

Embedding Process Models in Object-Oriented Program Code

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

ABSTRACT

Process modeling has usually a strong connection with run time platforms that allow dynamic configuration and adjustment. While this is reasonable for the operation of large applications, it is of no help in cases when program code is to be engineered, documented or verified with respect to process models. We propose a design pattern for process models that allows to embed complete model semantics in object-oriented program code fragments. The program code can thus be validated and executed with respect to process model semantics and design tools.

Categories and Subject Descriptors

D.2.2 [Software]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.4 [Software]: Software/Program Verification—*Validation*; D.2.11 [Software]: Software Architectures—*Domain-specific architectures, Patterns*; F.3.2 [Theory of Computation]: Semantics of Programming Languages—*Process models*

1. INTRODUCTION

Process models can be designed, validated and executed by various tools and frameworks. The focus of these approaches is to allow dynamic configuration of processes with as little relation to source code as possible, sometimes even without involvement of IT departments. This entails the existence of frameworks that read process descriptions, walk through the process and execute business logic attached to several stages of it. The flexibility to design and alter processes descriptively comes at the cost of overhead for this frameworks and also for integration layers to the program code constituting the business logic.

While this makes sense for large distributed applications, it is of no help in cases when program code is to be engineered, documented or verified with respect to process models. Such program code is reasonable in cases when integrated or even

embedded applications are developed and it is undesirable or even impossible to incorporate the overhead associated with process run times. Moreover, little support is given by current modeling technologies to design behavioral aspects of program code in all detail. Modeling languages that aim to represent all behavioral semantics tend to become as complex as general-purpose programming languages [7]. When program code is generated from models, it usually has to be refined to meet the requirements in detail. The code evolves in this case because of enhancements, corrections or tuning activities and can hardly be related back to a model afterwards [4].

So, there is in many cases a gap between abstract model definitions and program code that represents (behavioral) execution logic in detail. We earlier proposed to embed behavioral models in object-oriented code structures to maintain different levels of abstraction in the same program code [2]. This is possible when we define a design pattern that represents the complete semantics of a behavioral model. We will develop such a pattern for process models in this contribution. The program code can thus be considered at different levels of abstraction: A process model can always be extracted from the pattern code and be used to design and validate the model in appropriate tools, for example visual editors. At the same time, the model structures are embedded in and connected with arbitrary other program code that may represent arbitrary (behavioral) semantics.

To explain the approach, this contribution is structured as follows: In section 2 we will describe the process model we want to represent by a design pattern; the pattern itself and the related program code fragments are explained in section 3. Based on this we show how process models can be designed with a modeling tool in the Eclipse IDE and executed with a lean framework afterwards in section 4. We evaluate the approach by means of a sample application in section 5, give an overview of related work in section 6 and afterwards conclude in section 7.

2. MODEL DEFINITION

The objective of the pattern to develop is to engineer program code for applications whose behavior can be expressed with process semantics. We must for this purpose define a specific process model that describes the features of interest. We will stick to the process meta model defined by the Eclipse IDE's [19] Java Workflow Tooling [18] project which

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

BM-MDA'09, June 23, 2009, Enschede, the Netherlands

Copyright © 2009 ACM ISBN 978-1-60558-503-1/09/06 ... \$10.00

aims to bridge differences between existing process model notations to build a uniform editing and monitoring tool set.

The definition of this pattern as such is not innovative. But, we need a clear definition to create the design pattern afterwards that is embedded unambiguously in object-oriented program code. Additionally, the model must – since it will be expressed in program code fragments – define interfaces to arbitrary other program code that belongs to the same applications under development.

The model itself consists of two types of elements, nodes and transitions, whose properties control the flow of the process.

2.1 Activities

Activity nodes contain an activity, which means execution of arbitrary business logic of the application that contains the process model. Execution control is here passed from the scope of the process model to the application. For the definition of the design pattern this means that an interface to this arbitrary program code must be defined that will be used here.

Each activity is – as borrowed from JWT – executed by an application which is defined by a class and method name. The activity can have input and output data.

2.2 Transitions and Guards

Transitions are directed edges that connect nodes and thus guide the process flow. They are most important when they emanate from *decision nodes*. In this case they mark the begin of branches that are based on a decision depending on variables. These variables are named and represent the state space of the application as far as it is of interest to the process model. At this point we have a second interface to the business logic of the application that provides the according variable values.

