Embedding Behavioral Models into Object-Oriented Source Code

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

Abstract: In this paper we address the inevitable problem of aligning a software model with its respective code in such a way that the synchronization between both views of the system can be effectively maintained. In traditional approaches to model-driven software development (MDSD) it is at least necessary to annotate the source code in order to mark its origin for round-trip engineering and to prevent it from being overridden when regenerating code from other models. Instead of just maintaining references to models, we suggest to embed full model semantics into source code. To realize this approach we modify the earlier concept of ViewPoints, so that the necessary configuration of ViewPoints happens inside the source code by using selected constructs of object-oriented programming languages. Our contribution presents the approach to maintain models and source code simultaneously by means of behavioral models.

1 Introduction

Many existing approaches to model-driven software development (MDSD) suffer the problem to relate evolution of source code back to the model [BIJ06]. Several problems prevent automated solutions and thus hinder parallel development of models and source code, especially in large scale industrial software development [HT06, BLW05]. In large applications we can expect different parts of the system to be designed according to different models, such as a process model that works in parallel with a data model. Additionally, parts of source code may be written manually to satisfy the needs of particular frameworks in use or non-functional requirements like performance. To avoid confusion, it is at least necessary to annotate the source code in order to mark its origin for round-trip engineering and to prevent it from being overridden when regenerating code from other models. The result is a huge effort regarding the synchronization. Models have to be kept in sync with each other, each model has to be kept in sync with the code and parts of the source code that are represented by more than one model have to be handled very carefully [VG05].

In cases where modeling technologies are used for system design and not for run time configuration, we want to avoid the overhead to synchronize code and models. Instead of just annotating code with references to models, we suggest to embed full model semantics into source code. If the model is seen rather as an abstract idea than as a visual repre-

sentation, source code structures can be just a different notation for models. The visual representation can be extracted from the code on demand, but is only a partial view on the whole system. This seems to be a promising approach since it is easier to extract a specialized view from the complete system instead of constructing and maintaining the complete system out of specialized views.

To realize this we use the concept of ViewPoints [FKG90, FKN+92] in a slightly modified version, so that the necessary configuration of ViewPoints happens inside the source code by using selected constructs of object-oriented programming languages. Several benefits can be expected from this approach:

First, the existence of an up to date version of the source code is a necessary condition for any useful software system. Once the system is deployed the consistency with any abstract model does not matter until some maintenance or evolution is due. Thus, if the model is maintained in and extracted from the source code for any development step source code and abstract model are in sync automatically. Moreover, different external model representations may be changed concurrently and have to be synchronized manually, but they are automatically in sync when changes are applied to the source code directly.

Second, an application can be aware of the models, since their complete semantic is available at run time via reflection mechanisms. This enables monitoring and model manipulation at run time if desired and appropriate.

Third, our approach integrates smoothly into any phase of the software development process, right after the first lines of source code have been written. In early design phases, single software components can be generated separately from their models. In later phases, these components are integrated manually into a larger system and models can be extracted to validate their respective correct integration. During testing and debugging, models can be extracted to validate whether bug fixes do not affect the intended behaviour.

This contribution is organised as follows: Section 3 introduces the combination of MDSD and ViewPoints at a general level in 3.1 and substantiates the idea for behavioral models in 3.2. Section 4 presents the realization using an actual set of technologies, followed by a case study for state machine models by means of a sample system. This case study is the basis for our assessment in 5. The paper concludes with a discussion of interesting open questions in future research and development in 6.

2 Related Approaches

Various techniques exist to embed semantic information and modeling constraints into object-oriented source code [BHS07]. Examples are the JAVA MODELING LANGUAGE (JML) [LBR99] offering an extensive syntax for specification annotations or the approach to use Smalltalk with its introspection capabilities as a meta language [DG06]. In contrast to such universal approaches we do not aim to present a notation for the specification of all possible system models. Contrary to tools like JAVA PATHFINDER [VHB+03] or abstract universal approaches like Introspective Model-Driven Development [BM06], our approach does not consider a whole application as model, but only designated parts of it.

