Support for Evolution of Software Systems using Embedded Models

Michael Goedicke, Michael Striewe, Moritz Balz
Specification of Software Systems
Institute of Computer Science and Business Information Systems
University of Duisburg-Essen, Campus Essen
Essen, Germany
{michael.goedicke, michael.striewe, moritz.balz}@s3.uni-due.de

Abstract: In this paper we show a new approach to evolution of software systems. We embed high-level specification information into program code patterns, so that such program code is interpretable at different abstraction levels. Since these model information is also acessed at run time for execution, we can avoid the situation that program code and high-level specifications are out of synch. Since the program code is thus a valid notation for the model syntax, we can apply transformations based on model semantics to it. An example will be provided that transforms software based on state machines to process models. This leads to a new perspective of software evolution in which the program code can be considered at higher levels of abstraction.

1 Introduction

In many cases software systems are in use longer than their developers anticipate. This implies especially for larger software systems that essential structural information regarding the overall software architecture and the way operations inside the system actually work is needed beyond the initial development of the system. Usually this kind of information is called documentation and is notoriously out of synch with the actual system after some development time.

Model-driven software development (MDSD) approaches have been advocated to remedy some part of this problem which means that models at a higher level of abstraction containing less detail are used to design a system and generate the actual running version by means of a sequence of transformations. Such transformations take care of mapping the high level concepts to low level details at the programming language level in order to obtain an efficiently running software system. Such transformations can be executed during development time or during run time which means direct execution of the specified models.

Such approaches do not take into account the issues of evolution over a long period of time. Software systems being in use and in constant development over decades is by now the common case while green field developments are very rare these days. Such a long development time requires tight synchronisation between the abstract models on the one

side and the code which is actually executed on the other side. This is due to the fact that the abstract models contain the main part of the essential information regarding system structure and system operation. This abstract specification information contained in the models experiences less change and is thus more stable than the actual program code which has to keep up with changing hardware and software platforms. To make this issue of volatility even worse, hand coded pieces are necessary to overcome some limitations of the automatic transformations. The inevitable evolution of the software system requires then a careful and difficult evolution of the transformations and re-coding of the hand coded pieces and fine tuning related additions.

A common observation is that although all stake-holders in the development process know about these interdependencies between the various development stages, at the end of the day the only piece of information being up to date and available is the program code.

Given these circumstances one would expect tools which help the developers to infer the abstract information of a software system from the actual code. In fact, many approaches [Ant07, Mef06, Mik98, EBM07, SD05, Nec02] pursue this goal. However, it is very difficult to distinguish automatically between program code which represents essential statements regarding business logic and system structure on the whole and code which exists only due to some complicated requirements of some frameworks or libraries which were used or had to be used in the code. So the result of these tools comes in many cases attached with a degree of precision of less than 100 percent. This is not satisfying especially in large software systems since additional analysis has to be done in order to assess the outcome of changes to the source code or models (in case of MDSD).

The need to support long living evolving software systems thus requires a tight integration of program code with its specification (the model) at an abstract level. An additional benefit would be that the specification is contained in the program code in such a way that the specification can be extracted and interpreted during runtime. Such a feature would allow to analyse many important aspects during runtime. This is especially useful in systems which need to run non-stop 24x7. An important aspect, for example, is the set of dependencies a system has on other systems. Modern software systems come with mechanisms like plug-ins which allow to add additional functionality at deployment or even during runtime of the system. Thus the analysis of the interdependencies in a whole landscape of software systems can only be done at a specific point in time regarding an actual set of running systems containing all plug-ins etc.

If the model and the code is tightly intertwined a specific step of evolving a software system entails various checks and (mostly local) transformations which ensure the desired integration of program code and its model. Of course, this also means that not arbitrary program code structures are possible. Such a concept has been described in our earlier work [BSG08, BG09]. Here we show an additional benefit of our approach. This is illustrated by the problem to transform the model which belongs to a specific class of models into an equivalent model of another model class. Of course, this is only possible for compatible model classes e.g. those specifying actions. Here we consider the class of state machine models and process models. This is useful when there is a need to create a new view on the system which has been recognised as an important need in the research field of views and viewpoints in software engineering.

