 2007.
18. Andy Schürr and Felix Klar. 15 years of triple graph grammars. In *ICGT*, pages 411–425, 2008.
19. P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proc. of the 10 Int. Conf. on Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 1–14. Springer, 2007.
20. M. Wimmer and G. Kramler. Bridging grammarware and modelware. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 159–168. Springer, 2005.
21. Yingfei Xiong. *A Language-based Approach to Model Synchronization in Software Engineering*. PhD thesis, Department of Mathematical Informatics, University of Tokyo, September 2009.
22. Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *ASE*, pages 164–173, 2007.

Analyzing Dynamic Models using a Data-flow based Approach

Christian Saad, Bernhard Bauer (Supervisor)
christian.saad@informatik.uni-augsburg.de

University of Augsburg

Abstract. Meta modeling as a method allows to devise languages suited for specific application domains, e.g. for describing the structural or behavioral aspects of software systems. As meta models constitute an abstract syntax (often enriched with static semantics) there exist obvious similarities to the area of formal languages. The research effort described in this paper is intended to examine to what extent and to what benefit compiler construction concepts, namely the data-flow analysis method, can be transferred to the modeling domain in order to validate static semantics and perform abstract interpretations on models.

1 Motivation

In the last decade, the use of meta models has evolved from a scientific approach to a widely used technique in areas like software development and business process modeling (BPM). This advancement has been greatly endorsed by industry standards like the OMG's Meta-Object Facility (MOF) framework that forms the basis for several prominent modeling languages like the the Unified Modeling Language or the Business Process Modeling Notation (BPMN) and even a model-centric development approach called Model-driven Architecture (MDA)[1].

Nevertheless, when compared to formal languages, an inherent flaw of the modeling technique becomes obvious: The restrictions on a language's structure which are given by its abstract syntax, i.e. a context-free grammar (CFG) or a meta model, are often not enough to ensure correct expressions - a check of the static semantics is also in order. For this purpose, CFGs are often extended with attributes, forming attribute grammars (AG), which enable compilers to validate statements based on the context in which they appear (cf. [1]).

A limited amount of static analysis can be performed using the OMG's Object Constraint Language (OCL, [7]) which enables to phrase constraints on the structure of MOF and UML-based meta models. However, it has several drawbacks: Through navigation statements, OCL constraints are tightly tied to the structure of the meta model. This can lead to difficulties if constraints require the consideration of a model element's context, e.g. the position of an action in an UML activity diagram relative to its preceeding/succeeding actions. As a

[1] http://www.omg.org/technology/documents/modeling_spec_catalog.htm

language which is mainly intended to be used to validate constraints by performing static queries on a model's elements, OCL does not contain semantics which would enable a fixed-point analysis since, in the case of cyclic dependencies this would require a continuous reevaluation the DFA's equation system. Finally, complex navigation statements make OCL rules very prone to be invalidated if the structure of the underlying meta model changes, often requiring extensive adjustments.

Since models comprise a graph structure, defining and calculating information flow enables an advanced analysis of a model's properties. These may either be properties of the graph structure itself, e.g. strongly connected component regions, or semantic attributes, e.g. the availability of a resource created at one point in a control-flow model in some other part of the model, This research is dedicated to adapting the data-flow analysis (DFA) approach commonly used in compiler construction for deriving optimizations from a program's control-flow graph, to modeling, thus enabling a fixed-point analysis on (meta) models.

2 Related Work

Since the definition of static semantics is a vital step when developing formal languages, many different techniques have been considered for this purpose.

The OCL can be considered a comparatively simple language for requirements exceeding syntactic expressiveness and by design is very well-integrated into the modeling domain. A critical evaluation of its expressive power can be found in [5], with emphasis on difficulties in calculating transitive closures.

Approaches considering the use of formal semantics include graph transformations, abstract state machines or first order logic (cf. [8] [6] [3]).

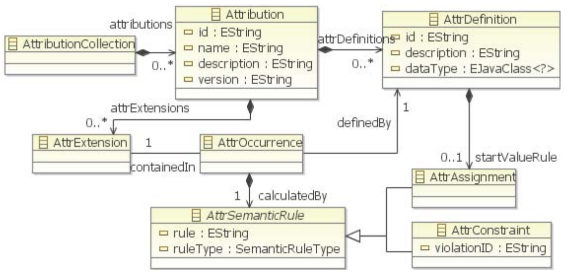
Several authors use DFA-based methods to derive information from models. In [9], a fixed-point calculation is used on executable models to derive *def/use* relationships between actions while the authors of [2] propose the use of control-flow information to improve software testing.

The focus in these cases is, however, directed towards implementing a specific case study. To the best of our knowledge, there is no other research effort with the goal of applying the concept of a fixed-point based DFA calculation to models.

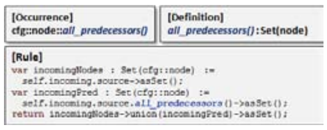
3 Approach for Model-based Data-flow Analysis

Both meta models and (context-free) grammars operate on different layers of abstraction. For grammars, these usually consist of EBNF, CFGs and language expressions. In the widely-used MOF one distinguishes between meta-meta (M3), meta (M2) and model (M1) layer. Because of conceptual similarity, both techniques can be aligned according to their levels of abstraction. This alignment requires that data-flow equations are attached to language artefacts (i.e. M2 meta classes) and instantiated/executed for expressions, i.e. M1 objects.

Since attribute grammars are a well-proven extension of this design, it was decided that this strategy will also be employed in the definition of data-flow equations: An *attribute definition* consisting of an identifier and a data-type can be bound to several meta model classes through the use of *attribute occurrences*. *Semantic rules* (corresponding to data-flow equations) are assigned to *attribute definitions* and *attribute occurrences* to calculate initialization and iteration values, respectively. Data-flow between attributes occurs if a *semantic rule* requests the value of another attribute as input.



(a) Data-flow analysis meta model



```
[Occurrence]
cfg::node::all_predecessors()
```

```
[Definition]
all_predecessors():Set(node)
```

```
[Rule]
var incomingNodes : Set(cfg::node) :=
  self.incoming.source->asSet();
var incomingPred : Set(cfg::node) :=
  self.incoming.source.all_predecessors()->asSet();
return incomingNodes->union(incomingPred)->asSet();
```

(b) Calculate predecessors

Fig. 1. Model-based DFA definition in the notion of attribute grammars

To stay consistent with the notion of modeling and to minimize frictional losses between different techniques, it is desirable to represent DFAs themselves as models, i.e. to provide a meta model which in the described alignment hierarchy acts as an extension of the M3 layer (cf. Figure 1(a)).

To calculate a DFA for a given model, attribute occurrences assigned to meta classes need to be instantiated for model elements derived from these classes. While the instantiation process is straightforward, it has to be noted that this must happen in compliance with principles like generalization, i.e. if defined for a super class, attributes must be inherited by instances of sub classes.

An example is shown in Figure 1(b) which calculates the transitive closure of predecessor nodes in a simple control-flow graph. The class *node* has an assigned attribute occurrence of the type *all_predecessors*. The associated rule, which is repeatedly executed for all *nodes*, recursively creates the union of direct predecessors and the value of *all_predecessors* at preceding nodes.

The fixed-point calculation significant differs from traditional DFA techniques: Since semantic rules may request the value of arbitrary attributes as input when they are executed, there exists neither an inherent flow direction nor any prior knowledge about output relationships between attribute instances. Since the commonly used worklist algorithm depends on this information, it is not applicable in this context. To accomodate for this, an algorithm has been developed that dynamically records input/output relationships by executing the rules recursively to create a dependency graph which can then be used as a basis to derive a nearly optimal execution order, thus minimizing the amount of rule executions.

4 Research

The research effort described in this abstract has to cover the following issues:

Definition and Alignment To transfer the method of DFA to the modeling domain, similarities and differences between both application areas must be identified and a new definition language has to be devised and aligned with the modeling techniques.

DFA Algorithms Suitable algorithms for calculating DFA equation systems have to be devised and proven to be correct.

Complexity and Performance The practical and theoretical performance of different DFA solving algorithms has to be evaluated.

Tooling and Use Cases To prove the feasability of this approach, a tooling environment has to be provided alongside the implementation of several use cases which need to be evaluated against implementations based on alternative theoretical foundations.

While efforts for DFA definition and integration into modeling are well advanced (as described in the previous section), a formalization of these results is still in order. The same goes for the DFA solving algorithm which has proven to perform very well in comparison to adaptions of conventional algorithms like the work list method.

The Model Analysis Framework (MAF[2]) is a fully functional prototype providing tooling support for the described concepts. It is built upon Eclipse modeling techniques, namely the Eclipse Modeling Framework (EMF), an implementation of the MOF standard. Once all artefacts required for an analysis (i.e. meta models, models and attributions based on the meta model shown in Figure 1(a)) are loaded into the respective repositories, the defined attributes are instantiated for the model's elements and handed to the selected evaluation algorithm. First experiments have shown that the developed algorithm preforms much better in calculating the result values than a traditional worklist algorithm that has been enhanced with the ability to handle dynamically discovered output dependencies.

Currently implemented case studies (which are available from the code repository) include: Calculating properties of control-flow graphs like transitive predecessor/successor sets and strongly connected components (SCC), definition/use relationships in workflows and the hierarchical subdivision of control-flows into its Single-Entry-Single-Exit (SESE) components using a token-flow algorithm (cf. [4]) which has been reimplemented using DFA. In the future, additional applications areas are planned like clone detection, validating modeling guidelines and generating test models for model-based testing approaches.

