Embedding State Machine Models in Object-Oriented Source Code

Moritz Balz, Michael Striewe, and Michael Goedicke

University of Duisburg-Essen
{moritz.balz,michael.striewe,michael.goedicke}@s3.uni-due.de

Abstract. This contribution presents an approach to maintain state machine model semantics in object-oriented structures. A framework is created that reads and executes these structures at run time and is completely aware of the model semantics. The goal is to embed such structures in arbitrary large systems and delegate program control to the framework. Hence we can debug and validate the system at run time and apply monitoring with respect to state machine model characteristics.

1 Introduction

State machines can be comprehensively specified, simulated and verified at design time. We present an approach to retain model semantics in executable systems to allow debugging, validation and monitoring at run time. Our approach will be introduced by a real-world example and formalized later on. The example depicts a load generator application in which we implemented a state machine model that controls program execution and invokes existing business logic.

Traditional ways to translate models into source code by either manual implementation or automated code generation [1] are not suitable for this application: The inherent loss of semantic information entails that models are related to developed systems only by the developer's knowledge [2], thus preventing automatic back tracking of changes [3]. Model Round-Trip Engineering concepts [4] make code synchronisation possible but require manual effort and are thus error-prone [5]. Additionally, generation tools often lack the capabilities to integrate their output into existing systems like our load generation application. Even if only one modeling language is used, the need to regenerate parts of the source code after local changes contradicts a gradual integration [6].

Model execution engines (e.g. Executable UML [7]) can avoid the mentioned problems by interpreting model descriptions. This is not appropriate either, when the system can not be entirely defined in an executable model. This leads to a loss of type information at integration layers between model and residual source code. In addition, bad performance might be experienced due to heavyweight integration layers or necessary data conversion.

Common to these approaches is the permanent existence of different types of model representations at several development stages or parts of the run time

system. Our approach aims at avoiding these differences by storing state machine model semantics explicitly in object-oriented structures. The goal is to embed such structures in arbitrary large systems and delegate program control to the framework. The framework analyzes the code structures and extracts an internal run time representation of the state machine. It walks through the state machine by evaluating guards and updates and invokes methods that represent transitions accordingly. These methods contain arbitrary code and connect the state machine to the application logic. Our approach naturally ensures that the executed system is equivalent to the designed model. Moreover, we can debug and validate the system at run time and apply monitoring with respect to state machine model characteristics. This benefits come at the cost of having to obey rules while writing the related source code structures, but without the (often not realized) effort to maintain the source code and a separate model at the same time. Section 2 of this contribution demonstrates the basic ideas by example, while sections 3 and 4 explain the formal approach and its mapping to a JAVA implementation in detail. Sections 5 and 6 show related work and draw the conclusions.

2 Example

We illustrate our approach on the basis of the mentioned load generator application that has been developed using JAVA. The control mechanism is modelled as a state machine. The program flow starts with some preparations for the measurement. Then an actual measurement run is performed wherein load is generated by worker threads. The result is evaluated and the number of workers is increased and decreased in order to explore the load behaviour of a system under test. The last two steps are repeated until a measurement result is achieved. The states before and after the measurement have transitions that fire depending on the last measurement results. During transitions the application will e.g. increase or decrease workers.

In order to maintain these state machine semantics in the source code in addition to application logic, we create classes that represent states. Methods in these state types represent transitions that invoke business logic and are decorated with meta data referencing the transition target state. This leads to a network of state classes being connected by transition methods that represent the state machine in object-oriented structures, which is partly shown in figure 1.

The state machine starts at the initial state and performs some preparations with the first transitions. Then it performs an actual measurement and reaches a state named "AfterMeasurement" depicted at the top of figure 1. The implementation of this state is shown in listing 1.1 with minor omissions. It shows the way classes and methods are interpreted as states and transitions and how the actual application components are invoked.