Below we describe our approach and illustrate it with the example transformation from state machine models to process models. In section 2 we describe our general approach which is not limited to models specifying actions. However, the specific approach using state machines and process models is explained as well. In section 3 we show how the actual program code based on a state machine model is transformed into a corresponding program code based on an equivalent process model derived by a small set of transformation rules. This presentation is complemented with a brief discussion of related work in section 4 and discusses pros and cons in the concluding section 5.

2 Embedded Models

Model-driven software development approaches have the objective of relating high-level models to executable systems. Thus, when high-level models are transformed between different notations, a mechanism must exist that derives algorithmic program code from these models. This development step is unidirectional since models cannot be unambiguously extracted from arbitrary program code again. While working with different abstraction levels is desirable for model-driven software development, this unidirectional step constitutes a break of the principle of using model transformations at a higher level of abstraction for software development.

2.1 Concept

We earlier proposed the concept of *embedded models* [BSG09] to overcome these problems. The basic idea of embedded models is to define program code patterns in object-oriented general-purpose programming languages that represent the abstract syntax of high-level models. The program code is thus interpretable at different levels of abstraction, that of the programming language itself and that of the formal model. For this purpose, only static object-oriented structures are used that are not only available at development time, but also at run time. We make especially use of the ability of modern programming languages to decorate object-oriented structures with type-safe meta data [Sch04], thus adding information to program code fragments relating them to high-level models.

Thus it is possible to consider the program code at development time with respect to a formal model. At run time, the same program code is executed by small frameworks that access the program code fragments by means of structural reflection [DM95]. According to the model semantics, acting on the static program code structures creates sequences of actions. Since the program code pattern is part of arbitrary program code, well-defined interfaces to the code outside the pattern must be defined to access the application's state and to invoke business logic. The complete definition of an embedded model thus consists of the following:

- A precise formal model definition.

- A program code pattern that is formed after the abstract syntax of this formal model.
- An appropriate execution semantics.
- Interfaces to arbitrary program code that are interpretable at the level of the model semantics and also provide an appropriate functionality at run time.

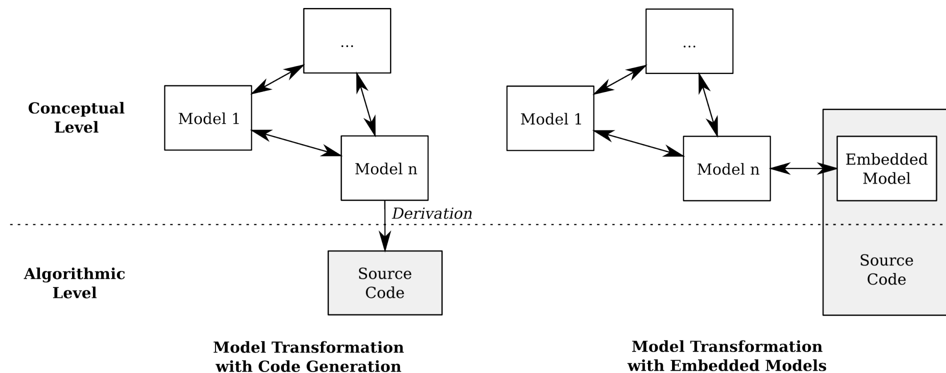

Figure 1: Model transformations with embedded models: The chain of possible bidirectional transformations does not terminate with program code generation. Instead, program code can carry different abstraction levels, thus making it another notation for embedded models and part of the related transformations .

With such embedded model definitions, the program is no longer unidirectionally derived from models. Instead, it is another notation carrying the semantics of the formal model, but includes the possibility to be executed by an appropriate framework while at the same time being integrated in arbitrary applications. Considering model transformations, this means that the program code can be fully integrated: When an embedded model exists, the related code can be source or target of a model transformation. This makes transformations a much more powerful tool for model-driven software development: Models can not only be transformed for communication and design purposes, but also to create *and* re-engineer executable systems. The principle of this approach is sketched in figure 1.

2.2 Example

An example for such a program code pattern is shown in figure 2. The program code fragments represent a part of a state machine model we described in previous publications [BSG08]. The language chosen for this implementation is Java [GJSB05] with its annotations enhancement for meta data inclusion [Sun04].