[2] `http://code.google.com/p/model-analysis-framework/`

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (August 2006), `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&amp;path=ASIN/0321486811`
2. Garousi, V., Bri, L., Labiche, Y.: Control Flow Analysis of UML 2.0 Sequence Diagrams (2005)
3. Georg, G., Bieman, J., France, R.: Using Alloy and UML/OCL to specify run-time configuration management: a case study. Practical UML-Based Rigorous Development Methods-Countering or Integrating the eXtremists 7 (2001)
4. Götz, M., Roser, S., Lautenbacher, F., Bauer, B.: Token Analysis of Graph-Oriented Process Models. New Zealand Second International Workshop on Dynamic and Declarative Business Processes (DDBP), in conjunction with the 13th IEEE International EDOC Conference (EDOC 2009) (September 2009)
5. Mandel, L., Cengarle, M.: On the expressive power of the Object Constraint Language OCL. Available on the World Wide Web: http://www. fast. de/projeckte/forsoft/ocl (1999)
6. Ober, I.: An ASM Semantics of UML Derived from the Meta-Model and Incorporating Actions pp. 356–371 (2003)
7. Object Management Group: Object Constraint Language. http://www.omg.org/spec/OCL/2.0/ (Mai 2006)
8. Varró, D.: A formal semantics of UML Statecharts by model transition systems. Graph Transformation pp. 378–392 (2002)
9. Waheed, T., Iqbal, M., Malik, Z.: Data Flow Analysis of UML Action Semantics for Executable Models. Lecture Notes in Computer Science 5095, 79–93 (2008)

Using SLE for creation of Data Warehouses

Yvette Teiken

OFFIS, Institute for Information Technology, Germany
teiken@offis.de

Abstract. This paper describes how software language engineering is applied to the process of data warehouse creation. The creation of a data warehouse is a complex process and therefore costly. My approach decomposes the data warehouse creation process into different aspects. These aspects are described with different languages which are integrated by a metamodel. Based on this metamodel, large parts of the data warehouse creation process can be generated. With my approach data warehouses are created more comfortable in less time.

1 Problem Description and Motivation

Health Reporting describes the preparation and presentation of health relevant issues relating to population. It is used to give information to stakeholders in the health care system, politicians and interested non-professionals. Furthermore, risks are identified and appropriate warnings issued. In the Federal State of North Rhine-Westphalia this task is carried out by the government agency for public health called LIGA (Landesinstitut für Gesundheit und Arbeit). LIGA provides a variety of different reports and is able to answer ad hoc questions. The reports are based on data from different sources and different systems with different formats. These sources are e.g. data from different public health departments or insurances. To fulfill these requirements, software support is needed. This means data has to be integrated into one dataset so that different sources can be related. Support is also needed for transforming data on regular basis into the integrated dataset. Also frontend and report generation has to be developed.

One software system providing this support is the MUSTANG platform developed at OFFIS. MUSTANG is used at LIGA. The base of each MUSTANG instance is an integrated dataset, this is also called data warehouse (DWH). In an integrated dataset all relevant organizational knowledge is stored for complex analysis. For fast data navigation and analysis, Online Analytical Processing (OLAP) [4] is often used. OLAP is an approach that allows navigation and querying data more comfortable than using exact queries like SQL. For OLAP a multidimensional data model is needed. The initial build-up of a DWH with a multidimensional integrated dataset is a complex task [8]. During the initial build-up of DWH the following analyzing and design steps have to be performed:

Analysis of organizational data: To find data that can be used in the resulting DWH, existing data sources have to be analyzed. This analysis includes the content, format and the accessibility of the data. This kind of data is called fact data. In a DWH this fact data is extracted and integrated into the so called integrated dataset. **Define information demand:** Define what information should be provided by the DWH. This can be simple figures or complex computations. **Data source transformation:** Fact data has to be transformed in the data format of the integrated dataset. Therefore, for every data source a transformation has to be designed that translates data into the format of the integrated data set and stores it there. **Define multidimensional data model:** Defines how fact data can be described multidimensionally and grouped together in hierarchies. **Data quality:** Based on DWH, analysis decisions are made so it is important to define data quality standards and how to identify invalid data.

To perform these steps no standardized process exists. Documentation of these steps is usually done with a large number of documents. A problem with this kind of documentation is missing, distributed or inconsistent information. Another aspect is that during realization a lot schematic work has to be done. For example, a multidimensional schema has to be designed and realized in the OLAP system, the integrated dataset, and at the frontend software.

2 Related Work

Data Warehouse analysis and design as described in [1] consists of different phases. For these different phases of the DWH creation, languages and tools have been developed. In case of multidimensional modeling, languages like Application Design for Analytical Processing Technologies (ADAPT) by [3] exist. There are also languages that describe mapping for relational databases like R2D [2] or languages that describe data quality issues like InDaQu [15].

Another field of related work is automated creation of DWHs. The feasibility to connect MDA with the DWH process has been shown in [12]. They also developed a MDA framework for DWH. It covers data integration, data sources, and multidimensional models. The authors show the application of their approach through a case study. However, their main focus are models and not languages.

More related to SLE is the work of Rizzi. His group deals with modeling different aspects of DWHs. For example in [13] modeling technique for data cubes and data flows are suggested and in [9] a UML based approach for *what if*-analysis is provided. Another work that deals with SLE and aspects of DWH creation is [7]. They use modeling languages to generate multidimensional schemas.

All these approaches only deal with a certain aspect of DWH creation, not with the whole process with language support. In my thesis, I will develop an approach with languages that cover the whole process of DWH creation. These languages are integrated through a common metamodel and can deal with multidimensional structures. Based on the metamodel I will create transformations that allow generating large parts of the resulting DWH. With these transformations, schematic work in the step of realization is reduced. Furthermore, I will create a process model that orders the steps described in combination with the developed languages to improve documentation. With the process model, the different aspects are connected and refined.

3 Proposed Solution

Data warehouses, as describes here, are very complex systems with different views, aspects, and levels of detail. To create a single language, these systems are difficult and not easy to use and maintain. Therefore, it is necessary to decompose a DWH creation into different aspects and create languages for each. The different languages will be used by different roles and provide a different level of detail. A first result is that the process can be decomposed in six aspects:

Data Sources Schemas: Describes all relevant or available data stored in its operational system of an organization. This aspect contains the subject, the representation, and technical accessibility. The advantage is that all relevant sources are described together with their formats and accessibility at one place. For this aspect the development of an own language may not be necessary but a meta model will be sufficient.

Data Source Transformation: Describes how data from the sources have to be transformed to match the analysis schema. Such a description makes it possible to abstract from the concrete target system and some automatic matching process can be applied.

Analysis Schema: This aspect describes the multidimensional schema of the resulting DWH. The multidimensional schema consists of cubes and dimensions. In a cube, fact data is stored. Each cell represents fact data that is numerical and can be aggregated. They are described by dimensional metadata. In a cube with a time dimension, fact data is stored for dates so monthly and yearly values are computed by aggregation. The language is based on ADAPT. With such a language it is possible to reduce schematic work in the realization phase, as shown in [17], and it can be used to communicate with domain experts.

Measures: Describes what kind of information is intended to be stored in the DWH. This can be simple fact data like infections. It also includes the designated granularity of information. For example, to predict an epidemic infect, data should be available for every date. Measures link fact data with mathematical operations. In Health Reporting, these are mostly crude rates, interest, and average. Measures are usually defined by domain experts, in case of LIGA by epidemiologists. Measures are refined in the analysis schema. With a language for measures the definition can be used in the realization process and does not need to be reimplemented.

Hierarchy: The hierarchy aspect is a central one in my thesis because all other aspects use this directly or indirectly. This aspect describes how data is aggregated. In most cases hierarchies have many members and a complex structure and they are used in the multidimensional model. For example, imagine a geographical dimension that contains countries and cities. Using an own language, these structures can be modeled appropriate. It can help to build a repository that can be reused in a different DWH. The concrete syntax of the hierarchy language is based on ADAPT. However, it only allows to model hierarchies conceptually. To model hierarchies logically a tabular extension was created. To create parent-child relationships a query language was also integrated.

Data Quality: In a DWH it is important to ensure that certain quality issues are met. Naturally, data quality is an aspect of the data sources but when integrating different systems it would be very costly to deal with quality at the sources because many different systems have to be considered and changing these systems is rarely possible. Data quality is a large research field. In my approach, I want to integrate existing approaches to show how data quality issues can be integrated. InDaQu [15] is integrated to deal with data consistency.

The languages for the described aspects are developed independently. Each language is developed with SLE techniques [6] and based on tools like EMF [14] and MS DSL Tools [5]. To generate a DWH, the different languages and their metamodels have to be integrated into one metamodel because the different aspects are very strongly related. Furthermore, to be able to cover the whole process refering elements from other aspects is necessary.

The integrated metamodel covers all aspects of a DWH at one place. A common metamodel is a standardized documentation of the whole DWH system. Other possiblities of an integrated metamodel have been shown in [16]. To integrate a metamodel, [10] suggests two ways, via transformation and via common elements. I decided to use integration via unidirectional transformation for integration into metamodel. I created my DSLs with MS DSL Tools but for better analysis and transformation I move the models to EMF. The common metamodel consists of different separated metamodels. These are kept in different files, as suggested in [11]. The different metamodels are integrated via common elements and the instances via soft references.