This limited domain of behavioral models can thus be examined more thoroughly and with respect to a formally well founded background.

The approach to use declarative code structures to maintain relations to models has already been explored, for example in the context of UML models [WS05]. This relies on the existence of external models that have to be reconciled with the source code or vice versa. The same applies to Framework Specific Modeling Languages [AC06]. In contrast, our approach facilitates the principle of having only one representation for model and source code and thus avoids round trip engineering.

Executable models, for example "executable UML" [MB02], automate the transformation of models into executable systems or directly execute model specifications. While this approach does allow a clean and model-centric view of systems, it assumes that entire applications can be expressed as models. Especially in heterogeneous environments this assumption will fail often since different frameworks are in use or a high priority of non-functional requirements, e.g. performance demand the application of certain patterns which causes a deviation from the structure an abstract model would imply. In such cases an automated integration into existing systems is not easy to achieve. In our approach the model structures in the source code can co-exist with non-model parts of an application, thus enabling a seamless integration into larger systems. In addition, our approach executes only compiled byte code and interprets no external representations, which avoids possible performance problems.

3 ViewPoints, MDSD and Embedded Model Semantics

Here we discuss briefly how MDSD and the concept of ViewPoints can be brought together and how this is related to embedded model semantics. At first we discuss this at a general level. Then we substantiate the idea by talking about behavioral models and imperative programs using Java as sample programming language.

3.1 The general Problem

Each step in MDSD is characterized by a certain view on the system under construction or evolution. A computational independent model (CIM) neglects any algorithmic aspect and puts the focus on domain knowledge and business processes, for example. In terms of ViewPoints, this would be part of the view of an business expert. The same can be said for platform independent models (PIM), that may be part of the view of a system architect, and platform specific models (PSM), that may be part of the view of a database expert. While MDSD implies a sequential order of these models, ViewPoints allow to use these models in parallel. Moreover, ViewPoints may be defined in ways that do not clearly correspond to the levels of CIM, PIM or PSM.

The obvious goal of the software development process is to create a system that covers the semantics of all models. In terms of ViewPoints this is realized by finding a consistent

configuration of all views. In terms of MDSD it is realized by incremental annotation and generation of models according to their sequential order, so that the semantics of a CIM is embedded in a PIM and the semantics of a PIM is embedded in a PSM and finally the generated code realizes all the semantics expressed in a PSM.

Comparing both ideas, we can observe two problems. On the one hand, a strong benefit of ViewPoints is the concept of configurations and establishing relationships between the involved ViewPoints. In general, they are easy to manage and to evolve and they ensure that any necessary update is made, but nothing more. If one actor in the development process makes a change in his/her view, the system must be updated by changing the configuration, but never the other views. This is not valid for MDSD. If a CIM changes, all subsequent models have to be updated or regenerated, but if a PSM changes, there is no check whether it still conforms to the original CIM. On the other hand, this order of models and derivation of subsequent models by annotating previous ones is a strong benefit of MDSD. In the concept of ViewPoints, parallel changes in all models are allowed and relationships have to be established manually but systematically. This can lead to ambiguous cases in which more than one relation is possible.

The solution suggested in this paper is to use the source code as the superior and unambiguous representation of semantics. In terms of ViewPoints, source code configures views by identifying and annotating their objects and sharing them between views if necessary. In terms of MDSD, the semantics of each model are directly embedded into source code instead of being deferred to another model. Hence any change in one view or model affects the source code directly and thus updates all other models and views if and only if it is necessary.

The key of ViewPoints to the case proposed here is that establishing explicit relations between the code view and the model(s) view defines the necessary semantic elements in representing development artefacts at different levels of detail and abstraction. Such a consistent multi-view representation scheme enables the application of methods and tools at the corresponding abstraction level and allows the effective translation of related results to the corresponding other levels.