The class at the top represents a state where the class name equals the name of the state. The method in the state class represents a transition. It is decorated with meta data referring to the target state class and a “contract” class containing guards and updates. An

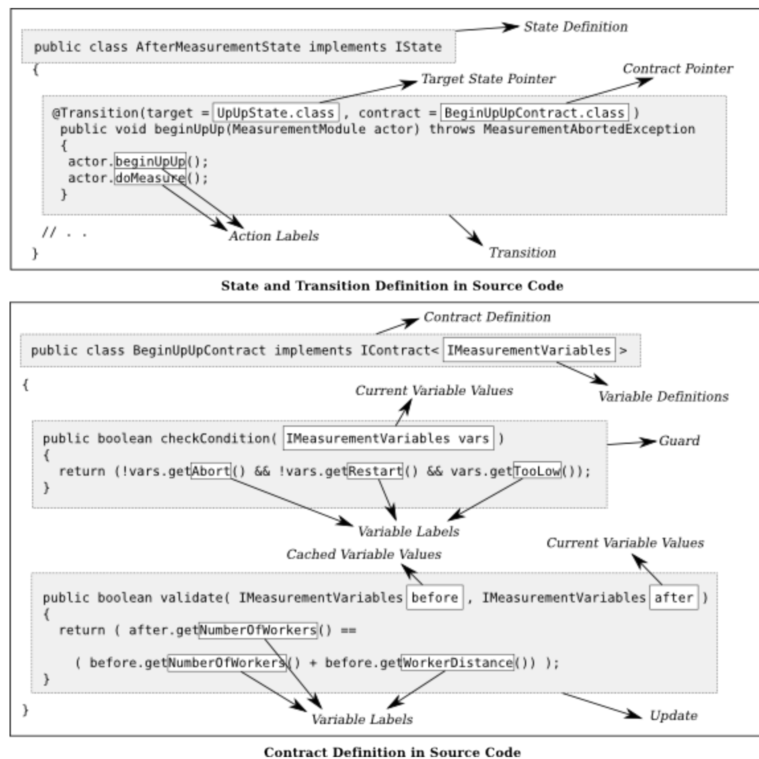

Figure 2: A state definition with an outgoing transitions and its contract. The first method of the contract checks a pre-condition with respect to the current variable values, while the second method checks a post-condition and may thus compare the current values to the previous values.

interface type referred to as “actor” is passed to transition methods. Its methods are interpreted as action labels which can be called when the transition fires.

Guards and updates are implemented as two methods in a “contract” class which is shown at the bottom of the figure. Both evaluate boolean expressions which serve as guards. These guards use the current variable values of the state machine to determine if a transition can fire, the update compares the current values with the values from the point in time before the transition fired to determine the changes to the state space. For this purpose both methods access a “variables” type which is a facade type representing the variables constituting the state space of the state machine. This type contains “get” methods for each variable, which are by this means defined with a label and a data type.

The execution framework can access the classes, methods and annotations by means of reflection. It invokes guards and determines a transition that can fire. After the transition method has been invoked, the update is called and the next state is reached. For this embedded model we already developed a transformation into the input language of the state machine model checker UPPAAL [LPY97] which enabled graphical design and verification of the model.

The complete example state machine consists of 10 states, 27 transitions and 8 variables. It belongs to a load generator application for performance tests and entails a user interface, networking functionality for remote controlled load generation and statistical evaluation of measurement results. These issues are hard to express in models, so that a complete model-driven development of the system was not feasible. However, the core of the load generation process is the strategy used to generate load. It can be modeled as an embedded state machine because it has a well-defined behavior, works with a limited set of variables and initiates the execution of business logic depending on the current state.

3 Program Code Evolution by Model Transformation

In this section we are going to discuss a concrete example on how software evolution can be supported by embedded models. In this example the core behaviour of a software system is designed as a finite state machine. At the level of models this state machine can be transformed into a process model automatically while losing only just a few features of state machines that cannot be expressed in process models. Such an automatic transformation is only possible if the program code adheres to the rules and complies with the model semantics. Thus, embedded models are needed, which define program code structures that are unambiguous.

3.1 Sketch of Concept

Program code can be expressed as its syntax tree generated by a parser. This tree can be enriched by additional semantic information (e.g. edges denoting references) and this way be extended to an attributed graph. Since embedded models rely on static structures, these parts of a model are reflected by the generated graph. Thus model transformation rules that are able to transform a state machine model into a process model or vice versa can be rewritten in order to transform program code with an embedded state machine into program code with an embedded process model or vice versa respectively. From state machines to process models, this transformation consists of several steps:

- All states of the state machine have to be converted to decision nodes in the process model.
- All transitions in the state machine have to be converted to activity nodes properly connected to decision nodes in the process model.
- Each activity node that contains more than one action label has to be split up into a sequence of activity nodes. This step can be performed here or at any later point in time.
- Each decision node having exactly one incoming and one outgoing transition can be discarded, connecting the nodes of the incoming and outgoing transition directly.