The advantage of using SLE for the creation of DWHs is that experts can design and analyze all aspects of the DWH independently in adequate domain specific languages. Using the integrated metamodel the generation of a DWH is easier because all information and transformations are in that single model.

4 Research Method

My hypothesis will be validated via implementation. I will implement the described languages, metamodels, and transformations on basis of the MUSTANG platform. My prototype will be able to generate a configuration for a DWH. I will regenerate parts of the LIGA DWH that was developed by OFFIS. I will compare the steps taken. When using my approach these steps will be reduced.

5 Conclusion

Currently I have developed three languages for hierarchies, analysis schema, and data quality. I integrated them in a common metamodel. Based on this, transformations for generating multidimensional schemas in databases and integration interface with consistency were built. The next action is to develop languages for data sources and data integration as well as the extension of the common metamodel. The current state of my thesis shows that modeling and generation of data warehouses can be possible and reasonable with SLE.

References

1. Bauer, A., Günzel, H.: Data-Warehouse-Systeme. Architektur, Entwicklung, Anwendung. Dpunkt Verlag (2008)
2. Bizer, C.: D2R MAP - a database to rdf mapping language. In: WWW (Posters) (2003)
3. Bulos, D.: OLAP database design: A new dimension. Database Programming&Design Vol. 9(6) (1996)
4. Codd, E.F., Codd, S.B., Salley, C.T.: Providing OLAP to User-Analysts: An IT mandate. White paper, E.F. Codd Associates (1993)
5. Cook, S., Jones, G., Kent, S.: Domain Specific Development with Visual Studio DSL Tools (Microsoft .net Development). Addison-Wesley Longman, Amsterdam (2007)
6. Favre, J.M., Gasevic, D., Lämmel, R., Winter, A.: Editorial - software language engineering. IET Software 2(3), 161–164 (2008)
7. Gluchowski, P., Kurze, C., Schieder, C.: A modeling tool for multidimensional data using the adapt notation. In: HICSS. pp. 1–10. IEEE Computer Society (2009)
8. Golfarelli, M.: Data Warehousing Design and Advanced Engineering Applications: Methods for Complex Construction, chap. From User Requirements to Conceptual Design in Data Warehouse Design - a Survey. Information Science Reference (2009)
9. Golfarelli, M., Rizzi, S.: UML-Based modeling for What-If Analysis. In: DaWaK '08: Proceedings of the 10th international conference on Data Warehousing and Knowledge DiscoveryMazon. pp. 1–12. Springer-Verlag, Berlin, Heidelberg (2008)
10. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons (2008)
11. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Longman (2008)
12. Mazon, J.N., Trujillo, J., Serrano, M., Piattini, M.: Applying MDA to the development of data warehouses. In: DOLAP '05: Proceedings of the 8th ACM international workshop on Data warehousing and OLAP. pp. 57–66. ACM, New York, NY, USA (2005)
13. Pardillo, J., Golfarelli, M., Rizzi, S., Trujillo, J.: Visual modelling of Data Warehousing Flows with UML profiles. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK. Lecture Notes in Computer Science, vol. 5691, pp. 36–47. Springer (2009)
14. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley Longman (2008)
15. Teiken, Y., Brüggemann, S., Appelrath, H.J.: Interchangeable consistency constraints for public health care systems. In: Shin, S.Y., Ossowski, S., Schumacher, M., Palakal, M.J., Hung, C.C. (eds.) SAC. pp. 1411–1416. ACM (2010)
16. Teiken, Y., Flöring, S.: A common meta-model for data analysis based on dsm. In: The 8th OOPSLA workshop on domain-specific modeling (2008)
17. Teiken, Y., Rohde, M., Appelrath, H.J.: Model-driven ad hoc data integration in the context of a Population-based Cancer Registry. In: ICSOFT 2010 (2010)

Domain-Specific Languages for Digital Forensics

Jeroen van den Bos

Centrum Wiskunde & Informatica
Nederlands Forensisch Instituut
jeroen@infuse.org

Abstract. Due to strict deadlines, custom requirements for nearly every case and the scale of digital forensic investigations, forensic software needs to be extremely flexible. There is a clear separation between different types of knowledge in the domain, making domain-specific languages (DSLs) a possible solution for these applications. To determine their effectiveness, DSL-based systems must be implemented and compared to the original systems. Furthermore, existing systems must be migrated to these DSL-based systems to preserve the knowledge that has been encoded in them over the years. Finally, a cost analysis must be made to determine whether these DSL-based systems are a good investment.

1 Problem Description and Motivation

Digital forensic investigations are nearly exclusively performed using software, but high pressure in the form of strict deadlines combined with case-specific requirements severely complicates its use. The problems associated with constantly changing software are well-known [11]. This research attempts to address this problem by introducing and migrating to DSL-based systems.

The use of software in digital forensics is the result of constantly increasing storage device sizes (a gigabyte halves in price every fourteen months [10]), increasing connectivity (the amount of households having a broadband connection has more than doubled in the past five years [4]) and the pervasiveness of digital devices (currently there are more active mobile phones in The Netherlands than there are citizens [4]).

This explosion in connectivity and storage capacity has many advantages, but also drawbacks, one of them being the increased use of these capabilities by criminals. This in turn has caused both the amount and scale of digital forensic investigations to explode. Extensive automation appears to be the only feasible approach to deal with these increases, as manual inspection of even a single gigabyte for possible evidence could take years to complete.

Some things haven't changed however, including legal requirements around (pre-charge) detainment of suspects. This means that forensic investigations nearly always have very strict deadlines. Additionally, the variety of devices, applications and communication channels ensures that digital forensic investigations typically require custom tools.

Application-specific knowledge, in the form of communication protocols, storage device layouts and embedded systems implementation details are often case-specific while the software used to recover data are tools implementing general algorithms.

The algorithms rarely have to change (although new ones regularly emerge), but the specifications they are working with constantly change.

Solutions exist that employ methods of software reuse and abstraction to reduce modification time, but in practice it turns out that these still require software engineers to actually make the changes. In a situation where a large amount of changes to software must be made within a couple of days, the process of transferring knowledge to software engineers who then make the actual changes is time-consuming, error-prone and hard to trace.

2 Brief Overview of Related Work

Extensible digital forensic applications exist, such as TULP2G [2] for analyzing embedded devices and ReviveIt [14] and Scalpel [16] for recovering lost and hidden files from storage devices. They all have similar limitations however in that they tangle implementation of application-specific knowledge (e.g. file formats and memory layouts) and recovery algorithms. In TULP2G, the application-specific knowledge in the form of communication protocols must be defined in an imperative language along with the recovery algorithms. Both ReviveIt and Scalpel use an external notation to specify application-specific knowledge in the form of file format specifications, but their notation is heavily based on the actual recovery algorithms used, making it difficult to develop new algorithms without changing the search patterns.

There is a lot of research in interpreting structured data. Parsing techniques [7] using grammar formalisms such as ANTLR [15] and SDF2 [18] are targeted at textual programming languages however and lack features to support complex data dependencies between elements in a protocol, file format or memory structure. Data-dependent grammars extend traditional parsing technology to allow the definition of such dependencies [8]. These grammars are used to derive parsers for Data Description Languages (DDLs) [6]. PADS/ML [12], DataScript [1] and Zebu [3] are all examples of a such DDLs. These DDLs are typically positioned as productivity-enhancing tools for programmers, making them less suitable for use in forensic investigations where users are often reverse engineering data structures instead of developing them from scratch.

There is extensive work in the area of developing DSLs [5] [17] [13] [9], however some open questions remain such as how to use knowledge from existing systems in their design.

3 Proposed Solution

Our proposed solution is to develop one or more DSLs to meet the challenges posed by digital forensics. Model-driven engineering in general and DSLs in particular may be a good fit for digital forensic software.

3.1 General Approach

An analysis of both literature and practice shows that digital forensic investigations often have multiple concerns that may be valuable in applying a model-driven approach:

Application-specific knowledge Information that is application-specific refers to knowledge about devices, formats and protocols that are specific to a certain brand, type, standard or version of something. For example, when analyzing communication streams, this includes the protocols and compression and encryption methods. When analyzing a digital storage device (e.g., a hard drive), this includes the disk layout, file systems and file formats. When analyzing an embedded system (e.g., a mobile phone), this includes the memory layout.

Recovery algorithms Methods that belong to this category are techniques that take application-specific knowledge and use algorithms to recover data from the actual stream, storage or embedded device. They are often specific for a certain area of digital forensics, such as reassembly and identification algorithms for recovering deleted or hidden files from a confiscated hard drive. Although specific for a certain area, they are independent of specific changes to a format or version.

The proposed solution to increase usability and modifiability is to create one or several DSLs for forensic investigators. This will allow them to express application-specific knowledge that they obtain during their investigations in a language that is appropriate to them and abstract away all implementation details such as recovery algorithms. These in turn will be implemented in general-purpose languages and be maintained by software engineers.

However, to be able to do this several research questions must be addressed, such as:

1. How can qualities such as flexibility and adaptability be compared between an existing system and a DSL-based system?
2. How can the difference in development and maintenance costs between an existing system and a DSL-based system be measured?
3. How should the form of a DSL be determined?
4. How can the knowledge encoded in an existing system be used in the design of a DSL?
5. How can a DSL be developed so that it can be maintained by general developers?