We now discuss how such different levels of abstraction and detail can look for imperative programs.

3.2 Behavioral Models and Programs

In order to embed behavioral models in object-oriented structures we will extract the relevant features from behavioral model specifications that we want to use explicitly. For our purpose we define behavioral models to consist of (at least some of) the following parts:

- States are specific moments in the behavior of a system.
- Activities represent arbitrary functionality being executed by this system. When activities are executed, control flow is handed over to application logic and returned back to the model when the activity has finished.

- Transitions are possible paths between states.
- Guards represent conditions that indicate if a transition can fire.
- Updates specify how activities are expected to change the system state.
- Compositions may encapsulate other elements to simplify the depiction of complex systems and thus build hierarchies of such behavioral models.

This abstraction is applicable to various behavioral modeling techniques. For finite state machines [PGS01] the elements are composed using the following rules: (1) States do not modify a system, but represent a certain condition of that system; (2) guards decide which transition emanating from the current state will fire; (3) activities are executed during transitions; (4) updates describe the changes in the system state after transition activities have been executed. The application of the aforementioned abstraction to process models [Coa95] is slightly different: (1) Activities are executed during states; (2) after execution, guards decide which transition to a next state will fire; (3) updates are usually not part of process models, but may be included in states to validate activity execution by describing the expected changes in system state. Similar rules may be found for e.g. UML activity diagrams [OMG04].

Now we need to define representations in the source code that allow us to specify the model structures in systems at run time. The related programming language constructs must be explicitly defined in the source code and the corresponding information must be accessible at run time, i.e. after compilation, e.g. by means of structural reflection [DM95]. In the context of modern object-oriented languages, for example Java [GJSB05], basic constructs like classes and interfaces are static definitions of application structures and data. The code of actual business logic is contained in methods. Meta data constructs (such as Java Annotations [Sun04]) annotate other constructs with additional information. When model structures are embedded in the source code, reasonable compositions of these constructs must be defined. For this purpose relations between the constructs are used – be they implicit, e.g., that methods are contained in a certain class, or explicit, for example a reference to another class in meta data. The implementation in Java will be explained in section 4 and demonstrated with a larger example. The basic structure is defined by the following rules:

- Each state is represented by a class definition.
- Each state class may contain methods that represent related activities. The actual activity is constituted by the source code in the method body.
- Transitions are defined by meta data referring to the target state and a guard. Depending on the model type, these meta data can be attached to activity methods (e.g. in the case of state machines) or state classes (for process models).
- Guards and updates are implemented as methods without parameters that return a boolean value. The method content consists of logical expressions, either single statements for a certain value or pairs of statements that define value ranges. The

contained code must completely follow these rules so that logical expressions can be extracted for modeling.

- Updates may be referred to from transitions or state classes, depending on if and where a system validation is reasonable after activities.
- To decouple source code elements related to models and arbitrary source code for reasons of clarity, a façade type will be defined that encapsulates the business logic. An instance of this façade, denoted ACTOR in the following, is passed to activity methods by the execution framework.
- The variables defining the considerable system state must be available explicitly to the model, even if they are defined and modified in arbitrary source code. For this purpose another façade type, denoted VARIABLES in the following, will be used to make variable values available to the model execution. This way variables are easy to identify by the method names and thus usable for model extraction.

These simple definitions allow to execute activity models with a simple framework using static reflection: The class structures of an application are examined and state classes are identified. Based on the initial state, possible transition methods are detected and their guards invoked. If a transition can fire, the activity code is executed and connects the state machine to the actual application logic. Afterwards the update method may be called to validate the system state. The next state is then selected from the method meta data.

4 An Instance: Automata and Java Components

The concept sketched above has been used with finite state machines as a specific instance of behavioral models [BSG08]. We will now explain how the concept applies to the selected technologies and describe a non-trivial real-world application (see section 4.2) that has been modeled by means of our approach.

