Monitoring Model Specifications in Program Code Patterns

Moritz Balz, Michael Striewe, and Michael Goedicke

Paluno - The Ruhr Institute for Software Technology University of Duisburg-Essen, Essen, Germany {moritz.balz,michael.striewe,michael.goedicke}@s3.uni-due.de

Abstract. Numerous approaches exist that derive executable systems from well-defined specifications. However, model specifications are not available in program code of such derived systems, which impedes continuous validation and verification at run time. We earlier proposed to embed model specifications into well-defined program code patterns to bridge this semantic gap. We now present an elaboration of our approach to monitor such systems at run time with respect to the underlying abstract models. For this purpose, different techniques are considered that allow to access the modeling information without relying on additional metadata. Based on this, we present a tool that monitors the execution of state machines.

1 Introduction

The creation of software based on formal models is supported by means of various modeling, simulation and verification tools. However, current technologies for model-driven software development (MDSD) cause a loss of semantic information when such models are transformed into source code by manual or automated code generation [1]: The inherent loss of semantic information entails that models are related to derived systems only implicitly [2], thus preventing us from being able to monitor the execution with respect to the model semantics.

To bridge this semantic gap, we proposed to embed model specifications in object-oriented program code [3], for example for state machines [4]. Such *embedded models* introduce program code patterns representing the abstract syntax of models. This single-source approach allows not only to verify programs at development time with respect to the related models, but also to execute embedded models at run time by frameworks relying on structural reflection. In this contribution we consider this an opportunity to monitor the execution: Since these program code patterns represent model specifications completely, different degrees of abstraction are available in the code at the same time. Hence we can monitor model execution at run time without using other representations than the program code. At the same time we can observe how models behave with application data.

This paper is structured as follows: Section 2 describes the monitoring approach by introducing concepts for embedding and identifying modeling information in program code. Then we describe different possible techniques to access

the related program code fragments at run time in section 3. Based on these, a tool for monitoring state machines is introduced in section 4. Afterwards we give an overview of related work in section 5 and draw conclusions in section 6.

2 Approach

The objective of this contribution is to monitor executed software with respect to high-level specifications, but without using additional meta information, so that no inconsistencies can occur and the tool chain is as small as possible. While monitoring as a way of verifying the execution of software systems at run time is well-established, few approaches realize verification with respect to formal models the software is based on. The reason, as mentioned in the introduction, is that the related specifications are not naturally available in the program code that constitutes programs at run time: The code usually describes execution logic only and not its abstract semantics. When it is monitored or verified, the resulting information is generic, focuses on technical details of program code, or must rely on tracing metadata to relate the code to formal models. Considering these problems, we introduce in this section our general approach of coupling model specifications and program code.

2.1 Embedded Models

A monitoring as described above means that the program code must contain the specification information. Considering object-oriented programming languages like Java, we can observe a trend to increase the expressiveness of program code fragments. For example, *embedded DSLs* [5] are domain-specific languages that are embedded into other languages, so that semantics of DSLs are used inside a general-purpose language. In addition, some general-purpose languages are able to carry type-safe metadata, e.g., Java Annotations. This enables *attribute-enabled programming* [6] making program code interpretable even at run time.

Embedded models build upon these concepts to relate program code to abstract specifications systematically. Each embedded model provides a program code pattern representing the abstract syntax of a formal model so that a bijective projection between both exists. The pattern elements rely on the semantics of the underlying programming language and its expressiveness regarding single fragments and their interconnections. The statical elements of the programming language and their relations are considered building blocks constituting the pattern. They are of interest in our context since expressiveness of the monitoring depends on their accessibility by appropriate mechanisms at run time.

The pattern code is interpreted by means of structural reflection at run time to execute the model specifications. Each embedded model provides an execution framework that accesses and invokes the language elements and thus creates a sequence of actions matching the related model semantics. Considering the monitoring, it is essential that the program code pattern elements and their expressiveness regarding relations to the abstract specifications are by this means accessible at run time.

2.2 Implementation for State Machines

An instance for embedded models exists for the domain of state machines. Since meaningful monitoring in our context depends on the availability of model elements in the program code at run time, we will introduce the program code pattern here and refer to it later. Figure 1 shows an example containing all program code structures of interest. The class at the top represents a state; the class name equals the name of the state. The method in the state class represents a transition. It is decorated with metadata (the annotation **@Transition**) referring to the target state class and a "contract" class containing guards and updates. An interface type referred to as "actor" is passed to transition methods. Its methods are interpreted as action labels which can be called when a transition fires. Thus, a sequence of actor method invocations inside a transition method is interpreted as a sequence of action labels for this transition.