3.2 Example Application

An example application of the general approach discussed is a file carver. File carving is the process of recovering deleted or damaged files from data storage devices (e.g. hard drives). The implementation of file systems allows deleted files to often be recoverable, although the contents of these files may be spread out across different sections of a device. To undo this so-called fragmentation, file carving algorithms use heuristics, operating system implementation details and file format recognizers to attempt to recover complete files.

If these recognizers are implemented using a general-purpose language, their method of recognizing file types is typically hard-coded in each instance. Furthermore, they may become tangled with the implementation of the recovery algorithms. A result is that changing a recovery algorithm or adding a new file format requires a significant software engineering effort.

However, using a data description language to define file formats along with a code generator to transform these declarative descriptions into recognizer implementations may reduce the required effort significantly. Furthermore, separating definition of file formats from the implementation of recovery algorithms may make it easier to modify or add new algorithms afterwards. Finally, a code generator may perform additional optimizations that would be difficult to perform manually (such as take several file format definitions and generate a single recognizer, improving scalability by reducing reads).

4 Research Method

To validate our hypothesis and answer the research questions, the following steps will be undertaken:

DSL Development Experiments will be performed by implementing one or several DSLs in the digital forensic domain. This will require experiments in the area of domain analysis as well.

DSL Validation The implemented DSL-based systems will be validated by comparing them in general use to existing digital forensic software, to determine whether the approach is viable in areas such as functionality, runtime performance and flexibility.

Automated Analysis Several techniques will be employed to aid DSL development:

- Automated model extraction to assist DSL design.
- Automated comparison between the existing systems and the DSL-based systems, to find implementation differences and preserve knowledge.

5 Conclusion

To keep up with the size of storage devices, speeds of network connections and amount of digital devices in use, digital forensic investigations rely heavily on custom software applications to perform large parts of analyses. However, the continuous introduction of new consumer applications and devices requires forensic software to be exceptionally flexible and adaptable.

To realize these requirements, using domain-specific languages to raise the level of abstraction and separate different concerns in the domain may be a viable approach. However, this requires analysis, design and implementation of systems employing these techniques as well as evaluation to compare existing systems to these alternative solutions.

This research will perform these steps by implementing one or several DSL-based systems, comparing them to existing systems to determine their relative performance and employ automated analysis techniques to aid in the design and preserve knowledge.

References

1. Back, G.: DataScript—a specification and scripting language for binary data. In: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GPCE'02). LNCS, vol. 2487, pp. 66–77. Springer (2002)
2. van den Bos, J., van der Knijff, R.: An Open Source Forensic Software Framework for Acquiring and Decoding Data Stored in Electronic Devices. International Journal of Digital Evidence 4(2) (2005)
3. Burgy, L., Reveillere, L., Lawall, J.L., Muller, G.: A language-based approach for improving the robustness of network application protocol implementations. In: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'07). pp. 149–160 (2007)
4. Centraal Bureau voor de Statistiek: De Digitale Economie. CBS (2009)
5. van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices 35(6), 26–36 (2000)
6. Fisher, K., Mandelbaum, Y., Walker, D.: The Next 700 Data Description Languages. Journal of the ACM 57(2), 1–51 (2010)
7. Grune, D., Jacobs, C.: Parsing Techniques—A Practical Guide. Springer (2008)
8. Jim, T., Mandelbaum, Y., Walker, D.: Semantics and Algorithms for Data-Dependent Grammars. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10). pp. 417–430. ACM (2010)
9. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press (2008)
10. Komorowski, M.: A History of Storage Cost (2009), `http://www.mkomo.com/cost-per-gigabyte`
11. Lehman, M.: Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68(9), 1060 – 1076 (1980)
12. Mandelbaum, Y., Fisher, K., Walker, D., Fernandez, M., Gleyzer, A.: PADS/ML: A Functional Data Description Language. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07). pp. 77–83. ACM (2007)
13. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. ACM Comput. Surv. 37(4), 316–344 (2005)
14. Metz, J.: ReviveIt 2007, `http://sourceforge.net/projects/revit/`
15. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf (2007)
16. Richard, III, G.G., Roussev, V.: Scalpel: A Frugal, High Performance File Carver. In: Refereed Proceedings of the 5th Annual Digital Forensic Research Workshop (DFRWS'05) (2005)
17. Spinellis, D.: Notable design patterns for domain-specific languages. Journal of Systems and Software 56(1), 91–99 (2001)
18. Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis, University of Amsterdam (1997)

Towards Better Support for Pattern-Oriented Software Development

Dietrich Travkin

Software Engineering Research Group,
Heinz Nixdorf Institute & Department of Computer Science,
University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany,
travkin@upb.de

Abstract. Design patterns document approved solutions for recurring design problems. Due to their vague description such patterns are widely applicable, but their application is error-prone. Since pattern applications are rarely documented, the originally intended design tends to deviate during software evolution. With my research I aim at explicitly modeling and validating pattern applications in design models in order to maintain an up-to-date documentation and reduce design deviation.

1 Introduction

When developing or adapting software, developers have to solve similar design problems over and over again. In order to reuse good solutions for common design problems, experts have documented approved solutions as *software patterns*. One of the most famous summaries of software patterns are *Design Patterns* described by Gamma et al. [7]. A design pattern has a unique name and consists of a problem description, the intent, a general description of the design that solves the problem, and consequences.

For the purpose of making such solutions reusable in many cases, they are described informally and in a very general way. Instead of documenting a concrete design, only the main idea is described. Unfortunately, this results in a lot of work and complicated decisions that are necessary to transfer a solution to and refine it for a concrete software design. The same holds for other software pattern descriptions, too [3, 2, 10, 8, 1].

Any pattern's exemplary design described in literature has to be manually adapted to the existing software design, i.e. a developer has to identify design parts that are to be added or adapted. The resulting design changes are usually performed manually. Furthermore, once a software pattern is applied, it is rarely documented or the current design deviates from its documentation due to subsequent design changes. This may result in design erosion [12] and the loss of the original developer's intent (e.g. to decouple certain software parts, make them replaceable and thus make the software more flexible).

My research aims at the development of methods and tools for model-driven software development that support software engineers in formally specifying software patterns, flexibly applying such patterns to an existing software design model, documenting the pattern applications, and validating the design in case of changes affecting the pattern applications.

2 Related Work

Since the introduction of software patterns, esp. design patterns [7], about 15 years ago, several researchers tried to formally specify patterns [11], e.g. in order to develop tools that automatically or interactively apply a pattern to an existing design or check whether the design conforms to a pattern. Most of them use sets of logic formulas to describe a pattern. But in many cases formulas are less comprehensible dable than graphical design description languages like UML, where relations are clearly visualized.

An example for a graphical and yet formal pattern specification language is the LePUS language [9, 11]. The proposed approach enables developers to check if a pattern implemention complies with its formal LePUS specification. A major drawback is the need for extensive user interaction before a conformance check can be performed.

There are also some approaches where patterns are specified in UML or UML-like languages. One of them uses specialized class and sequence diagrams as well as OCL [6, 11]. However, currently there is no tool support for validation of applied patterns.

Furthermore, none of the approaches mentioned so far [11] provides support for application of patterns in an existing design and applied patterns cannot be documented in design models.

An early research result [5] enables developers to specify their own patterns, to apply patterns in an existing Smalltalk application, and to check previously specified constraints in order to ensure that the implementation complies with the pattern. Nevertheless, the pattern application and validation operations have to be implemented manually, which complicates the specification of new patterns. Furthermore, they are applied to code, not to design models which are preferred in a model-driven software development process.

Commercial tools like IBM Rational Software Architect[1], Sparx Enterprise Architect[2], and Borland Together [3]also use UML to specify software patterns. In the Rational Software Architect, pattern applications are explicitly modeled and thus documented in UML models. Applying a pattern modifies a UML model and creates classes, methods, etc. But for each new pattern an expert has to manually implement the pattern application operations. Enterprise Architect and Together only save exemplary design parts in order to re-instantiate them later. Enterpise Architect in addition provides rudimentary functionality

[1] http://www-01.ibm.com/software/awdtools/swarchitect/websphere
[2] http://www.sparxsystems.de
[3] http://www.borland.com/us/products/together/index.html

to merge such an example model with an existing design. Besides, neither of these tools supports validation of applied patterns.

3 Proposed Approach

With my research I focus on model-driven software development. I assume that there is a design model that is created by a developer and used for code generation. Software patterns will be applied on such a model and will be documented and checked in such a model.

In order to formally specify software patterns, I propose a language that is capable of describing the pattern structure and most of the corresponding behavior as well as constraints that define the properties to be preserved after a pattern application. In order to base my language on concepts such as types, attributes, operations, type relations like associations and inheritance, as well as behavioral aspects like read or write access, delegation, etc., I restrict the design models to object-oriented models like the UML. In addition, I plan to support compositions of pattern specifications in order to increase reuse and to simplify the specification of complex patterns.

I am currently developing a meta-model and a graphical syntax for the pattern specification language that defines all language constructs. A first draft of a pattern specification based on this language is illustrated in Figure 1.