4.1 Concept

The rules for source code structures mentioned above can be applied to Java constructs without change:

- Each state is represented by a class definition implementing a given interface named `IState`, which defines no methods, but allows to type-safely identify these state types. Since Java allows to implement multiple interfaces in one class, a class could be marked to represent a part of more than one model if necessary.
- Each state type contains methods that represent activities. The actual activity is constituted by the source code in the method body. All activity methods must take

no parameters except one instance of a type that encapsulates the business logic of the actual application.

- The activity methods are defined as transitions by meta data (in this case Java annotations) that refers to a target state and a class containing guards and updates.
- Guards and updates are implemented as two methods without parameters that return a boolean value. The method content consists of logical expressions, either single statements for a certain value or pairs of statements that define value ranges. Both methods are executed at run time to decide if a transition can fire and if the activity left the system in an expected state.

4.2 Sample System

To validate that the approach is suitable for the mentioned purposes, we use a load generator component as a sample system. It is part of a larger system for performance testing and is able to generate load according to different measurement modes. The actions of the load generator for each mode are modelled using state machine models. Here, we discuss an exploration mode, in which load is generated by worker threads whose number is adjusted depending on the turnaround time of measurement requests and the acceptable turnaround time range being given by the user.

The state machine to control this application consists of 10 states and 27 transitions. The start state can be left by only one transition to do some preparations and reach a standby state. From here, there is again only one possible transition to perform load generation and reach a state called "AfterMeasurement". Depending on the result of the measurement, the number of load generating workers is adjusted, leading to different states, and another measurement is taken. This process continues until a performance mark is determined and the state machine terminates.

4.3 Code Structures

Some code structures needed in this scenario to represent states, transitions and activities can be seen in figure 1, which is cut out of the source code of the actual system. All model structures are represented by roughly 350 lines of code, being less than 3% of the whole application source code. The class `AfterMeasurementState` shown here implements the interface `IState` and is thus marked to be a state of a state machine. Three transitions are visible in this example, all represented by methods with a `@Transition` annotation. The first parameter of these annotations refers to a class that has implemented the `IState` interface as well. The second parameter refers to classes that implement guards and updates. Activities are simple statements in the method body. In our case, all possible activities are encapsulated by method calls to the `actor` instance.

Figure 2 gives a closer look at the way guards and updates are represented in our frame-

```
public class AfterMeasurementState implements IState {
  @Transition(target = AfterMeasurementState.class,
          contract = RestartContract.class)
  public void restartMeasurement(MeasurementModule actor) {
    actor.increaseNumberOfRestarts();
    actor.doMeasure("Restarted measurement");
  }

  @Transition(target = UpUpState.class,
          contract = BeginUpUpContract.class)
  public void beginUpUp(MeasurementModule actor) {
    actor.resetRestarts();
    actor.beginUpUp();
    actor.doMeasure("Exploration by distance upwards");
  }

  // . . .

  @Transition(target = TerminationState.class,
          contract = AbortContract.class)
  public void abortMeasurement(MeasurementModule actor) {
    actor.terminateMeasurement();
  }
}
```

Figure 1: Class `AfterMeasurementState` with some outgoing transitions

```
public class BeginUpUpContract implements
        IContract<IMeasurementVariables> {
  public boolean checkCondition(IMeasurementVariables vars) {
    return (!vars.getAbort() && !vars.getRestart() &&
            vars.getTooLow());
  }

  public boolean validate(IMeasurementVariables before,
          IMeasurementVariables after) {
    return (after.getNumberOfWorkers() ==
          (before.getNumberOfWorkers() + before.getWorkerDistance()));
  }
}
```

Figure 2: Guards and updates in `BeginUpUpContract`

```
public boolean validate(IMeasurementVariables before,
        IMeasurementVariables after) {
  return (after.getCurrentTurnaroundTime() >= 0 &&
          after.getCurrentTurnaroundTime() <= 10000);
}
```

Figure 3: A validation method that checks for a non-deterministic update.

work. The execution component expects both to be represented in one class that implements an interface named `IContract`. This interface enforces to implement the two methods `checkCondition` for guards and `validate` for updates. Obviously guards can be represented directly by boolean operations on variable values that are retrieved from the VARIABLES façade `vars` via simple method calls. A more interesting case is the representation of updates. Since the actual update is performed by the activities inside the transitions, update definitions from the model can be transformed into validation conditions to monitor the system during run time. Thus, if the model defines that a transition increases the number of load workers by a given delta, the method can check this as shown in the source code example.