Fig. 1. A state definition with an outgoing transitions and its contract. The first method of the contract evaluates a pre-condition with respect to the current variable values, while the second method evaluates a post-condition by comparing the current values to the previous values.

Guards and updates are implemented as two methods in a "contract" class which is shown at the bottom of figure 1. Both evaluate boolean expressions and return a single boolean value. The guards use the current variable values of the state machine to determine if a transition is able to fire, the updates compare the current values with the values from the point in time before the transition fired to determine the changes to the state space. For this purpose both methods access a "variables" type which is a facade type representing the variables constituting the state space of the state machine. This type contains "get" methods for each variable. The name and return type of each method are interpreted as name and data type of the corresponding variable.

The execution framework interprets and invokes these fragments at run time. The surrounding program code accesses for this purpose the execution framework and passes the class definition of the initial state as well as the variables and actor facade types as parameters. The state machine is then executed as follows:

- 1. The initial state's class and variables interface are passed to the execution framework. All states reachable from the initial state are instantiated.
- 2. The current state is set to the initial state.
- 3. All transition methods of the current state are visited and the variables type instance is passed to the related guard method to determine if the transition is able to fire.
- 4. The current variable values are cached.
- 5. The method representing the transition that is able to fire is invoked.
- 6. The current variable values and the cached variable values are passed to the update method for the validation of variable updates.
- 7. The current state is set to the target state of the executed transition. The process is continued until the current state is a final state or the state machine runs into a deadlock.

2.3 Monitoring at Run Time

As can be seen in the state machine example, embedded models introduce program code patterns whose elements are related to model specifications. The models are thus views on the program code and need not to be stored in separate notations, so that no inconsistencies between model and implementation can occur. Consistency is not only maintained at development time, but also at run time: Since the related code fragments are not supplementary or optional, but instead used by the execution framework, executed systems with embedded models carry complete information about related specifications naturally.

This availability of models at run time is important for our objective to monitor programs with respect to models, since the model views can be extracted from the code during execution. For this purpose the well-defined elements of the program code patterns serve as entry points for interpreting and monitoring the program code. This enables a validation of programs with two purposes: First, the model view itself is of interest for monitoring the model execution by the framework, so that inferences can be made on correctness of the model from this information. Second, embedded models are tightly integrated with arbitrary program code. This allows for high flexibility during implementation, but causes the need to validate correctness of the surrounding code with respect to the model. This is supported with appropriate monitoring since the behaviour of the model with application data can be observed.

Monitoring of embedded models thus considers program code pattern instances, for example of state machines, as well as program code of the execution framework: Since it controls the execution, it is an entry point for actions to be monitored. Inside the execution framework for state machines, the following steps can be considered:

- The execution framework iterates on the state machine flow until a final state is reached. The current active state is denoted by a variable inside this iteration pointing to the state class definition. Changes to this variable must be monitored in order to determine state activation.
- Once a state is activated, the execution framework iterates the contained transition methods. The transition under examination is also denoted by a variable that must therefore be observed.
- For each transition the execution framework invokes the guard and update methods and passes the variables facade instance as a parameter. Of interest are all operations inside this methods, especially those that comprise state machine variable values. To interpret the guards and updates thoroughly, the composition of the overall result of these methods from single operation results is also important to monitor.

We will now introduce appropriate monitoring techniques and afterwards a tool that implements the approach.

3 Monitoring Techniques

Our objective is to use this approach for monitoring program execution with respect to models at run time, but without artificial tracing information. Thus it is important to consider the accessibility of the program code patterns and their elements during execution. We will introduce the basic technological approaches for this purpose here. While all of them have already been used by other approaches for monitoring, our contribution here is the application to program code patterns carrying the abstract syntax of formal models. We will therefore not focus on the general technologies, but on their adequacy for monitoring the references to model specifications at run time, in which we encounter important differences.

3.1 Listener Approach

Since all information about the running system and the embedded state machine semantics is available inside the state machine execution framework, the easiest way for monitoring is to extend this framework in order to emit information of interest for monitoring. The execution framework is based on structural reflection and accesses and interprets a considerable part of the program code structures constituting the pattern. Besides setting listeners programmatically, modulebased platforms (like OSGi [7] in the context of Java) allow for a loose coupling of execution framework and components receiving information about the execution. In the case of state machines, listeners can be notified about events for every operation performed on the embedded model:

- Initialization and start of a state machine. This includes information about all states, transitions and variables as extracted from the Java code via reflection. States are uniquely identified by their fully qualified class names.
- Activation of states. This indicates that guard evaluation and transition selection in this state will happen subsequently.
- Selection of transitions. This indicates that program control will be handed over to the business logic in this transition.
- Validation of updates after a transition. The variable values are updated in this event. Additionally, the cached variable values are supplied to allow for comparisons. Additional information is supplied if the validation failed. When this event is fired, program control has been taken over by the state machine execution framework again.