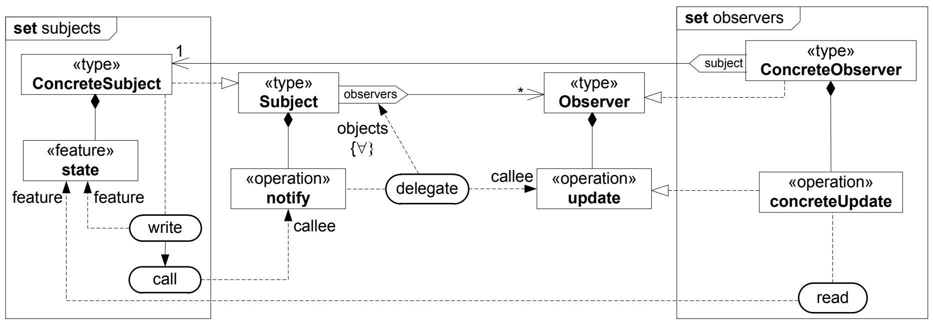


Fig. 1. Sketch of an Observer pattern specification

As an example, I specified the Observer pattern [7]. Between the types *Subject* and *Observer* a reference *observers* with cardinality * is specified. The subject's *notify* operation delegates its functionality to an observer's *update* operation, which is illustrated by a *delegate* node and corresponding arrows. In contrast to other approaches, it is also specified on which objects the *update* operation is to be called: in this case on each *Observer* object (denoted as {∀}) referenced by the *observers* reference. Moreover, control flow can be specified as well. For example, a write access to the *state* feature (this can be an attribute or a reference) is followed by a call of the *notify* operation (denoted as solid arrow between the nodes *write* and *call*). Set fragments (rectangles labelled with *set*) specify groups of elements that as a whole can be created multiple times in the target design model, e.g. a concrete observer type with a corresponding concrete update operation can be created several times.

I plan to provide tool support for the specification of patterns as well as their semi-automated application. In order to apply a pattern in an existing design model (e.g. a model of classes), a developer first specifies which parts of the design he would like to reuse. For that purpose, the developer maps existing design model elements (e.g. existing classes) to the pattern roles (nodes in Figure 1) that are to be played. In a next step, the tool will automatically adapt the design model by creating new model elements for each role that is not mapped. This way, the pattern application remains flexible and the developer decides in each situation what is going to be re-used or created anew.

The actual pattern application will be performed by means of a model transformation which is generated based on the selected pattern and the selected design model elements to be re-used. Due to their formal semantics, available tool support, and our research group's expertise, I plan to use so-called *story diagrams* [4] to specify the transformations. Story diagrams are specialized UML activity diagrams where the actions are graphically specified as graph grammar rules. These rules describe the object structures to be found and corresponding structure modifications.

Pattern applications will be documented as first-class constructs in the target modeling language so that developers do not loose track of applied patterns. This can be done by annotating the original design model (e.g. a class diagram) and visualizing the elements involved in a pattern application as well as the roles they are playing. For this purpose, I will provide an adequate notation and annotation mechanism.

If the design model is changed after a pattern application, the affected design elements are to be checked if they still comply with the pattern specification. For example, it can be checked if once created model elements still exist and if the specified constraints are satisfied, e.g. the design complies to specified access restrictions.

4 Summary and Evaluation Idea

Tool support as described above can significantly ease software development. By applying patterns, design solutions can be re-used on a high level of abstraction, the pattern applications are automatically documented in the model, and corresponding constraints are automatically checked after design modifications so that the original intent is preserved. The major goals of this approach are improved comprehendability of design models and reduced design deviation.

I plan to provide a prototype and to apply my development methodology to a realistically-sized software project in order to assess the benefits and drawbacks of my approach. Furthermore, I plan to compare traditional and the proposed pattern-oriented software development in an experimental setting.

References

1. Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns – Best Practices and Design Strategies.* Core Design Series. Prentice Hall, Sun Microsystems Press, 2 edition, 2003.
2. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*, volume 1 of *Software Design Patterns.* John Wiley and Sons, Ltd, 1996.
3. James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*, volume 1 of *Software Patterns Series.* Addison-Wesley, 1995.
4. Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations (TAGT'98)*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998.
5. Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool Support for Object-Oriented Patterns. In *ECOOP'97 – Object-Oriented Programming*, volume 1241/1997 of *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, 1997.
6. Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, 2004.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, 1995.
8. Robert S. Hanmer. *Patterns for Fault Tolerant Software.* Software Design Patterns. John Wiley and Sons, Ltd, 2007.
9. Jonathan Nicholson, Epameinondas Gasparis, Amnon H. Eden, and Rick Kazman. Automated Verification of Design Patterns with LePUS3. In *1st NASA Formal Methods Symposium, Moffett Field, California*, April 2009.
10. Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns.* John Wiley and Sons, Ltd, 2000.
11. Toufik Taibi, editor. *Design Pattern Formalization Techniques.* IGI Publishing, Hershey, PA, USA, 2007.
12. Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *The Journal of Systems and Software*, 61(2):105–119, 2002.

Using Product Lines to Manage Variability in Mobile Context-Aware Applications

Dean Kramer

School of Computing and Technology
Thames Valley University, London, UK, W5 5RF
dean.kramer@tvu.ac.uk

1 Introduction and Motivation

Today, the penetration of modern smart phones is vastly increasing with over 172 million smart phones shipped worldwide in 2009 [5]. It is quickly becoming predictable that these smart phones will contain sensors such as Global Positioning System (GPS) receivers, accelerometers, close proximity sensors and digital compasses. These components make smart phones a good candidate for pervasive computing and context-awareness. Context-awareness can be described as a systems ability to adapt and alter its behavior based on the situation it is current aware of [9]. Pervasive computing and context-awareness are becoming a highly interested area as it allows for applications to become more intelligent and more usable, unlike traditional software applications which would only respond to direct user input [6].

People and clients using software systems are becoming increasingly demanding regarding configuration and system features, thus requiring customization. In software development, re-developing software from scratch for each difference is purely uneconomical. This can be solved using Software Product Lines (SPL), creating many similar products from a single set of core assets [12]. Systems can be customized using variation points within its design, thus allowing for variants to shape the final product based on the configuration. This variability can be modeled using commonly used feature-modeling, which can help express different required features and variations of features. Because of the fragmentation of hardware/firmware, firmware/hardware constrained features should be modeled identifying this dependency which currently is not supported in feature modeling.

Though a SPL approach offers a viable approach to handling variability, deriving products from SPLs can be a challenging task. To compliment SPLs, the use of Model-Driven Development (MDD) and Domain Specific Languages can help allow for model transformations.

Domain Specific Languages (DSL) allow the developer to raise abstraction away from the software implementation making development easier. DSLs can be used within SPL engineering [14], and used to help provide model transformations. These transformations include horizontal (model to model), and more importantly in my case, vertical (model to code).

In this research, I propose to help integrate the modeling of context, features and their dependencies on hardware and firmware platforms. To compliment this, a DSL will be defined as method of expressing the model, which can then provide model-code transformations providing large amounts of application structure and features based on the domain feature implementations and variations.

2 Related Work

Developing for mobile platforms vastly differs to desktop and larger systems. Firstly, there are many different constraints that need to be considered. These constraints include lack of screen size, processing power, memory, and power consumption [8]. Though this idea may look out-dated, the expediential growth in software complexities and sizes causes this to remain true. Furthermore single application development is becoming increasing rare, with the trend more on a set of similar applications. Because of this, and increasing software complexities, the management of this re-usable software needs to be made simpler. The use of product lines is already evident in mobile gaming [10].

2.1 Context-Awareness

Much work in the past for aiding the production of context-ware applications has been through the developments of frameworks. The Java Context-Aware Framework [1] is a lightweight Java-based framework for creating context-aware applications. The feasibility of using this framework is help backed by its use in medical scenarios [2]. An issue with using this approach is the dependence on an external context-service, which for smart phone applications is not a desired method and can also bring security issues when transmitting sensitive data. Context-reasoning and challenges of mobility have been addressed with the use of a sentient object model [3].

MDA approaches have been proposed for context-aware software development. CAMEL (Context Awareness Modeling Language) [13], a DSL design for the initial stage of software development provides a method for modeling context dependent behaviors. Authors pointed out that currently the language does not handle model transformations to executable code.

2.2 Software Product Lines

UbiFEX [7] has been presented as a feature model notation to context-aware SPLs. This notation allows for context rule specific product configuration, but does not complete the need for modeling the complete system combining features and contexts.

Modeling context and its adaptation requires modelling within a product line, to help indicate how it relates to each feature. Parra et al. [11] creates a composition of assets binding context adaptation to features. This was later used with the FraSCAti platform[1], an open source implementation of the Service Component Architecture standard. This approach is similar to my plans, but there is a lack of concern for issues relating to how hardware/firmware may constrict different contexts being monitored and how features within the application may/may not be compatible with differing requirements. Furthermore, though the use of dynamic product line derivation is a useful approach for larger systems, because of different issues relating to mobile development a less dynamic derivation may prove more suitable. Firstly if a high amount of variation is in the application, this may 'bloat' the application unnecessarily, taking up already limited space on the device.

Despite the research being carried out into making product lines easier to implement, the use of SPLs for asset re-use is still having low adoption rates. This is because it affects the whole software life cycle, compared to other methods including Model Driven Development and Service Oriented Architecture [4]. Reasons for the lack of adoption include hardware/software integration issues, unpredictability caused by the recession and lack of SPL experts and low cost training.

3 Proposed Research

The primary aim of this research is to experiement and explore with a DSL to help support product derivation in context-aware mobile software product lines. This product-derivation will need to handle platform version differences, hardware differences and context differences. The proposed research will be a combination of action research and case-study based research. Within the research, two case studies will be developed, creating differing product-lines. The first of these case studies will be on a sports application, with the second more focusing on a mobile enterprise application.