Modelling languages for state machines sometimes allow not only updates with fixed values, but also non-deterministic choices from a range of values. This can be expressed in the `validate` method, too, as shown in figure 3. Of course, both in guards and updates the simplest method content would be to return `true`, when a transition is always enabled or a update does not need any validation.

4.4 Generated Models

In our case study, the original implementation was coded manually, but of course the initial code skeletons can be generated from models. Much more important for our approach is the possibility to extract the models from the source code. In our case study, we used graph transformation techniques [Str08] to analyse the syntax and define a mapping to the data format of the UPPAAL system for timed automata [LPY97]. The resulting state machine model is shown in figure 4.

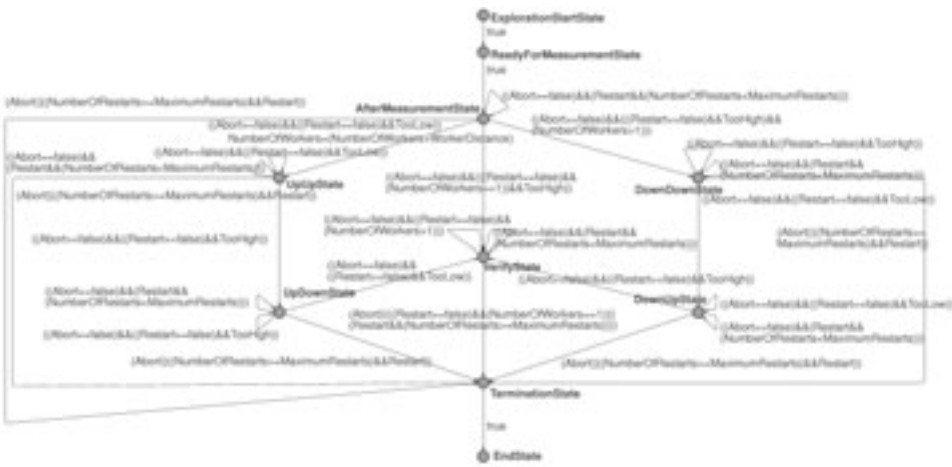

Figure 4: The UPPAAL model

This shows that a model can be easily recovered by a tool from source code which has been

prepared according to the proposed rules. Since the model is independent from the choice of Java, the entire approach is not limited to a single programming language. Moreover, it is thereby possible to relate a PIM directly to source code without touching the PSM. The extracted model could be used to generate state machine structure implementations in other languages as well. The content of these structures, i.e. access to Variables and Actor components, would have to be manually added since they are intentionally considered a black box in our approach.

5 Conclusion

We have seen that object-oriented constructs and basic metaprogramming facilities are sufficient to embed behavioral models into source code and extract all model semantics for development purposes. The real-world application serving as our example shows that a complete state machine representation can be extracted from given JAVA source code and validated in the UPPAAL modeling tool. The way to execute, monitor and debug the model at run time has been outlined. So we have shown that our objective to develop applications in a larger context simultaneously to model specification, validation and simulation for parts of the application is fulfilled. Since no double effort to maintain different representations is required we can state that our approach can effectively be used to avoid maintaining and merging different abstraction layers.