The variables are evaluated by guards that are attached to the transitions emanating from the decision node. The guards consist of an expression that considers some of the variables and produces a Boolean value. During execution, the guard of each transition is evaluated; the first transition whose guard evaluates to *true* is selected.

The expressions in guards consist of

- a left side which is always a variable identifier,
- an operator out of $\{==, !=, <, <=, >, >=\}$,
- a right side which is a literal value or another variable identifier.

These simple expressions can be aggregated to complex expressions with union ($\|\|$) and intersection ($\&\&$) operators.

2.3 Simple Nodes

Apart from activity and decision nodes, further nodes exist that have no properties except a name. However, they

control process flow with the transitions attached to them. Beginning and termination of a process are marked by *start nodes* and *end nodes*. For concurrent execution of processes, *fork nodes* can define the beginning of a parallel execution. Concurrent flows are united in *join nodes*. *Merge nodes* bring different possible paths after decisions together.

3. PATTERN DEFINITION

Now that we have outlined the process model features that we want to use to engineer program code, we can define the design pattern that represents these model semantics. This approach builds upon the concept of so-called *internal DSLs* [8], i.e. domain-specific languages that are embedded into other languages (host languages). Semantics of DSLs are by this means available inside a general-purpose programming language. Furthermore, *attribute-enabled programming* [15] uses the capability of modern programming language versions to incorporate type-safe, compiled meta data to annotate source code fragments. These annotations can be used to make program code semantically interpretable even at run time. We combine these existing concepts to embed the model definitions in code fragments. The rules for this are explained below. For each section an illustration is given that shows the assignment of program code fragments to process model elements. They are taken from a larger example that will be explained in section 5.

The mapping has so far been implemented for the Java programming language, version 5 [9], which includes all necessary language elements, especially annotations for type-safe meta data [17].

3.1 Nodes and Transitions

The foundation of process models is the graph structure of process nodes and transitions between them. In the object-oriented program code fragments for the pattern, each process node is represented by a class definition which implements the given interface `IProcessNode` (“node class” in the following). This interface defines no methods, but allows to distinguish between node classes and arbitrary other classes type-safely. The class name corresponds to the process node name. Process nodes (except activity nodes, see below) are decorated with meta data denoting their type (one of the enumeration constants *START*, *END*, *FORK*, *JOIN*, *DECISION*, or *MERGE*).

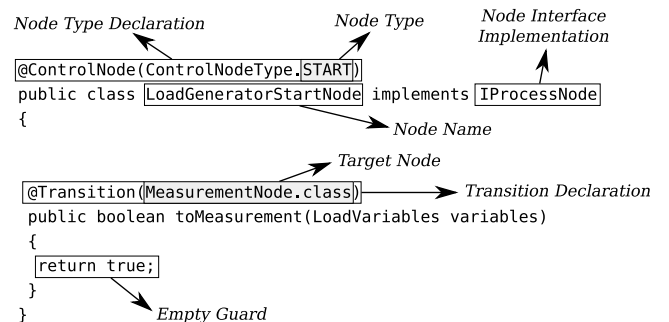


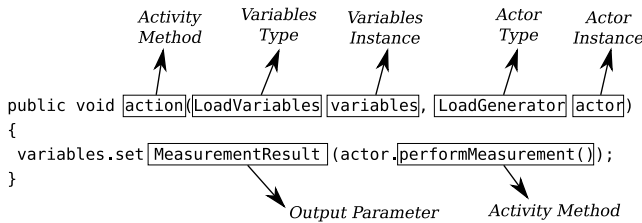
Figure 1: A process node represented in the design pattern. The single program code fragments refer to the node’s name and type, a transition declaration and an empty guard.

Each transition is represented by a method (“transition method” in the following) inside the node class it emanates from. A transition method is decorated with meta data containing a pointer to the class definition representing the transition’s target node. With these simple definitions, the basic process graph is embedded in static object-oriented structures. The program code that is required for this is shown in figure 1.