3.1 Research Objectives

1. Investigate how you express the context aware environment and it's impact on application development. Modeling and defining the contexts for the product line and how it relates to the main application logic will be required. Metamodels for context, platform and application will be developed which will form the basis for the DSL.
2. Research into how to manage variability with the different contexts that will be supported within the application. Also, variability will need to be mapped and modeled to deal with different smart-phone hardware components and firmware versions.
3. Acquirement and analysis of DSL requirements, primarily driven through the development of two software product lines.

[1] http://frascati.ow2.org

4. Define and develop a DSL to help enable model-code transformations using Domain Specific Modelling (DSM) and implement over several iterations to suit case studies.
5. Use or develop an analysis framework for validating the proposed DSL. This validation will help ensure that the language meets all the requirements of a language.

3.2 Product Line Approach

The approach I propose for the product line is shown in Figure 1. This approach is made up of a product line (PL) meta model, and the DSL to specialise and deviate products into specialised applications (App). To help gain requirements of the DSL, meta-models will be developed including:

– Context Metamodel, a model indicating the different conditions that will constitute a context including user and environmental conditions.
– Platform Metamodel, a model showing the similarities and variability within the platforms. This will include hardware and software differences.
– Application Metamodel, a model describing the application. This will include different components and features included, with emphasis on core and additional/alternative parts of the application.

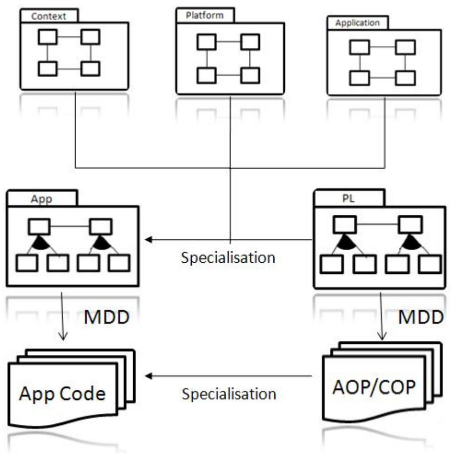


Fig. 1. Proposed Product Deviation Approach

For code generation, the PL will have software assets produced by MDD using a mix of Aspect-Oriented Programming (AOP) and Component-Oriented Programming (COP). Using aspects can easily seperate application concerns, in this case different platforms and/or contexts. These assets are then specialised using the DSL to produce partial application specific code, to which further implementation can be added.

References

1. Bardram, J.E.: The java context awareness framework (jcaf) - a service infrastructure and programming framework for context-aware applications. In: Pervasive Computing. 2005: Munchen. pp. 98–115 (2005)
2. Bardram, J.E., Nørskov, N.: A context-aware patient safety system for the operating room. In: UbiComp '08: Proceedings of the 10th international conference on Ubiquitous computing. pp. 272–281. ACM, New York, NY, USA (2008)
3. Biegel, G., Cahill, V.: A framework for developing mobile, context-aware applications. In: PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04). p. 361. IEEE Computer Society, Washington, DC, USA (2004)
4. Catal, C.: Barriers to the adoption of software product line engineering. SIGSOFT Softw. Eng. Notes 34(6), 1–4 (2009)
5. De La Vergne, H.J., Milanesi, C., Zimmermann, A., Cozza, R., Nguyen, T.H., Gupta, A., Lu, C.: Competitive landscape: Mobile devices, worldwide, 4q09 and 2009. Tech. rep., Gartner (2010)
6. Du, W., Wang, L.: Context-aware application programming for mobile devices. In: C3S2E '08: Proceedings of the 2008 C3S2E conference. pp. 215–227. ACM, New York, NY, USA (2008)
7. Fernandes, P., Werner, C., Murta, L.: Feature modeling for context-aware software product lines. In: SEKE. pp. 758–768 (2008)
8. Gaedke, M., Beigl, M., Gellersen, H.W., Segor, C.: Web content delivery to heterogeneous mobile platforms. In: ER '98: Proceedings of the Workshops on Data Warehousing and Data Mining. pp. 205–217. Springer-Verlag, London, UK (1999)
9. Häkkilä, J., Schmidt, A., Mäntyjärvi, J., Sahami, A., Åkerman, P., Dey, A.K.: Context-aware mobile media and social networks. In: MobileHCI '09: Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services. pp. 1–3. ACM, New York, NY, USA (2009)
10. Nascimento, L.M., Almeida, E.S.d., Meira, S.R.d.L.: A case study in software product lines - the case of the mobile game domain. In: SEAA '08: Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications. pp. 43–50. IEEE Computer Society, Washington, DC, USA (2008)
11. Parra, C., Blanc, X., Duchien, L.: Context awareness for dynamic service-oriented product lines. In: SPLC '09: Proceedings of the 13th International Software Product Line Conference. pp. 131–140. Carnegie Mellon University, Pittsburgh, PA, USA (2009)
12. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
13. Sindico, A., Grassi, V.: Model driven development of context aware software systems. In: COP '09: International Workshop on Context-Oriented Programming. pp. 1–5. ACM, New York, NY, USA (2009)
14. Voelter, M.: Using domain specific languages for product line engineering. In: SPLC '09: Proceedings of the 13th International Software Product Line Conference. pp. 329–329. Carnegie Mellon University, Pittsburgh, PA, USA (2009)

Towards Privacy Policy-Aware Web-Based Systems

Ekaterina Pek
pek@uni-koblenz.de

ADAPT Lab,
Universität Koblenz-Landau,
Koblenz, Germany

1 Problem Description and Motivation

The Web provides different ways to communicate and perform activities that vary from simple to sophisticated: using forums on websites, online shopping, orchestrating Web Services. All of them involve processing data about the user, be it technical (e.g., the version of the user's browser or the IP address) or personal information (name, address, gender, credit card number, etc.). In order to give the user control over their data on the Web, P3P, the Platform for Privacy Preferences, [5] was officially recommended by the World Wide Web Consortium in 2002. P3P is a language and a protocol allowing websites to describe data practices: what data is collected, what for, how long it will be stored and what parts of the data are exposed to other parties. These policies are declarative and non-executable, which leads to the topic of presented work: How to make Web-based systems comply with declared policies? How to make systems policy-aware?

2 Overview of Related Work

2.1 Previous Attempts at P3P Enforcement

Agrawal et al. [6] proposed translating a P3P policy into a set of restrictions in relational database management system (RDBMS) on the level of columns, rows or cells, provided that a user query would contain information about purpose and recipient. This approach requires the support from database producers, since it proposes new language constructs and implementation design for fine-grained access control.

Ashley [7] suggested implementation of an external policy framework that has integration points (Privacy Monitors) between the Privacy Server that contains policies and the application environment. This approach uses Reference Monitors (see below).

IBM developed Tivoli Privacy Manager [1] and related technologies: the Declarative Policy Monitoring [8] and Reference Monitor [12], but the Privacy Manager was withdrawn from marketing in 2009 [2] and corresponding technologies have been retired.

In the work of Hayati and Abadi [10] the authors develop a language-based approach for modeling and verifying aspects of privacy policies. They use the programming language Jif [4], an extension of Java with information-flow types, to show how to prevent leaks of the data from the system and how to implement P3P notions of purposes and retentions. However, this work does not cover all aspects of the P3P language, e.g., base/custom data schemes.

2.2 P3P as an Intermediate Representation

Karjoth et al. [13] proposed the Platform for Enterprise Privacy Practices (E-P3P), which uses P3P to present a coarser-grained privacy policy to the customer, while for internal enforcement a new language is suggested.

In the work of Salim et al. [17], P3P policies are used as an intermediate level between the extended Digital Rights Management model and the user, because P3P preferences are more abstract than a license and it's easier for data owners to specify the purposes for which data is to be collected. In the end, P3P preferences are transformed into MPEG REL (Moving Picture Expert Group Rights Expression Language) licenses that can be enforced by the framework.

2.3 General Solutions to Policy Enforcement

He and Antón [11] propose a framework to bridge the gap between high-level privacy requirements and low-level access control policies by modeling privacy requirements in the role engineering process. The framework provides a basis for enforcing privacy requirements with RBAC (role-based access control) model. The work does not address any high-level privacy requirements language in particular, though, uses P3P elements (e.g., purpose) as an example of standard privacy policy entities.

Mont et al. [15] introduce a notion of "sticky policy" in order to prevent leaking of personal information. The proposed privacy model involves Tracing Authorities as a main point to log and audit the disclosures of confidential data as well as to notify the owner of the data. Such a model requires a request to the Tracing Authority each time when a service wants to transfer the data outside.

Ringelstein and Staab [16] introduce a notion of "sticky logging" in order to collect different kinds of data usage (create, copy, read, update, transfer, delete) in distributed environments. This allows to reconstruct the execution afterwards, which might be useful, if the customer requests the report about data usage. This work does not directly address any kind of compliance of a reconstructed execution with existing privacy policies of a system.

3 Proposed Solution

The relation between the system and the policy can be twofold.

One case is that a P3P policy is created for the existing system by analyzing the behaviour of the system and translating it into the P3P language. This means that a P3P policy can be seen as a by-product of the system and ideally could be generated from the system.

The second case is that a P3P policy exists before the system. This means that a P3P policy can be seen as a specification of the system or as additional constraints at the modelling phase of the system.