- Each decision node without incoming transitions is changed to a start node.
- Each decision node without outgoing transitions is changed to an end node.
- Each descision node with multiple incoming transitions and only one outgoing transition is changed to a merge node.

While this list of steps applies to the model transformation itself, using embedded models requires additional steps because formal aspects of program code (e.g. import statements) have to be taken into account. Note that special concepts like state machines communicating over channels, that have to be expressed by parallel and joining processes, are not considered here in order to keep this example short.

All rules can be implemented as graph transformation rules acting on the graph generated by parsing program code. All changes can be written back as local changes without overriding program code statements that are not part of the model. For example, methods can be moved, copied or renamed and even modified by adding or removing annotations or parameters without touching the body of the method. However, if the transformation requires additional code, new source files can be generated.

3.2 Graph Transformation Rules

The actual set of graph transformation rules used to implement the concept sketched above consists of 21 rules. The implementation has been done using AGG 1.6.4 [AGG] as graph transformation engine with GGX-Toolbox [GGX] for parsing and rewriting Java files. The algorithmic steps listed in the previous section could be implemented by transformation rules straight forward. At first, two rules are concerned with converting states to decision nodes and transitions to activity nodes, implementing the first two steps. After that, a set of six minor rules do some necessary housekeeping to the graph like reordering imports or removing unnecessary annotations. The next two rules remove source code not longer needed and useless decision nodes according to the fourth step of the algorithm. A set of three simple rules is the next to be executed, implementing the last three steps of the algorithm. Afterwards only one major rule is left for splitting up activitiy nodes, which was deferred until here. Another set of seven rules is finally concerned with some adjustments to the code.

One of the most important rules – changing states to process nodes and creating activity nodes – is shown in figure 3 in a simplified manner. Due to the use of embedded models, elements to be moved can easily be identified by their annotations on the left hand side of the rule and thus reassembled on the right hand side. Similarities between state machines and process models allow to reuse larger parts of existing program code, e.g. complete method bodies.

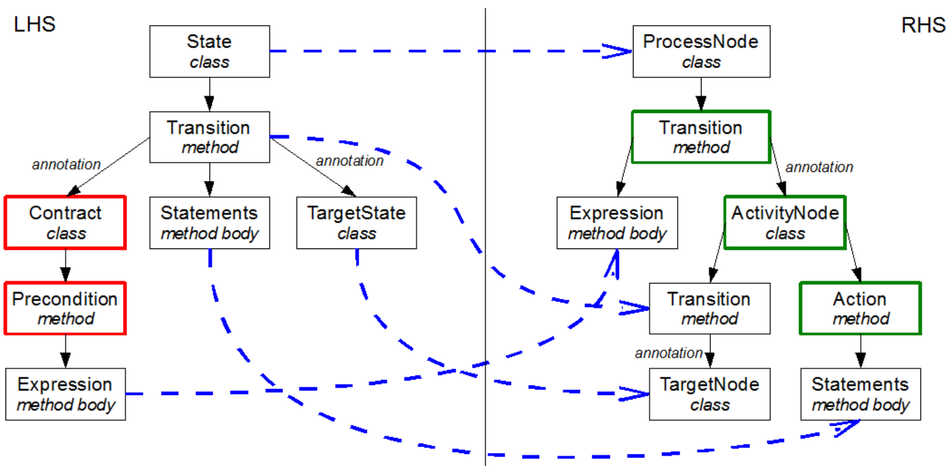

Figure 3: Simplified graph transformation rule for transforming states into process nodes. Nodes deleted from the syntax graph are marked in red while newly created nodes are marked in green. Some of the preserved nodes are renamed during transformation. Note, how contents from the original state node are moved to a newly created activity node, while contents from the original extra contract node are moved to the existing process node.

3.3 Generalization of the Approach

In order to gain reusability these rules can be grouped into five different categories. The main criterion is whether they are concerend with elements of the embedded model before transformation (source model), elements of the embedded model after transformation (target model), arbitrary source code or graph elements that are not relevant for the syntax of program code.