The advantage of listeners is their easy integration into tools based on the Java platform, especially in module-based environments. Since the listeners are accessible from inside the same Java Virtual Machine (provided appropriate programming interfaces or module lookup services exist), even self-monitoring of applications is possible. Thus an application can gain information about its own execution inside the state machine. This is possible without concurrency problems since the framework passes control of the program flow to the listeners during notifications, so that all actions are handled sequentially.

While the approach is working at this level, the degree of detail is limited: Method contents in Java are not accessible by means of reflection and thus black boxes. For this reason operations inside guards and updates are not visible, but only their results after the related method was invoked by the framework.

3.2 Aspect-Oriented Approach

Aspect-oriented programming (AOP) aims to separate cross-cutting concerns from business logic. Monitoring and tracing are often-mentioned examples for AOP usage: Emission of monitoring information is formulated as aspects that are woven into program code. To monitor state machine execution, the code structures of interest are accessed by pointcuts. Appropriate advice written in AspectJ [8] are shown in listing 1.1. The first and the third pointcut wrap around guard and update methods, invoke them and read the result. Afterwards the monitor is notified about the contract class and the current result. The second pointcut is invoked before a transition method is executed, i.e., any method in a class implementing the IState interface. It notifies the monitor about the related state class and transition method name.

```
// \ensuremath{\texttt{W}}\xspace{\texttt{rap}} guard method invocation and notify about the result
boolean around(Object vars) : execution(* IContract.checkCondition(..)) && args(vars) {
 boolean result = proceed(vars);
 monitor.notifyGuard(thisJoinPointStaticPart.getSignature().getDeclaringType(), result);
 return result;
3
// Notify about forthcoming transition method invocation
before() : execution(* *.*(..)) && target(IState) {
 monitor.notifyTransition(thisJoinPointStaticPart.getSignature().getDeclaringType(),
    thisJoinPointStaticPart.getSignature().getName());
}
// Wrap update method invocation and notify about the result
boolean around(Object before, Object after)
       execution(* IContract.validate(..)) && args(before, after) {
 boolean result = proceed(vars);
 monitor.notifyUpdate(thisJoinPointStaticPart.getSignature().getDeclaringType(), result);
 return result;
```

Listing 1.1. The AspectJ monitoring aspect. All points of interest in the program code pattern are clearly identifiable by simple rules regarding their classes and method names, so that pointcuts can be defined unambiguously.

The main advantage of AOP in this context is that monitoring can be applied without the need to modify the execution framework. With load-time weaving, monitoring capabilities can even be supplemented in systems after the program code has been compiled. This allows for flexible mechanisms that can be applied depending on the context. This is enabled by the fact that the pattern elements of embedded models are well-known and obligatory: Aspects can identify them so that advice and pointcuts can address program code elements related to model elements. Similar as with listeners, this also allows for self-monitoring.

However, while this exterior view on the pattern allows for dynamic extension of such software, it prevents full access to information of interest: Pointcuts can handle information regarding the location of program code in which they are executed (keyword thisJoinPointStaticPart). But, they do not gain access to information in terms of sequences of pointcuts: In each state, a certain number of guards is evaluated. Afterwards, one transition method is invoked. While pointcuts are informed about the single actions, they cannot determine which guard belongs to the transition being executed; this information has to be guessed or supplemented by interpreting the program code afterwards. To solve this problem, the execution framework could be changed to make pointers to the objects of interest available as fields.

3.3 Debugging Approach

The debugging approach delegates low-level observation of the program state to the executing platform. The related Java Platform Debugger Architecture (JPDA) [9] provides well-defined programming interfaces to access related events so that those of interest for our monitoring approach can be filtered from the event queue. In the case of embedded state machines, state activation and transition selection are monitored by observing fields containing the related references in the execution framework with ModificationWatchpointEvents. For guards and updates, MethodExitEvents are of interest that are triggered after all code of a method has been executed, but before the method is left. We use them to access return values of variable interface methods when they are invoked. Together with information about local variable values we can monitor evaluation of guards and updates with such events, too: Since only expressions are used inside these methods, the evaluation is fully comprehensible afterwards by inspection of the values of local variables. The return value of the method and thus the result of the evaluation is also available in this event.