State classes are simply marked with the `IState` interface. Transition methods are marked by an annotation that refers to the `target` state class and a `contract`

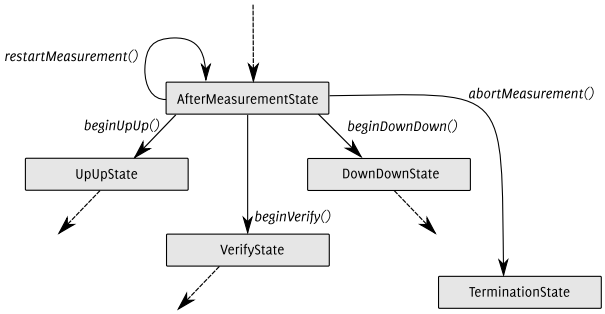

Fig. 1. State classes and transition methods. The node descriptions are class names, the edge labels represent method names.

```
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();
  }
}
```

Listing 1.1. 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()));
  }
}
```

Listing 1.2. Guards and updates in `BeginUpUpContract`

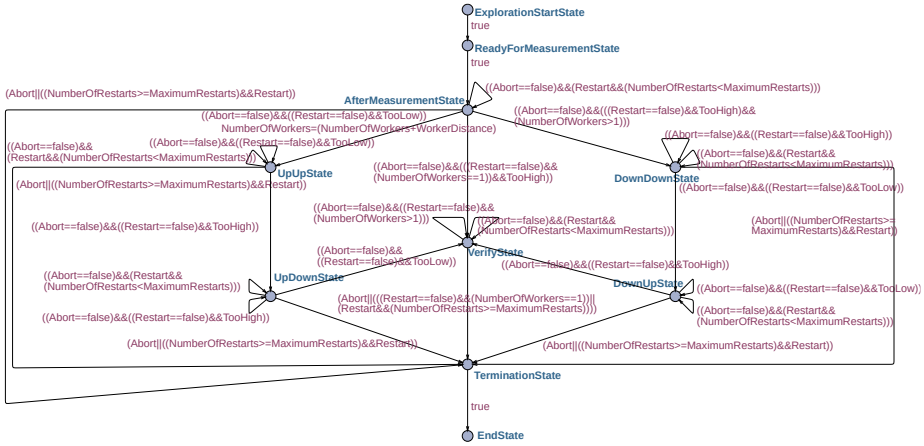

Fig. 2. The state machine model of the load generator

class containing guards and updates for this transition. The method contents use a facade object `actor` that encapsulates the application logic and separates it from the model structures.

Listing 1.2 shows the class containing guards and updates for the `BeginUpUp` transition. Variable values defining the state space are provided by another encapsulating type, denoted `vars`, and used to evaluate guards in the `checkCondition` method. State changes during transitions can be verified with the `validate` method, that does not perform the actual update, but checks whether it has taken place in the implementation as desired. Simple comparisons and logical operations in both methods are mapped one-to-one. Updates are represented as tests for equality as shown in listing 1.2, where the worker increment is validated by checking that $numberOfWorkers' = (numberOfWorkers + distance)$.

Embedding model semantics in code structures allows us to read the complete model at design time and validate it in state machine modeling tools. So far a complete extraction is possible for UPPAAL [8]: The state machine shown in figure 2 is completely extracted from the existing source code and just laid out manually. How to do this is discussed in section 3.4.

As can be seen, these structures are able to cover states, transitions, guards and updates and hence include nearly all state machine semantics. The only missing item is the initial state, which is covered by the execution component necessary to walk through the machine. This will be discussed in section 3.3, after the formalization of our approach.

3 Formalization of the Approach

The code structures containing the state machine model semantics will be executed at run time and used as an input to state machine modelling tools

at development time. Thus we need a universal definition as a formal base for well-defined interpretations. For our approach we define state machines as $M = \{S, T, V, P, U\}$ with

- S a set of states.
- $T \subset S \times S$ a set of transitions between states.
- $V = \{v_1 \dots v_n\}$ a set of named variables.
- $P = \{p_t | t \in T\}$ a set of guards for each transition.
- $U = \{u_t | t \in T\}$ a set of updates for each transition.

The application state is modified only when transitions are fired. Execution control will be passed to application components at this point of time and return to the state machine when the next state is reached. The variables are used at modeling time for state space analysis and are provided at run time by a source code component representing the application state.

Each guard is an expression related to one or more variables that evaluates to a boolean value. Guards will be used at design and run time to decide which transition in the current state should fire. Comparisons and basic arithmetic operations can be performed on variables and literals inside expressions.

Each variable update consists of atomic assignments that define either a single value or a range of values as update for one variable. New values may be constants or variable values which can be connected using basic arithmetic operations as above. Additionally, each variable value of the previous state is accessible to allow relative changes. At design time updates are used to define and change the model state space. At run time they can be interpreted as post-conditions in order to monitor if the application is in an expected state.

3.1 Embedded Model Specification

To represent the model in source code, distinct object-oriented structures will be defined that map to the model semantics. Because they are part of arbitrary source code, arbitrary state spaces will exist beside the well-defined model information. Hence the model must also define interfaces between state machine structures and other source code to pass program execution control and variable values and thus hide the application logic.

So the "Embedded Model" is defined as $\Delta = \{Actor, \Sigma, \Theta, \Lambda, \Phi, \Psi\}$ with

- *Actor* a facade type representing application logic which is invoked during transitions.
- $\Sigma = \{\sigma_s | s \in S\}$ a set of unique identified types that represent states.
- $\Theta = \{\theta_{t,\sigma} | t \in T, \sigma \in \Sigma\}$ a set of methods in state type σ, each representing a transition t.
- Λ an interface defining methods $\{\lambda_v | v \in V\}$ that return the current value for a variable v.