The first category contains rules that act both on elements of the source model as well as of the target model like the rule shown above. These rules can hardly be reused in a general approach since they are specific for a transformation between a specific pair of model types.

The second category contains rules that are responsible for deleting elements from the source model. They can possibly be reused if a transformation starts from this type of source model and has to delete these elements, independent from the target model. Similary, the third category contains rules that create elements from the target model without considering the source model. They can possibly be reused if a transformation has to create the same type of target model.

The fourth category contains rules that work on arbitrary source code elements that are not related to embedded models directly. They might be useful in general, even without working with embedded models. The same applies to the fifth category, containing rules that work on the structure of syntax graphs itself.

Besides rules from the first category, all other rules are reusable at least in some other transformations, thus the approach can easily be generalized. Building a larger library of rules would allow to define different transformations between embedded models, e.g. from process models back to state machine models, by combining rules in the right way.

4 Related Work

Approaches exist that try to relate program code to higher-level specifications. When these specification are formal models, round-trip engineering [SK04] concepts can be applied. Informal specifications can inferred from program code by detecting patterns [PSRN05, Shi07]. Although specifications can be extracted from program code based on design patterns [NWZ01, DLDvL08, WBHS07, NKG+07, Mef06, GSJ00, MCL04, MEB05], all of these approaches still require manual effort or are based on heuristics and are thus error-prone. Approaches to formalize design patterns [SH04, Mik98, MDE97] work at a lower level of abstraction and are not related to abstract specifications.

Approaches which consider the program code itself as a model [VHB+03, HJG08, Vol06], for example using model checking techniques, also work with program code semantics at a low level of abstraction and do not consider abstract specifications.

Integrating models in code has also been studied in [BM06]. Compared to our approach, the references to model elements are generic and not related to specific properties of formal models.

The main issue discussed in this presentation refers to model transformation. This area of research is very active and well explored. A good overview of bidirectional transformations can be found in [CFH+09]. In addition, the field of program transformation is addressing similar goals. In most cases program transformation is performed in form of refactorings which is a different case compared to our approach here: In refactoring approaches a program is (locally) restructured in order to avoid bad code smells. Here we transfer a given program which has a specific structure into an equivalent program with another specific structure.

5 Conclusion

In this paper we described an approach which reports on the benefits of tight integration between program code and its specification in form of a given class of models. The approach transforms a software system based on a state machine model into an equivalent program code again with a model based on the class of process models. This could be then the basis for further developments. The tool chain to support this kind of high level transformation builds on graph transformation and a graph representation of the programs which contains additional model information in form of annotations as available in Java. The actual set of transformation rules is surprisingly small. Of course, there are the usual

preparation and house keeping transformations which provide the necessary build up and cleaning process. The actual transformation is done by just a few transformation rules. Of course, it can be argued that this is due to the small conceptual distance between state machines and process models. But it is also clear from the discussion on views and viewpoints in the literature that each view is a legitimate and useful way in its own right to provide a specification of a system or a system part.

Tools have been implemented to support the actual transformations for multiple purposes, as we described in our previous publications. The objective to recover specifications unambiguously can be used to transform between models as presented in this contribution, but also for visual design and verification. Since embedded models are based on static program code structures that are accessible by means of reflection at run time, they can also be extracted from running systems and transformed into abstract representations, for example for monitoring.

If the approach is applied to model classes being more apart than the two classes used in this paper we envisage still a great part of the transformation process being supported automatically. The parts which have to be supported manually is very small and has clear interfaces to the rest of the system. Thus specific support can be created for these manual steps as well.

Support for additional model classes is currently in development. This will extend the approach to structural models of the systems and widens the support for more runtime related monitoring, refurbishment and evolution of software systems.

References

[AGG] AGG website. `http://tfs.cs.tu-berlin.de/agg/`.

[Ant07] Michal Antkiewicz. Round-trip engineering using framework-specific modeling languages. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 927–928, New York, NY, USA, 2007. ACM.

[BG09] Moritz Balz and Michael Goedicke. Embedding Process Models in Object-Oriented Program Code. In *Proceedings of the First Workshop on Behavioural Modelling in Model-Driven Architecture (BM-MDA)*, 2009.

[BM06] Thomas Büchner and Florian Matthes. Introspective Model-Driven Development. In *Software Architecture, Third European Workshop, EWSA 2006, Nantes, France, September 4-5, 2006*, volume 4344 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2006.