A debugger can hence access all elements of the program code pattern in model implementations as well as all local variables in the execution framework. Different to the listener and AOP approaches, this allows for monitoring guard and update method contents. Since all details of expressions are available, the evaluation of guards and updates can be recorded and presented to the developer for each step. The debugging approach is therefore the only one able to access all elements of the program code pattern. Access to variables and method invocation results is possible without additional effort when they are accessed by the application itself. For the state machine model this is sufficient since the variables are of interest only when they are evaluated in guards. A debugger would also allow to invoke methods at any time. This could be of interest for variable methods to determine their current value. This is, however, intrusive to the program flow, since variable methods may contain arbitrary business logic, which would be executed at times not expected by the developer.

The main influence of debuggers, however, is the need for two running instances: The application being debugged and the debugger itself that controls execution. All information that can be gained is accessible only by the latter, so that a self-monitoring of applications is not possible. In addition, debuggers in general have a strong impact on performance, so that a monitoring of production systems is currently not desirable with this technology. We thus expect that this approach can be used as debuggers are used in general – when the applications are validated during development or maintenance. In this case the relation to abstract models is more meaningful than debugging at the source code level only.

4 Monitoring Tool

These approaches enable monitoring of program code based on embedded models without using tracing information or other metadata, but by considering welldefined code structures only. We will now introduce a tool that is based on such approaches and monitors the related information. Its user interface shown in figure 2 reflects our requirements for the practical use of the monitoring.

The graphical view allows to watch activated state classes and transition methods. Current and cached variable values are shown to exhibit the current state space and to enable monitoring of changes during transitions. Updates that



Fig. 2. A state machine model being monitored. Left hand we see the state machine with the active state and transition highlighted, right hand the variable values constituting the state space.

could not be validated successfully are listed separately; since updates do not have impact on the program flow, this information allows developers to look for the causes of such inconsistencies later on. The state machine flow altogether can be paused and resumed by the user. This is possible since business logic is invoked during transitions, and execution control will afterwards return to the state machine. The third button visible on top of the screenshot notifies the execution framework that the state machine flow should pause after the current transition; the button to the right allows then for stepwise execution.

The tool is realized on the Eclipse platform, making it easy to be integrated in Eclipse-based development tools. It uses listeners that are loosely coupled over the OSGi service registry that is provided by the Eclipse platform: Listeners like our tool are hence OSGi bundles being deployed alongside, but independent from business logic. The listener is registered as a named OSGi service that is detected by the execution framework. The resulting architecture as sketched in figure 3 allows to use almost arbitrary tools to be notified about events for every operation performed on the embedded state machine.

5 Related Work

Following our objective to monitor the execution of program code that is related to model specifications, we must consider related work with respect to two



Fig. 3. Component architecture with the monitoring listener. Applications are composed of components using the execution framework based on the OSGi platform. The listener component is optional and hence only coupled via the service registry.

topics: First, general approaches that relate program code to high-level specifications which are in theory appropriate for monitoring; second, the application of monitoring in specific technological environments.

Round-trip engineering [10] relates generated program code to models but targets the development time instead of the run time and cannot be fully automated [11]. Informal specifications can be inferred from program code by detecting patterns [12], and similar, specifications can be extracted from program code based on design patterns [13]. However, this requires manual effort or is based on heuristics and not appropriate for a precise monitoring. Executable models [14] are accessible at run time, too. However, they are only appropriate for applications completely expressed as models, while we consider cases where models are connected to program code and thus monitor the related data exchange.

Monitoring for compliance with so-called *design models* [15] or design pattern contracts [16] is based on low-level semantics of detailed patterns. Similarly, model checkers for program code work with low-level semantics of the programming language and thus consider whole applications as models [17]. In contrast, monitoring with embedded models is related to abstract specifications. For this reason it can also clearly be distinguished from general debugging approaches.

We do not aim to present a notation for the specification of all possible system models like the JAVA MODELING LANGUAGE (JML) [18] or the approach to use Smalltalk with its introspection capabilities as a meta language [19]. In contrast to static analysis tools like DISCOTECT [20] we do not target detection of unknown structures and models, but focus on well-known models that can thus be examined more thoroughly and with respect to a formally-founded background. We also do not require changes in the program code to introduce references to specifications as is necessary for PATHFINDER's verification statements [21] or the approaches to monitor OCL constraints with aspect orientation [22, 23], which rely on metadata in source code comments. Instead, we can infer all model specifications directly from the program code pattern.