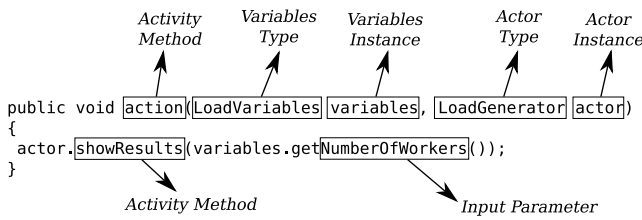
3.2 Activities

The part of the design pattern that represents activities is slightly more complex since it considers method contents also. As defined in the model above, we will in this context need variables that are available in the model semantics and represent input and output parameters of activities. The variables are represented by an interface definition. This interface provides methods for retrieving and setting variable values (“getter” and “setter” methods in the following) which are identified by the names of these methods.

An activity node is also represented in the program code by a class definition. However, activity nodes implement an interface `IActivityNode` which itself extends `IProcessNode`. This interface defines a method `void action(Object actor, Object variables)`, i.e. a method whose body is an activity in program code. Passed to this method are two parameters: First, the *actor* which is a type that encapsulates business logic of the surrounding application; second, an implementation of the variables interface.



(a) Activity with Output Parameter



(b) Activity with Input Parameter

Figure 2: Two exemplary activity methods. The first returns a value which is used as an output parameter of this activity node by calling a setter method in the variables facade. The second has an input parameter which is represented by a getter method.

The method body content follows these rules, as can be seen in figure 2:

- The actual activity is performed with a call to one of the methods of the actor instance.

- The actor method can optionally take parameters. These parameters must each be an expression containing a call to a getter method of the variables implementation. They represent input parameters to this activity as defined in the process model.
- The actor method can optionally return a value which is passed to a setter method of the variables implementation. This represents an output parameter of this activity. Since the return value is used for this purpose, we limit the number of possible output parameters in our pattern to 1.

3.3 Guards

As defined above, transitions between nodes are represented by methods. When the methods emanate from decision nodes, they are required to have guards, i.e. evaluations of variable values to allow for decisions. For our pattern we define that these are represented by the method body of transitions methods. The body can therefore contain expressions as defined in section 2.2, but with calls to getter methods of the variables type. For example, the guard expression `Value1 > Value2` would be represented in the source code as `variables.getValue1() > variables.getValue2()`. The according rules apply for interleaving of expressions. An example is shown in figure 3.

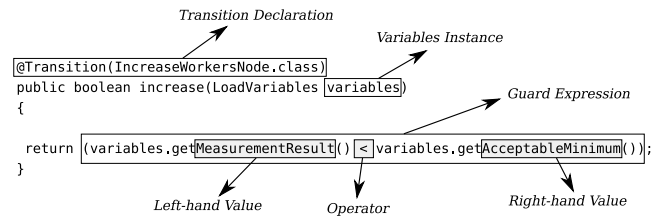


Figure 3: A guard represented by an expression in the body of a transition method.

However, a significant part of transitions methods are defined outside of decision nodes and must have a well-defined content too. We define for this purpose that the default content of such a method is `return true`. This is a valid program code statement and will mean that no guard is defined for this transition.

4. USAGE AND TOOLS

While the use of the pattern itself may already structure the development process by facilitating the definition of a mental model by the programmer, tool support for the interpretation of the well-defined pattern fragments is desirable. We have so far developed two tools: An execution framework that interprets the fragments at run time, and a connector to a visual modeling tool that allows to design the program code with process semantics at development time.

4.1 Execution

The execution framework is based on Java reflection mechanisms. To start execution of a process from arbitrary program code, the start node, an instance of the variables type and the actor instance are passed to the framework. It instantiates all node types that are reachable by transition annotations. Afterwards the framework walks through the pro-

cess by following transitions. Guards of transitions are evaluated in decision nodes by invoking the transition method and passing the variables instance to it. In activity nodes the framework calls the activity method and passes the variable and actor instance to it. In fork nodes, new threads are started that walk through the additional defined paths. The process is continued until the current state is the end state or the framework runs into a deadlock in a decision node when no guard evaluates to *true*.