While these scenarios are polar, we hope to bridge them in practical experiments (for more details see Section 4). In the end, we aim to develop language support for describing policies as part of the programming effort: that is, a policy-aware programming language supporting idioms for expressing policy-related constraints. We see this language as a simplified, idealized language or calculus, similar to other language design efforts that use Featherweight or Classic Java.There may be the following language constructs for privacy awareness: annotations of the data model with the privacy-related categories; annotations of persistence actions with duration or access information; annotations of service calls so that sharing of data is classified.

4 Research Method

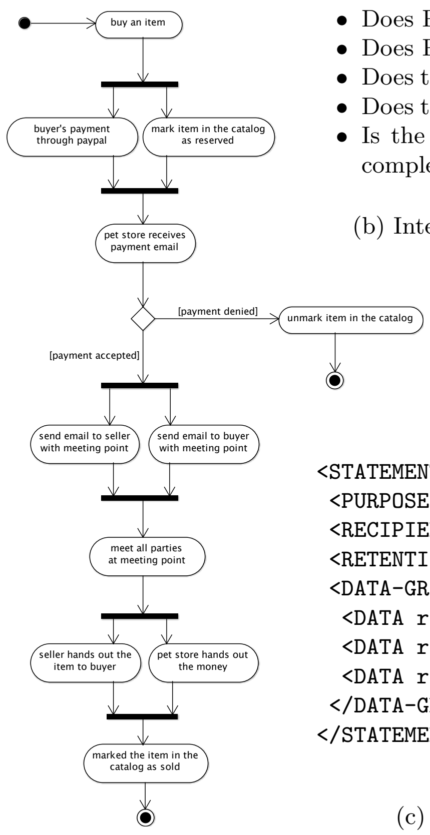
We started our research with an empirical study of P3P policies in the wild [14]. We believe that an empirical study of the language at hand is an important stage of working with the language. While one can start from the specification of the language and try to devise a solution top-down, it can be the case that some combinations of language elements/constructs are seldom or never used, inadequate in practice, or even contradictory [18, 14].

For example, we've found out, that even with low P3P usage – for the seed of 1,450,660 URLs, we were able to download only 4,158 XML files with P3P policies[1]– the coverage of the base data schema proposed in P3P specification is 76%. In other words, the language is used to its full extent and there can be no short-cuts in our effort.

Now that we know the shape of P3P policies, we can start with a realistic and interesting case study to guide the development of a prototype system. This is our plan of attack:

– First, we consider a typical lightweight architecture for a Web-based system: persistence layer, domain-specific logic, presentation layer and, optionally, Web Services. At this point we have decided to dedicate ourselves to Java Pet Store [3], a sample Web application, designed to run on the Java Enterprise Edition 5 platform. While this application has all technical aspects highlighted above, it also suggests non-trivial information flow issues (see Fig. 4). We intend to put some developing efforts into the system in order to make it complete w.r.t. those issues.

[1] Please, note, that this low usage is partially because our approach used one seed of URLs (Google Directory). Cranor et al. [9] found that P3P had been deployed on 10% of the sites returned in the top-20 results of typical searches, and on 21% of the sites returned in the top-20 results of e-commerce searches.


buy an item
buyer's payment through paypal
mark item in the catalog as reserved
pet store receives payment email
[payment denied]
unmark item in the catalog
[payment accepted]
send email to seller with meeting point
send email to buyer with meeting point
meet all parties at meeting point
seller hands out the item to buyer
pet store hands out the money
marked the item in the catalog as sold

(a) The scenario

- Does PayPal get the buyer's contact information?
- Does PayPal get the seller's contact information?
- Does the seller get the buyer's contact information?
- Does the buyer get the seller's contact information?
- Is the contact information deleted upon purchase completion?

(b) Interesting questions about information flow

```
<STATEMENT>
 <PURPOSE><current/></PURPOSE>
 <RECIPIENT><ours/></RECIPIENT>
 <RETENTION><indefinitely/></RETENTION>
 <DATA-GROUP>
  <DATA ref="#user.name"/>
  <DATA ref="#user.home-info.online.email"/>
  <DATA ref="#user.home-info.postal"/>
 </DATA-GROUP>
</STATEMENT>
```

(c) A P3P policy (excerpt) for the seller

Fig. 4: Privacy concerns in a Web purchase scenario

– Then we write a P3P policy for the system, capturing the system's behaviour in the P3P language. Once we have the policy and the system, which is *unaware* of it, we try to map system's parts to policy's parts – to understand the system in policy terms. To this extent we intend to use some sort of a dynamic, run-time analysis so that to see the flow of customer-related data.
– After that we experiment with different ways to achieve policy-awareness, using such mechanisms as annotations, aspects, assertions, etc. From these experiments we expect insights essential to suggest a development methodology for policy-aware systems.

References

1. IBM Tivoli Privacy Manager Solution Design and Best Practices. IBM Press (2003)

2. IBM Tivoli Privacy Manager info page. `http://www-01.ibm.com/software/tivoli/products/privacy-mgr-e-bus/` (Jul 2010)
3. The Java Pet Store 2.0 Reference Application. `http://java.sun.com/developer/releases/petstore/` (Sep 2010)
4. Jif home page. `http://www.cs.cornell.edu/jif/` (Sep 2010)
5. W3C, the platform for privacy preferences 1.1 (P3P1.1) specification. `http://www.w3.org/TR/P3P11/` (Jul 2010)
6. Agrawal, R., Bird, P., Grandison, T., Kiernan, J., Logan, S., Rjaibi, W.: Extending relational database systems to automatically enforce privacy policies. In: ICDE '05: Proceedings of the 21st International Conference on Data Engineering. pp. 1013–1022. IEEE Computer Society (2005)
7. Ashley, P.: Enforcement of a P3P privacy policy. In: Proceedings of the 2nd Australian Information Security Management Conference, Securing the Future. pp. 11–26. School of Computer and Information Science, Edith Cowan University, Western Australia (2004)
8. Bohrer, K., Hada, S., Miller, J., Powers, C., Wu, H.f.: Declarative Privacy Monitoring for Tivoli Privacy Manager. `http://www.alphaworks.ibm.com/tech/dpm` (Jul 2010)
9. Cranor, L.F., Egelman, S., Sheng, S., McDonald, A.M., Chowdhury, A.: P3P deployment on websites. Electronic Commerce Research and Applications 7(3), 274–293 (2008)
10. Hayati, K., Abadi, M.: Language-based enforcement of privacy policies. In: Proceedings of Privacy Enhancing Technologies Workshop (PET). Springer-Verlag (2004)
11. He, Q., Antón, A.I.: A framework for modeling privacy requirements in role engineering. In: Proceedings of the 9th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'03) (2003)
12. Hill, R.K., Fritz, P.: Reference Monitor for Tivoli Privacy Manager. `http://www.alphaworks.ibm.com/tech/refmon` (Jul 2010)
13. Karjoth, G., Schunter, M., Waidner, M.: Platform for enterprise privacy practices: privacy-enabled management of customer data. In: PET'02: Proceedings of the 2nd international conference on Privacy enhancing technologies. pp. 69–84. Springer-Verlag (2003)
14. Lämmel, R., Pek, E.: Vivisection of a non-executable, domain-specific language; Understanding (the usage of) the P3P language. In: Proceedings of ICPC 2010. IEEE (2010)
15. Mont, M.C., Pearson, S., Bramhall, P.: Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services. In: DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications. p. 377. IEEE Computer Society (2003)
16. Ringelstein, C., Staab, S.: DIALOG: Distributed auditing logs. In: ICWS-2009 - 7th IEEE Int. Conference on Web Services. Los Angeles, CA, USA (2009)
17. Salim, F., Sheppard, N.P., Safavi-Naini, R.: Enforcing P3P policies using a digital rights management system. In: PET'07: Proceedings of the 7th international conference on Privacy enhancing technologies. pp. 200–217. Springer-Verlag (2007)
18. Yu, T., Li, N., Antón, A.I.: A formal semantics for P3P. In: SWS '04: Proceedings of the 2004 workshop on Secure web service. pp. 1–8. ACM (2004)

User-centric Programming Language

Wiktor Nowakowski

Warsaw University of Technology, Warsaw, Poland
nowakoww@iem.pw.edu.pl

Abstract. Ambiguously formulated and constantly changing requirements for software systems make it hard to translate them into working code. To overcome these problem, we propose an approach that consolidates the requirements specification level with the design and implementation levels in order to shorten the path from initial requirements to the final code. The end-user of the system will be able to specify requirements in a precise, semantically rich and domain independent User-centric Programming Language (UcPL), which will allow for direct transformation into application logic code.

1 Problem Description and Motivation

Typical software development lifecycles comprise activities that lead from the initial requirements to the final working system. These activities produce various artifacts at different level of abstraction: design models and code. The level of technical complexity makes these artifacts inaccessible to the end-users of the system. However, end-users can discuss the logic (functionality) of the system that abstracts away all the technical details of the software system. This should be done in a language understandable to them. The logic of the problem is usually expressed within the user requirements specifications.

Considering this two main problems arise. The first problem pertains to the process of transition from end-user needs into design and implementation. This includes problems with requirements elicitation and translating the functional requirements into the logic of the system. Attempts to solve these problems are mainly based on the introduction of formal and semi-formal requirements specification languages and automation of transformation between requirements and certain technical artifacts.