6 Conclusion

We presented our approach to monitor model specifications that are embedded in object-oriented program code. We were acting on the assumption that the related program code pattern structures are precise enough to allow for inference to model specifications even at run time. To show this, different approaches for information retrieval have been evaluated as possible alternatives. Our conclusion is that all are appropriate to monitor the state machine semantics, although in different degree of detail and with different impact on the necessary changes to the program code. All are non-intrusive regarding the source code of the monitored system and two of them are even non-intrusive to the source code of the execution framework. However, the degree of detail varies since only debugging approaches allow to monitor guards and updates in detail. On the other hand, listeners and AOP require less overhead at run time. With AOP, monitoring aspects can even be attached dynamically to the programs since they can work on the pattern specifications after compilation.

For the current implementation of a monitoring tool, the listener approach was chosen since it allows to access the most important information with little effort and provides the ability for self-monitoring. However, if the required environment is available, the debugging approach is more thorough and allows to monitor every detail of the state machine execution. Future work will thus include the development of an appropriate monitoring tool. Due to the maturity of the JPDA and related user interfaces in integrated development environments, we will then be able to integrate the monitoring with the debugging user interface of development environments. With this integration, the monitoring of abstract model specifications can be seamlessly integrated with debugging of arbitrary Java applications, thus making model validation at run time an integral part of the development process.

References

- Brown, A.W., Iyengar, S., Johnston, S.: A Rational approach to model-driven development. IBM Systems Journal 45(3) (2006) 463-480
- 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
- Balz, M., Striewe, M., Goedicke, M.: Continuous Maintenance of Multiple Abstraction Levels in Program Code. In: Proceedings of the 2nd International Workshop on Future Trends of Model-Driven Development - FTMDD 2010, Funchal, Portugal. (2010) 68-79
- Balz, M., Striewe, M., Goedicke, M.: Embedding State Machine Models in Object-Oriented Source Code. In: Proceedings of the 3rd Workshop on Models@run.time at MODELS 2008. (2008) 6–15
- Kabanov, J., Raudjärv, R.: Embedded Typesafe Domain Specific Languages for Java. In: PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, New York, NY, USA, ACM (2008) 189–197
- Schwarz, D.: 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.

- OSGi Alliance: OSGi Service Platform, Core Specification, Release 4, Version 4.1. IOS Press, Inc. (2005)
- Colyer, A., Clement, A., Harley, G.: Eclipse AspectJ. Addison-Wesley (2004)
 Sun Microsystems, Inc.: JavaTM Platform Debugging Architecture API http://
- Sun Microsystems, Inc.: Java^{1M} Platform Debugging Architecture API http:// java.sun.com/javase/technologies/core/toolsapis/jpda/.
- 10. Sendall, S., Küster, J.: Taming Model Round-Trip Engineering. In: Proceedings of Workshop on Best Practices for Model-Driven Software Development. (2004)
- 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, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings. Volume 3713 of LNCS., Springer (2005) 476-491
- Philippow, I., Streitferdt, D., Riebisch, M., Naumann, S.: An approach for reverse engineering of design patterns. Software and Systems Modeling 4(1) (February 2005) 55-70
- Mili, H., El-Boussaidi, G.: Representing and Applying Design Patterns: What Is the Problem? In Briand, L.C., Williams, C., eds.: MoDELS. Volume 3713 of Lecture Notes in Computer Science., Springer (2005) 186-200
- Hen-Tov, A., Lorenz, D.H., Schachter, L.: ModelTalk: A Framework for Developing Domain Specific Executable Models. In: Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling. (2008)
- Sefika, M., Sane, A., Campbell, R.H.: Monitoring Compliance of a Software System With Its High-Level Design Models. In: ICSE '96: Proceedings of the 18th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (1996) 387-396
- Soundarajan, N., Hallstrom, J.O., Tyler, B.: Monitoring Design Pattern Contracts. In: Proceedings of the the 3rd FSE Workshop on the Specification and Verification of Component-Based Systems. (2004) 87-93
- Holzmann, G.J., Joshi, R., Groce, A.: Model driven code checking. Automated Software Engineering 15(3-4) (2008) 283-297
- 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
- Ducasse, S., Gîrba, T.: Using Smalltalk as a Reflective Executable Meta-language. In 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. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 604-618
- Yan, H., Garlan, D., Schmerl, B., Aldrich, J., Kazman, R.: DiscoTect: A System for Discovering Architectures from Running Systems. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2004) 470-479
- 21. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering Journal **10**(2) (2003)
- 22. Richters, M., Gogolla, M.: Aspect-Oriented Monitoring of UML and OCL Constraints. In: Proceedings of Workshop Aspect-Oriented Software Development with UML. (2003)
- Chen, F., D'Amorim, M., Roşu, G.: A formal monitoring-based framework for software development and analysis. In: Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04). Volume 3308 of LNCS., Springer-Verlag (2004) 357-373