4.2 Modeling

The full benefit of working at different layers of abstraction can only be realized with modeling tools that allow for visual representation of process semantics. We chose to develop a connector to the JWT process modeling framework mentioned above. The connector is realized inside the Eclipse's Java Development Tools [20] environment. When the user selects a package with Java classes inside Eclipse, the connector allows to transform the source code inside this package into a process model. For this purpose it searches for classes that are of interest to the pattern, i.e. classes that implement the `IProcessNode` or `IActivityNode` interface. It builds a graph from them afterwards and considers the following content:

- Node classes and transition methods are directly transformed to nodes and edges in the model by using their respective names.
- Data types related to Java data types are defined for all variables.
- All methods defined in the actor class are defined as applications in the process model.
- Activity methods are analysed according to the rules defined in section 3.2.
- If a transition method is inside a decision node and contains statements other than `return true`, it is a guard and as such validated for conformity with the rules defined in section 3.3 and extracted in a tree structure of single statements linked with the valid set operators.

The model information is serialized into an XMI [14] file which is interpreted by the JWT editor. The connector interprets the pattern complete enough to extract the whole model. An example is shown in figure 4. The load generator application that can be seen there is extracted from the code of an example that will be introduced in section 5.

Currently, our connector does not create or change Java code from models designed in the editor. However, since all relevant information about the model is available in the editor, this is as straight-forward as extracting the model out of the source code.

5. EXAMPLE

As can be seen in the graphical representation, these simple rules for our pattern are sufficient to embed complete process semantics in program code. We will now illustrate

the code structures in detail by means of an example application written in the Java programming language. The example is a process model in a load generator application for performance tests. We assume that the load generation is a complex operation that may include networking issues, for example remote controlling of worker threads on different physical machines. The details of load generation are therefore not in the focus of the process model, but only the aspect of the behavior that controls the measurement, which can be reduced to a few well-defined variables.

5.1 Description

The user can give a range of acceptable response times defined by two variables *AcceptableMinimum* and *AcceptableMaximum* which denote the limits. Load is generated by a number of worker threads which is available in the variable *NumberOfWorkers*. The number of workers is adjusted depending on the measurement results across different single measurements. The latest measurement result, represented by the average response time of the system under test, is stored in the variable *MeasurementResult*. The related interface *LoadVariables* in Java is shown in listing 1. It contains getter methods for the range limits and the number of workers and pair of get and set method for the measurement result.

```
public interface LoadVariables
{
    double getAcceptableMinimum();
    double getAcceptableMaximum();

    double getMeasurementResult();
    void setMeasurementResult(double result);

    int getNumberOfWorkers();
}
public class LoadGenerator
{
    public void increaseWorkers()
    { // ... }
    public void decreaseWorkers()
    { // ... }
    public float performMeasurement()
    { // ... }
    public void showResults(int numberOfWorkers)
    { // ... }
}
```

Listing 1: The variables interface for the load generator example with the according get and set methods

This process model controls the flow of the application, the activities are delegates to arbitrary program code. For this purpose, the actor type *LoadGenerator* is defined as an interface to the business logic as seen in listing 1. The most important method here is *performMeasurement* (see listing 2) that generates load with the given number of worker threads (initially 1). It returns a number value with the measured average response time. Afterwards, the number of workers must be increased (method *increaseWorkers*) or decreased (method *decreaseWorkers*) when the load was below or above the given range. After the measurement has finished, the result, i.e. the acceptable number of workers, is shown to the user. The method *showResults* takes for this purpose one parameter with the according number of workers.