[BSG08] Moritz Balz, Michael Striewe, and Michael Goedicke. Embedding State Machine Models in Object-Oriented Source Code. In *Proceedings of the 3rd Workshop on Models@run.time at MODELS 2008*, pages 6–15, 2008.

[BSG09] Moritz Balz, Michael Striewe, and Michael Goedicke. Embedding Behavioral Models into Object-Oriented Source Code. In *Software Engineering 2009. Fachtagung des GI-Fachbereichs Softwaretechnik, 2.-6.3.2009 in Kaiserslautern*, 2009.

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective—GRACE meeting notes, state of the art, and outlook. In *ICMT2009 - International Conference on Model Transformation, Proceedings*, LNCS. Springer, 2009. To appear.

[DLDvL08] Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde. The QSM Algorithm and its Application to Software Behavior Model Induction. *Applied Artificial Intelligence*, 22(1-2):77–115, 2008.

[DM95] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[EBM07] Ghizlane El Boussaidi and Hafedh Mili. A model-driven framework for representing and applying design patterns. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 97–100, Washington, DC, USA, 2007. IEEE Computer Society.

[GGX] GGX-Toolbox website. `http://www.s3.uni-duisburg-essen.de/research/ggx-toolbox.html`.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java™ Language Specification, The 3rd Edition*. Addison-Wesley Professional, 2005.

[GSJ00] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise Modeling of Design Patterns. In *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, pages 482–496, 2000.

[HJG08] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Model driven code checking. *Automated Software Engineering*, 15(3-4):283–297, 2008.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

[MCL04] Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun. Precise Modeling of Design Patterns in UML. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[MDE97] Theo Dirk Meijler, Serge Demeyer, and Robert Engel. Making Design Patterns Explicit in FACE. *ACM SIGSOFT Software Engineering Notes*, 22(6):94–110, 1997.

[MEB05] Hafedh Mili and Ghizlane El-Boussaidi. Representing and Applying Design Patterns: What Is the Problem? In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2005.

[Mef06] Klaus Meffert. Supporting Design Patterns with Annotations. In *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 437–445, Washington, DC, USA, 2006. IEEE Computer Society.

[Mik98] Tommi Mikkonen. Formalizing Design Patterns. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 115–124, Washington, DC, USA, 1998. IEEE Computer Society.

[Nec02] George C. Necula. Proof-Carrying Code. Design and Implementation. In *Proof and System Reliability*, pages 261–288, 2002.

[NKG+07] Oscar Nierstrasz, Markus Kobel, Tudor Girba, Michaele Lanza, and Horst Bunke. Example-Driven Reconstruction of Software Models. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR) 2007*, pages 275–286, 2007.

[NWZ01] Jörg Niere, Jörg P. Wadsack, and Albert Zündorf. Recovering UML Diagrams from Java Code using Patterns. In *Proceedings of the 2nd Workshop on Soft Computing Applied to Software Engineering, Enschede, The Netherlands (J.H. Jahnke and C. Ryan, eds.)*, 2001.

[PSRN05] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, and Sebastian Naumann. An approach for reverse engineering of design patterns. *Software and Systems Modeling*, 4(1):55–70, February 2005.

[Sch04] Don Schwarz. Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. *ONJava.com*, June 2004. `http://www.onjava.com/pub/a/onjava/2004/06/30/insidebox1.html`.

[SD05] Giovanna Di Marzo Serugendo and Michel Deriaz. Specification-Carrying Code for Self-Managed Systems. In *IFIP/IEEE International Workshop on Self-Managed Systems and Services*, 2005.

[SH04] Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and Rewards: Specifying Design Patterns. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 666–675, Washington, DC, USA, 2004. IEEE Computer Society.

[Shi07] Nija Shi. *Reverse Engineering of Design Patterns from Java Source Code*. PhD thesis, University of California, Davis, 2007.

[SK04] Shane Sendall and Jochen Küster. Taming Model Round-Trip Engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, 2004.

[Sun04] Sun Microsystems, Inc. JSR 175: A Metadata Facility for the Java™Programming Language, 2004. `http://jcp.org/en/jsr/detail?id=175`.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), 2003.

[Vol06] Nic Volanschi. A Portable Compiler-Integrated Approach to Permanent Checking. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society.

[WBHS07] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse Engineering State Machines by Interactive Grammar Inference. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 209–218, Washington, DC, USA, 2007. IEEE Computer Society.