- $\Phi = \{\phi_t | t \in T\}$ a set of methods that implement guard checks for transitions.
- $\Psi = \{\psi_t | t \in T\}$ a set of methods that implement update checks for transitions.

State types implement an interface which defines no methods but allows to type-safely distinguish between state types and other types. Transition methods are designated with meta data that refers to the target state, guard and update implementations. They have no return type and take as parameter an *Actor* instance. The *Actor* type itself has arbitrary, application-specific content and is treated as a black box. Transition methods only make calls to methods provided by the *Actor* instance and therefore respect the conceptual separation between model and code.

The methods in Λ may query the application logic for any application variable at any point of time, but must never manipulate the application state. This way Θ and Λ are the only well-defined interfaces between model and application source code that allow manipulation and query of the application state.

Each ϕ_t returns true iff the pre-conditions of the guard hold. Due to their simple structure described above, guards can be mapped to logical and arithmetical expressions in the source code. Each variable v_n used in guards is represented as the according call of the method λ_n. Obviously the simplest possible guard is that there is none, in which case the source code method instantly returns true.

Each ψ_t returns true iff the variable updates interpreted as post-conditions hold. Parameters taken by this method are two instances of Λ to allow comparisons, one granting access to the current values and one containing cached variable values from the point in time before the transition fired. Since the method does not perform an actual update but validates the state, variable updates are represented similar to guards as logical and arithmetical expressions. Each single value update is replaced by a test for equality and each range update by a pair of comparisons with lower and upper bound. If an update should be left unchecked, the method can return true instantly.

3.2 Representation in JAVA

For an implementation of the concept sketched above, JAVA as a widespread object-oriented programming language and run time environment was chosen. The JAVA-specific constructs and conventions are shortly outlined here. The approach is not limited to JAVA since we can assume that similar concepts exist in other modern object-oriented languages too. A subset of the available declarative structures [9] is used, namely classes, interfaces, methods and annotations [10]. In combination with generic types the latter ensure type safety both for source code and meta information and thus facilitate an accurate source code creation by the developer.

State types are classes that implement the interface `IState`. All methods in the state classes are treated as transitions when decorated with the `@Transition` annotation. It contains an attribute `target` to denote the transition's target state class and an attribute `contract` that refers to a class containing guard and update methods. Λ is realized as an interface providing `get`-methods for each variable λ_v. The interface itself and its implementation are provided by the application developer. The contents of these methods are black boxes, too. It is up to the programmer to ensure that no manipulations of variable values happen when one

of these methods is called. Guards and updates for a transition are located in classes implementing the interface `IContract` with the generic type of Λ. It defines the according methods `checkCondition` and `validate` which return a boolean value and take parameters of the Λ type.

3.3 Model Execution

To execute the state machine model an execution component is required that walks through the state machine by interpreting state class declarations and transitions annotations. In each state the guard methods for each transition method are invoked to determine which transition will fire. Accordingly a transition method itself will be invoked. To start model execution the application passes three parameters to the execution component: The initial state class, the *Actor* instance and the Λ implementation. All other parts of the state machine structure are inferred from these and instantiated on demand.

To save resources, update checks are only enabled in a "debug" mode. In this case the current variable values are cached before a transition fires and afterwards provided to the update method together with the most recent variable values. For this purpose a fourth parameter is passed to the execution component, the Λ interface class, which is needed for dynamic instantiation of this type in JAVA for update methods. In summary, the execution component can access all information related to the state machine model at run time: States, transitions, variable values and their use in guards and updates. This way it is possible to monitor the state machine operation in real-time or to log the information and make activities traceable afterwards with only a few modifications.