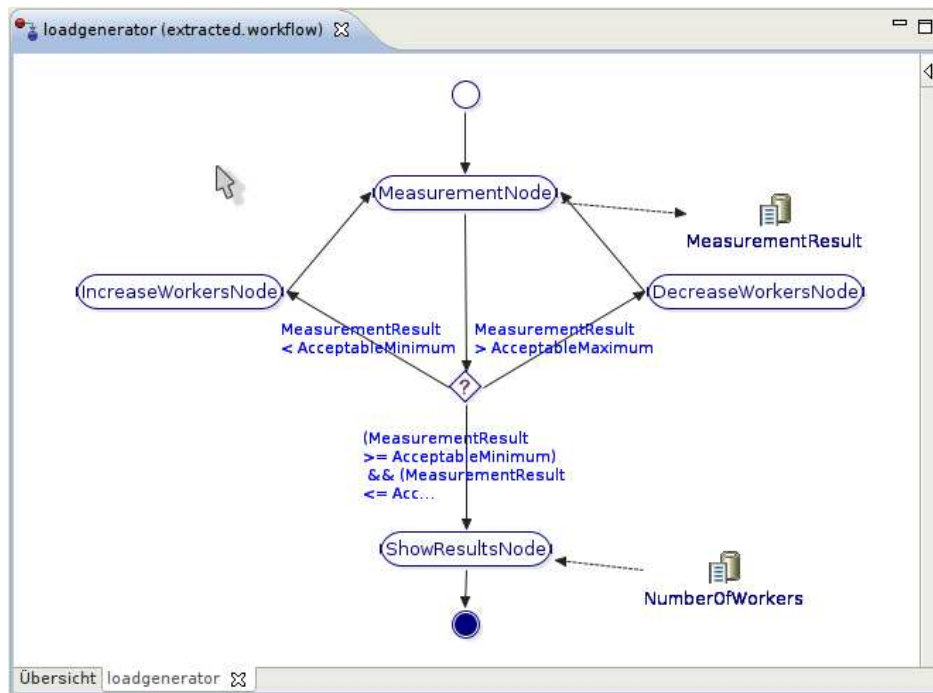


Figure 4: A process model extracted from a pattern in the JWT editor. The example application shown here is discussed in detail in section 5.

These requirements are sufficient to define the activity nodes. To complete the model, we must add the start and end node as well as a decision node that evaluates the results after each measurement. It makes the decision to increase or decrease the number of worker threads or alternatively finish the measurement based on the variable values and defines for this purpose guards as shown in listing 2.

5.2 Evaluation

The example shows that, based on the pattern definition, a process model can be embedded in the program code completely. The editing tool presented above allows to model the behavioral model structures visually. The relation between code and model is unambiguous, so that a transformation between both representations is possible. At the same time, the source code is the only necessary representation from which other representations for different degrees of abstraction can be extracted on demand.

In the use case of the load generator, working at different degrees of abstraction can be important. When we imagine that load generation is distributed over the network, working with time constraints for proper measurements and also using plug-ins that can generate different types of load, it would be hard to find a modeling tool that allows to express all this in behavioral models. In contrary, a framework for business processes would be a large overhead and not an appropriate solution for the need to model this simple process.

The embedded model pattern is therefore the adequate compromise: The process is embedded in arbitrary business logic with arbitrary behavior. The program semantics related to the process model are represented distinctly, the arbitrary

program state is reduced to well-defined variables for this purpose. The definition of the pattern code structures is certainly a little overhead, but allows to visualize the model on demand and therefore comprehend the concept behind code structures easily. For the definition in the context of the example application, the embedded model is therefore capable of providing different levels of abstraction with respect to process semantics.

6. RELATED WORK

Many approaches exist that try to engineer program code at different layers of abstraction.

Model Round-Trip Engineering concepts [16] allow to abstract from source code by synchronizing it to abstract models. However, they require manual effort [10] and are therefore not unambiguous enough to create these abstractions ad-hoc. In this context, the attribute-oriented programming approach has already been explored to map UML models to code structures [22], similar to Framework Specific Modeling Languages [1]. In contrast to embedded models, these approaches rely on the existence of different representations for different abstraction levels, and do not avoid round trip engineering.

Specification languages like the Java Modeling Language (JML) [3, 11] or the introspection capabilities of Smalltalk [6] enable the definition of semantic information and modeling constraints inside object-oriented source code and provide an extensive syntax for this purpose. These meta data are though not related to formal models like processes and are not detailed enough to extract model definitions completely into abstract representations.

```

public class MeasurementNode implements IActivityNode<
    LoadVariables, LoadGenerator>
{
    public void action(LoadVariables variables, LoadGenerator
        actor)
    {
        variables.setMeasurementResult(actor.performMeasurement());
    }

    @Transition(EvaluateMeasurementNode.class)
    public boolean toSetResults(LoadVariables variables)
    {
        return true;
    }
}

@ControlNode(ControlNodeType.DECISION)
public class EvaluateMeasurementNode implements IProcessNode
{
    @Transition(IncreaseWorkersNode.class)
    public boolean increase(LoadVariables variables)
    {
        return (variables.getMeasurementResult() < variables.
            getAcceptableMinimum());
    }

    @Transition(DecreaseWorkersNode.class)
    public boolean decrease(LoadVariables variables)
    {
        return (variables.getMeasurementResult() > variables.
            getAcceptableMaximum());
    }

    @Transition(ShowResultsNode.class)
    public boolean showResults(LoadVariables variables)
    {
        return ((variables.getMeasurementResult() >= variables.
            getAcceptableMinimum()) && (variables.
            getMeasurementResult() <= variables.
            getAcceptableMaximum()));
    }
}