The second problem is that the path from requirements to code makes it difficult to maintain appropriate traceability links. This is especially an issue in projects that face high changeability of the requirements. Attempts to solve these problems include rapid application development environments, certain agile methodologies, which try to shorten the path from requirements to code and certain generative approaches where the traces are generated automatically.

Despite shifting the level of abstraction through the introduction of the object-oriented paradigm, still, the existing approaches necessitate a significant level of technical expertise to develop a working system. Thus, there is a rising need to bring the end-user closer to the software developers. The idea here is to let the end-users write some significant portions of software systems. The Gartner Group, in its 2008 study predicts significant growth in end-user software development. In 2010 an average of 34% of companies are expected to conduct more than 20% of application development outside of IT. Unfortunately, there are no approaches to allow the users to construct software through writing the problem logic at the level of requirements. The traditional way of producing software systems is to shift from requirements to code in a generally manual process. More modern approaches try to utilize dedicated tools to automate that process. This necessitates formalizing requirements and constraining the natural language in which they are normally written.

2 Related Work

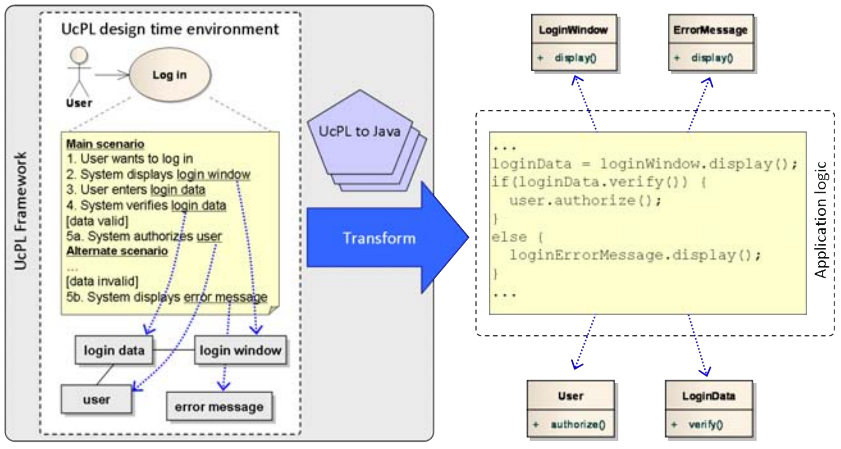
Requirements engineering has been established in the industry as a method for flawless transition from the early system vision over the design, implementation up to the validation and tests. The current practice in this area is more focused on requirements management [1] and requirements interchange [2] than requirements specification. There are also approaches where requirements are specified more formally, like the Four Variable Model [3], [4]. In UML [5] or in its profiles like SysML [6] or MARTE [7], requirements can be handled through creating semi-formally defined visual models. In RSL [8], requirements are specified in constrained natural language with a well defined grammar.

Few of the above mentioned general purpose requirements languages can execute the requirements specification written in it, or can be transformed directly into complete working code. On the other hand, in recent years there has been a drive towards developing executable domain specific languages (DSL) [9]. The idea is that it is more efficient to create a simple language and the corresponding code generators than to apply general purpose languages. The questions are how simple the language has to be and how frequent the language must change to cope with rising demands by its users.

3 Proposed Solution

To go further in resolving mentioned problems, we propose to allow the end-users actively participate in writing software while retaining their role of “requirements specifiers”. They would specify the system logic in the language understandable for them and then automatically transform such specifications directly into code. This would certainly lead to significant gains in productivity.

We aim at delivering a framework consisting of three main elements: the User-centric Programming Language (UcPL), the design time environment and the transformation engine along with appropriate transformation algorithms. Figure 1 shows an overview of the UcPL approach. In UcPL we will substitute procedural or object-oriented programming with, so called, user-centric programming. The user-centric language constructs would just allow to specify the logic of the problem. This means that the user-centric programs could be written and read



Main scenario
1. User wants to log in
2. System displays login window
3. User enters login data
4. System verifies login data
[data valid]
5a. System authorizes user
Alternate scenario
...
[data invalid]
5b. System displays error message

```
...
loginData = loginWindow.display();
if(loginData.verify()) {
  user.authorize();
}
else {
  loginErrorMessage.display();
}
...
```

Fig. 1. Overview of UcPL approach

by end-users with no software development background. Most of the design and technological decisions will be hidden by the language platform.

A scenario of developing software application with UcPL is very straightforward. End-users creates requirements specification in the form of user-centric program. Then they choose the desired transformation algorithm. Precision required in order to apply automatic transformation, necessitates extensive mechanism for validation and correction of the UcPL language constructs. Validation should precede the transformation step. The transformation engine takes complete requirements specification in the form of user-centric program as an input and produces complete code of the application logic (or controller in MVC model). The output code lacks implementation of the application's business model – only class stubs with appropriate methods are generated. The application logic code contains calls to these business entities. The same pertains to the user interface elements like windows, buttons, messages, etc. However, the generated code constitutes the application framework which can be compiled and executed, what significantly shorten the time needed to develop the whole system.

In order to generate code of a full system, appropriate extension of UcPL for specifying business logic and user interface would be needed. This is, though, outside the scope of this work.

A programming language (Java, C++, etc.) and all technological details (e.g. use of a specific user interface technology or specific application framework) of the generated code will depend on the transformation algorithm used. The transformation engine built into the UcPL framework will be able to run any transformation algorithm specified in appropriate transformation language.

In order to ensure end-user comprehension, the UcPL language will be based on use case scenarios written in natural language with simple imperative sentences (e.g. "User enters login data" or "System calculates exchange rate") and control sentences expressing conditions, loops, etc. (e.g. "exchange rate greater than previous exchange rate“).

The language will clearly separate description of the user-system interaction from the description of the domain (see Figure 1). Scenarios will be hyperlinked with appropriate notions defined in a separate vocabulary. Such hyperlinks could be then transformed into operation calls from application logic to business logic and user interface layer.

The syntax of the language will be precisely defined as a meta-model in MOF in order to enable automatic handling of user-centric programs.

4 Research Method

As we mentioned UcPL will need to combine informality with necessary precision, that the end-users would be able to comprehend and write user-centric programs, as well as they are able to understand and write common-prose requirements. The language that addresses most of mentioned issues is the RSL language [10] which was recently developed as a part of the ReDSeeDS project [11]. RSL allows for transformation from requirements specification into draft model of system's architecture. UcPL will be based on the RSL which will be additionally formalized by specifying precise semantics for all the language constructs.

Also an appropriate set of transformation algorithms will be implemented in the model transformation language MOLA [12]. Though there are many transformation languages like QVT [13] or ATL [14], MOLA is preferred for its readability. MOLA is a graphical transformation language where an advanced pattern mechanism is combined with simple traditional control structures. Moreover, MOLA offers comprehensive transformation engine.

An important part of the solution is the design time environment for writing user-centric programs and performing transformations to code. It will be implemented as an extension to the existing ReDSeeDS tool which offers appropriate infrastructure that can be utilized by using plug-in mechanism.

This will enable us to build a system that allows for developing software applications by the end-users by direct transformation from requirements to application logic code. Preliminary studies shows that the proposed approach is possible through skilful extension and combination of the existing technologies.

This goal of retaining end-user comprehensibility of the UcPL as well as useability and effectiveness of the whole framework will be assured and validated through extensive experimental studies. These studies will be mainly carried out by students during software engineering courses. The students will be divided into two subgroups. One group will be developing a system using UcPL approach while the second group will be developing the same or similar system in a traditional way. The results will be compared and analyzed taking into account such factors as time needed to develop the final system and quality of the system measured as the degree of initial user requirements fulfillment.

References

1. Leffingwell, D., Widrig, D.: Managing Software Requirements: A Use Case Approach, Second Edition. Addison-Wesley (2003)
2. ProSTEP: Requirements interchange format (rif). Technical Report PSI 6 Version 1.2, ProSTEP iViP (2009)
3. Parnas, D.: Systematic documentation of requirements. Requirements Engineering, IEEE International Conference on **0** (2001) 0248
4. Heitmeyer, C., Archer, M., Bharadwaj, R., Jeffords, R.: Tools for constructing requirements specifications: The scr toolset at the age of ten. Computer System Science and Engineering Journal **vol. 1** (2005) 19 – 35
5. Object Management Group: Unified Modeling Language: Superstructure, version 2.2, formal/09-02-02. (2009)
6. OMG: Sysml -omg systems modeling language. Technical report (2008)
7. OMG: Marte - uml profile for modeling and analysis of real-time and embedded systems. Technical report (2008)
8. Śmiałek, M., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T.: Introducing a unified requirements specification language. In Madeyski, L., Ochodek, M., Weiss, D., Zendulka, J., eds.: Proc. CEE-SET'2007, Software Engineering in Progress, Nakom (2007) 172–183
9. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, Inc. (2008)
10. Kaindl, H., Śmiałek, M., et al.: Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project (2007) www.redseeds.eu.
11. Śmiałek, M., Kalnins, A., Ambroziewicz, A., Straszak, T., Wolter, K.: Comprehensive system for systematic case-driven software reuse. Lecture Notes in Computer Science **5901** (2010) 697–708 SOFSEM'10.
12. Kalnins, A., Barzdins, J., Celms, E.: Model transformation language MOLA. Lecture Notes in Computer Science **3599** (2004) 14–28
13. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.0, formal/08-04-03. (2008)
14. Jouault, F., Kurtev, I.: Transforming models with the ATL. Lecture Notes in Computer Science **3844** (2005) 128–138

Author Index