3.4 Model Extraction for Design Time Analysis

For design time the Embedded Model is mapped to representations used in modelling tools. Because of the different emphases of existing modeling and verification tools, this cannot be done as universal as for object-oriented structures. Nevertheless the description of states, transitions and variables follows the general concepts of state machines and should hence be directly compatible with any modeling tool. On the other hand it has to be taken into account that the general theoretical concept of state machines is realized in different ways in common modeling techniques [11]. The example presented in section 2 showed the extracted model in the syntax of UPPAAL, which is one sample output for a tool-specific mapping. When selecting an actual modeling tool and formulating the necessary mapping, it must be carefully examined whether the chosen tool provides a syntax powerful enough to express the semantics of guards and updates described above. Checking guards by evaluating variables, logical operators, arithmetic operators and comparators to boolean values can be assumed to be possible in most cases. Updating variables with single values, obtained from variables and arithmetic operations, is a standard technique, too. A range update is interpreted as a random choice of a value from the given interval followed by an update of the variable with this non-deterministic value. More precisely,

the question whether a tool supports range updates is the question whether it supports non-deterministic choices and allows to merge states defining the range of values for a variable into one single state. In our example, UPPAAL supports range updates for variables based on non-deterministic choices. Some minor challenges regarding naming were solved in this example, too. The data type `boolean` is named `bool` in UPPAAL and the `get`-prefix of all variable methods is stripped for better readability.

For this contribution, the model extraction was performed by graph transformations, based on the abstract syntax trees of JAVA and the DOM tree of the UPPAAL data format. Triple Graph Grammars [12] can be applied here for parallel transformations of source code and tool data format with the general state machine model as mapping schema. The detailed description of this graph grammar is beyond the scope of this paper.

4 Discussion

It is important to notice that our approach inverts the traditional direction of model-to-code generators. There is no model that is manipulated at design time and transformed into source code from time to time. Instead there is a permanent model representation in the source code, which is extracted for analysis within modelling tools from time to time. On the one hand this eliminates any effort to maintain and merge different abstraction layers. On the other hand, the chosen approach is not independent from programming languages and execution environments, in our case JAVA, as it is possible when using some other model-driven development technologies [1]. Hence our future work at the tool level aims to enable permanent partial transformations in real-time and hence parallel development of source code and external model representation. At the conceptual level we plan to realize more transformations from model to tools, e.g. into UML state chart diagrams [13] or the CADENCE SMV model checker [14].

The execution component benefits from the permanent representation of the model in code structures. Because of this the actual work done by the execution component is limited to class instantiations and method invocations. Since all dynamic functionality is contained in the invoked methods, the execution is very efficient as regular JAVA code is executed. At the same time the state machine model integrates in arbitrary business logic without enforcing restraints on the non-model code. On the other hand, the developer has to take care to organize the source code accordingly: The approach will only work if the clear separation is maintained and only valid expressions are used in methods whose content is interpreted, i.e. guard and update methods. To detect errors here is possible only if the model is interpreted at design time.

At a more general conceptual level, we aim to analyze the application of our general concept to domains and modeling methods other than state machines. Especially in these cases additional benefits can be expected for larger projects, because one change in source code may influence more than one embedded model.

5 Related Work