```

Listing 2: The nodes that initiate and evaluate a measurement. The type is specified with annotations and implemented interfaces; The action method contains the activity with an output parameter, the other methods are transitions with guards

Contrary to the model checking approach of Java PathFinder [21] we do not consider the semantics of a complete program as such. The same applies to the concept of Introspective Model-Driven Development [5] that aims to identify unknown model structures in the source code. Instead, we relate only selected parts of it that are well-defined beforehand to existing formal models.

Similar to run time systems that execute process models are Executable Models, for example “executable UML” [12], that aim to avoid working with source code completely by direct execution of model specifications. This relies on the assumption that entire applications can be expressed as models, which is – especially for behavioral modeling – not realistic from our point of view, as mentioned in the introduction.

7. CONCLUSION

We presented an alternative approach to behavioral modeling. To allow working at different levels of abstraction when program code is engineered, we defined a pattern that represents process model semantics in object-oriented source

code fragments. Arbitrary source code can thus be enriched with semantical information where applicable. Although the pattern definition is still elementary so far, we have already proven that the approach is capable to fulfill the requirements: The creation of the connector to the JWT process modeling tool shows that an unambiguous interpretation of the pattern fragments is possible. We can also execute the process model at run time while it is still part of the program code of arbitrary applications.

Future work will focus on the aspect of editing the source code: Instead of only extracting the model from the code, we will create a connector for the opposite direction that creates source code from a model or merges changes back to the source code. Our vision is a modeling tool that allows for continuous working at different abstraction levels by offering different views on the same program code to engineer.

8. REFERENCES

- [1] M. Antkiewicz and K. Czarnecki. Framework-Specific Modeling Languages with Round-Trip Engineering. In Nierstrasz et al. [13], pages 692–706.
- [2] M. Balz, M. Striewe, and M. 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.
- [3] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach*. Springer-Verlag New York, Inc., 2007.
- [4] A. W. Brown, S. Iyengar, and S. Johnston. A Rational approach to model-driven development. *IBM Systems Journal*, 45(3):463–480, 2006.
- [5] T. Büchner and F. 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.
- [6] S. Ducasse and T. Girba. Using Smalltalk as a Reflective Executable Meta-language. In Nierstrasz et al. [13], pages 604–618.
- [7] M. Fowler. PlatformIndependentMalapropism, 2003. <http://martinfowler.com/bliki/PlatformIndependentMalapropism.html>.
- [8] M. Fowler. InternalDslStyle, 2006. <http://www.martinfowler.com/bliki/InternalDslStyle.html>.
- [9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java™ Language Specification, The 3rd Edition*. Addison-Wesley Professional, 2005.
- [10] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.
- [11] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
- [12] S. J. Mellor and M. J. Balcer. *Executable UML*. Addison-Wesley, 2002.
- [13] O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors. *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS*

2006, Genova, Italy, October 1-6, 2006, Proceedings, volume 4199 of *Lecture Notes in Computer Science*. Springer, 2006.

- [14] OMG. MOF 2.0 / XML Metadata Interchange (XMI), v2.1.1 specification, 2007.
- [15] D. 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>.
- [16] S. Sendall and J. Küster. Taming Model Round-Trip Engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, 2004.
- [17] Sun Microsystems, Inc. JSR 175: A Metadata Facility for the Java™ Programming Language, 2004. <http://jcp.org/en/jsr/detail?id=175>.
- [18] The Eclipse Foundation. JWT website. <http://www.eclipse.org/jwt/>.
- [19] The Eclipse Foundation. ECLIPSE website. <http://www.eclipse.org/>.
- [20] The Eclipse Foundation. Eclipse Java Development Tools, 2008. <http://www.eclipse.org/jdt/>.
- [21] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), 2003.
- [22] H. Wada and J. Suzuki. Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In L. C. Briand and C. Williams, editors, *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, volume 3713 of *Lecture Notes in Computer Science*, pages 584–600. Springer, 2005.