It is important to note that we don't propose our approach as a replacement of existing MDSD technologies for every development task. Systems that need descriptive model specifications being interpreted and potentially modified at run time are clearly not the focus of this code-oriented approach. Instead, integrated systems can be modeled as such and then be used including model semantics. Our approach is applicable in cases when modeling technologies are used for development purposes like system design, validation, documentation and monitoring. Under these circumstances the seamless integration of model semantics into source code structures offers clear advantages over existing approaches.

It also brings the programmer who likes to stick to code to the designer thinking in abstract models together. This is due to the fact that the entire structures – code and models – are kept closely together.

6 Future Work

The approach of embedding models semantics into object-oriented structures being available at development and run time enables a usage of the model information in running systems: We plan to enhance the execution framework so that the system state with variable values, selected transitions and current states can be monitored in real-time. Another similar extension could record all system states and decisions to make events comprehensible ex post. At run time the meta view to the system could be of use to pause and continue applications for debugging or maintenance purposes, as far as the application it-

self utilizes the framework to store its state. Since our approach is not limited to Java as a programming language, we plan to realize the framework in other languages that offer more capabilities for changing code at run time. The framework can then be used as a skeleton for self-adaptive systems that can reflect and change their own model.

At the conceptual level we will explore our approach for applicability with more model types. This concerns behavioral models with different semantics, for example UML interaction diagrams, as well as other model types like data models or component models. We plan not only to validate that our approach supports this models in theory, but also that helpful monitoring tools can exist at run time. For the existing state machine model support we will supplement the creation of hierarchies of embedded models to allow building even larger systems.

Most important to gain acceptance for this approach and to help efficient development processes is the creation of development tools. We plan to implement editors for existing Java development environments that allow graphical creation, manipulation and validation of behavioral models and persist these model structures in the source code constructs mentioned above. This would allow a parallel editing of model semantics and arbitrary source code and not only allow a faster development process, but also a simpler comprehension and validation of existing systems.

References

[AC06] Michal Antkiewicz and Krzysztof Czarnecki. Framework-Specific Modeling Languages with Round-Trip Engineering. In Nierstrasz et al. [NWHR06], pages 692–706.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach*. Springer-Verlag New York, Inc., 2007.

[BIJ06] A. W. Brown, S. Iyengar, and S. Johnston. A Rational approach to model-driven development. *IBM Systems Journal*, 45(3):463–480, 2006.

[BLW05] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In Briand and Williams [BW05], pages 476–491.

[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.

[BW05] Lionel C. Briand and Clay 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*. Springer, 2005.

[Coa95] Workflow Management Coalition. The Workflow Reference Model. Technical Report WFMC-TC-1003 Version 1.1, Workflow Management Coalition, 1995.

[DG06] Stéphane Ducasse and Tudor Gîrba. Using Smalltalk as a Reflective Executable Metalanguage. In Nierstrasz et al. [NWHR06], pages 604–618.

[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.

[FKG90] A. Finkelstein, J. Kramer, and M. Goedicke. ViewPoint Oriented Software Development. In *International Workshop on Software Engineering and its Applications*, 1990.

[FKN+92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2, 1992.

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

[HT06] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[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.

[MB02] Stephan J. Mellor and Marc J. Balcer. *Executable UML*. Addison-Wesley, 2002.

[NWHR06] Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna 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.

[OMG04] OMG. UML 2.0 superstructure specification. Technical report, Object Management Group, 2004.

[PGS01] Doron A. Peled, David Gries, and Fred B. Schneider, editors. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[Str08] Michael Striewe. Using a Triple Graph Grammar for State Machine Implementations. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Proceedings of the 4th International Conference on Graph Transformations (ICGT) 2008, Leicester*, volume 5214 of *LNCS*, pages 514–516, 2008.

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

[VG05] Marek Vokáč and Jens M. Glattetre. Using a Domain-Specific Language and Custom Tools to Model a Multi-tier Service-Oriented Application – Experiences and Challenges. In Briand and Williams [BW05], pages 492–506.

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

[WS05] Hiroshi Wada and Junichi Suzuki. Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In Briand and Williams [BW05], pages 584–600.