The attribute-oriented programming approach [15] with similar use of meta data in code structures has been explored to map UML models to code structures [16]. However, this does not leverage the principle of having only one representation for model and source code and does not avoid round trip engineering. The same applies to Framework Specific Modeling Languages [17], which could be of use if a state machine framework would control the application state. The concept of "executable UML" [7] tries to overcome inconsistencies between different representations by the use of automated transformations or by defining a primary representation that may generate and override all other representations. Our approach uses automated transformations to create the model from the source code and vice versa, but inside the source code the model is combined with non-model parts of the application, thus enabling a seamless integration into larger applications.

Different to the JAVA MODELING LANGUAGE (JML) [18], which offers a huge syntax for specification annotations, we do not aim to present a notation for the specification of all possible system models. This applies also to the approach to use Smalltalk with it's introspection capabilities as a meta language [19]. Contrary to JAVA PATHFINDER [20] our approach does not consider a whole application as the model, but only selected parts of it. Hence our approach can be more complete and formally founded and thus be used for explicit representation and validation in this limited domain of state machine specifications.

6 Conclusion

In this contribution we proposed to embed state machine model semantics in source code structures and extract concrete model representations on demand. The model execution and extraction components have been outlined. As shown by example, we can extract a complete state machine representation from given JAVA source code. All of the source code structures in the Embedded Model are used without change to execute, monitor and debug the model at run time. Hence the objective to let application development in a larger context happen simultaneously to model specification, validation and simulation for parts of the application without double effort to maintain two abstraction levels is fulfilled. So we can state that our approach can effectively be used to avoid maintaining and merging different abstraction layers.

References

1. Brown, A.W., Iyengar, S., Johnston, S.: A Rational approach to model-driven development. IBM Systems Journal **45**(3) (2006) 463–480
2. Tichy, M., Giese, H.: Seamless UML Support for Service-based Software Architectures. In Guefi, N., Artesiano, E., Reggio, G., eds.: Proceedings of the International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI) 2003,

Luxembourg. Volume 2952 of Lecture Notes in Computer Science., Springer-Verlag (November 2003) 128–138
3. Baker, P., Loh, S., Weil, F.: Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In Briand, L., Williams, C., eds.: Model Driven Engineering Languages and Systems. Volume 3713 of LNCS. (2005) 476–491
4. Sendall, S., Küster, J.: Taming Model Round-Trip Engineering. In: Proceedings of Workshop on Best Practices for Model-Driven Software Development. (2004)
5. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal **45**(3) (2006) 451–461
6. Vokáč, M., Glattetre, J.M.: Using a Domain-Specific Language and Custom Tools to Model a Multi-tier Service-Oriented Application – Experiences and Challenges. In Briand, L., Williams, C., eds.: Model Driven Engineering Languages and Systems. Volume 3713 of LNCS. (2005) 492–506
7. Mellor, S.J., Balcer, M.J.: Executable UML. Addison-Wesley (2002)
8. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Int. Journal on Software Tools for Technology Transfer **1**(1–2) (Oct 1997) 134–152
9. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java(TM) Language Specification, The 3rd Edition. Addison-Wesley Professional (2005)
10. Sun Microsystems, Inc.: JSR 175: A Metadata Facility for the Java™ Programming Language http://jcp.org/en/jsr/detail?id=175.
11. Crane, M.L., Dingel, J.: UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal. In Briand, L., Williams, C., eds.: Model Driven Engineering Languages and Systems. Volume 3713 of LNCS. (2005) 97–112
12. Schürr, A.: Specification of graph translators with triple graph grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Graph-Theoretic Concepts in Computer Science. Volume 903 of LNCS. (1994)
13. OMG: UML 2.0 superstructure specification. Technical report, Object Management Group (2004)
14. McMillan, K.: The Cadence SMV Model Checker http://www.kenmcmil.com/smv.html.
15. Schwarz, D.: Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. ONJava.com (June 2004)
16. Wada, H., Suzuki, J.: Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In Briand, L.C., Williams, C., eds.: MoDELS. Volume 3713 of Lecture Notes in Computer Science., Springer (2005) 584–600
17. Antkiewicz, M., Czarnecki, K.: Framework-Specific Modeling Languages with Round-Trip Engineering. [21] 692–706
18. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design. In Kilov, H., Rumpe, B., Simmonds, I., eds.: Behavioral Specifications of Businesses and Systems, Kluwer (1999) 175–188
19. Ducasse, S., Gîrba, T.: Using Smalltalk as a Reflective Executable Meta-language. [21] 604–618
20. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering Journal **10**(2) (2003)
21. Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of Lecture Notes in Computer Science., Springer (2006)