Model-based Monitoring of Integrated UML State Machine Models and Code

Marco Ehl[1], Marco Konersmann[2]

Abstract: Logging and monitoring are important in long-living software to understand how the software behaves in production. In current model-based software development technology, monitoring information is still either shown for the code level or the model level. The mapping between runtime information from the generated—and usually manually extended—code and the model has to be made manually. In this paper, we present an approach for creating integrated model/code runtime views for UML state machines in Java code, by exploiting code generation templates that allow for mapping models and code at runtime. We evaluated the applicability of our approach and the performance of our prototype tool in artificial use cases. We found that the approach is applicable in that context, but that there is a considerable performance overhead in the current implementation.

Keywords: model-based software engineering; logging; monitoring; UML state machines; Java

CEUR-WS.org/Vol-2814/paper-A5-1.pdf

1 Introduction

Logging and monitoring [vHWH12] are central concepts for maintaining long-living software systems. While many software systems use logging and monitoring to collect information about the current status of the execution and its history, this information alone is insufficient in model-based software engineering (MBSE) [Vö06]. Model-based systems are to some extend developed on a higher level of abstraction than the source code. These models of the system are used for generating code. Further code is *implanted* into the generated code as behavior descriptions, and *contextual* code links the generated code. Monitoring then happens via logging on the level of source code, by using source code log statements to capture state changes or similar events that happen during the execution of software. The information attached to a log event is usually a timestamp and brief description of the event in question. There are challenges with this type of logging in MBSE: (C1) Where and how to log events is usually not specified for a project but is left to the developers. When there is no such clear logging concept, the logging of events is not systematic and therefore often does not provide a good source of information. (C2) Log events can answer the question of what happened (observed behavior), but it does not answer why it happened and what should have happened (intended behavior). This is a problem because the deviations between the observed behavior and the intended behavior are of special interest.

[1] Universität Koblenz-Landau, Institute for Software Technology, Germany, mehl@uni-koblenz.de
[2] Universität Koblenz-Landau, Institute for Software Technology, Germany, konersmann@uni-koblenz.de

Copyright © 2021 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

A systematic logging with relations to the model and the code can help to overcome these challenges for a more efficient monitoring and debugging process of model-based systems. Model-based development has the potential to improve logging by employing code-generation techniques. When logging is implemented into code generation templates, log events can be systematically generated. But this log information only explains the model-based behavior. If the implanted or contextual code has issues or uses the generated code in a wrong way, code-based logging information is necessary. To holistically learn about the behavior, log information needs to explain the behavior on the model-level and on the code-level in combination. In this paper, we present our approach to integrated logging and monitoring of UML state machine models in Java source code. Our contributions are specifically: (1) a method for an integrated monitoring of UML state machines and the execution of Java code, (2) code generation templates for UML state machines, that incorporate systematic logging on the model and the code-level, (3) a prototype tool that implements the method and code-generation, and (4) an application and performance evaluation of our tool in artificial use cases.

We present an approach to create a unified view on the system behavior, that gives access to logged events at runtime, showing the code and model level, taking the meta model information into account for providing context information to developers. The remainder of this paper is structured as follows: Section 2 presents an example use case. Section 3 describes our approach on a conceptual level. Our implementation is sketched in Section 4. Section 5 shows the application on a model-based implementation of an ATM and discusses our approach. We describe related work in Section 6 before we conclude in Section 7.

2 Motivation Example

This section introduces a running example use case for showing the benefits of our approach for logging and monitoring with integrated UML state machine models and code. We will show the logging via the evaluation of state machine transition guards. For this event, we describe all the steps from the meta model specification to the monitoring system.

State machines are a powerful and intuitive tool for describing how a system reacts to events. It comprises states and transitions between states. Each state represents a specific behavior and transitions define changes between states depending on events and conditions. When the state of the state machine is changed, the observed behavior of the state machine changes too. Harel [Ha87] introduced the concepts of nesting and orthogonality to state machines, which were adapted by Object Management Group (OMG) for the UML Unified Modeling Language (UML) [OM17]. UML is a widely used general-purpose modeling language. In this work, we use UML state machines as the meta model for state machines.

As a use case, we have chosen an automated teller machine (ATM) which we want to monitor during its execution (see Fig. 1). The ATM consists of multiple states and reacts to events. A customer of a bank using the ATM would usually find the machine in an idle state.

When a bank card is inserted, the machine will start serving the customer. The card will be read for authentication, a transaction will be selected and executed, and finally, the card is ejected. After this, the machine goes back to the idle state for the next customer. Other states define the behavior when the machine starts up or when an error occurs.

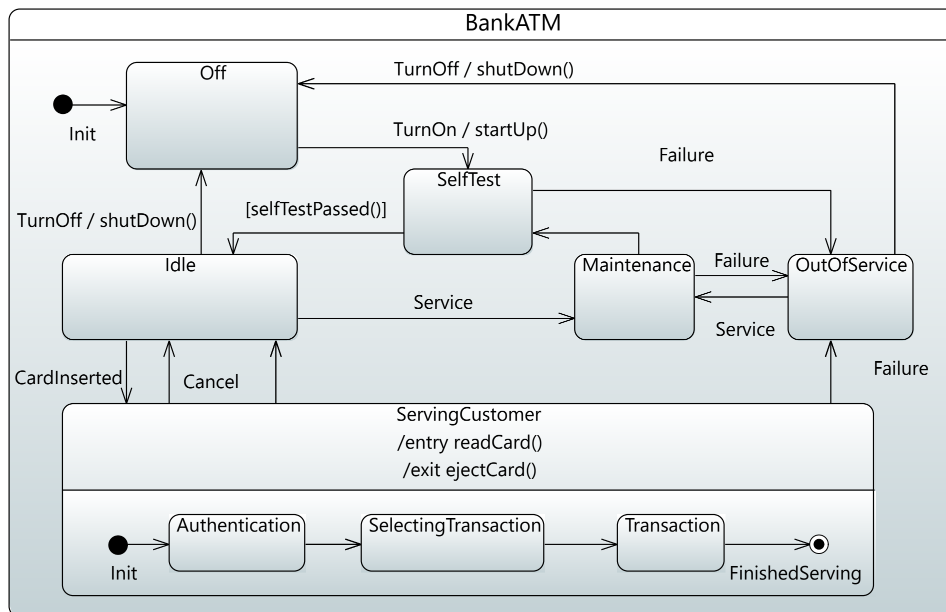

Fig. 1: UML state machine model of an automated teller machine (ATM). Guard and effect names are simplified for presentation purposes.

3 Integrating Runtime Information of UML State Machine Models and Code

In this section, we give an overview of the methods and artifacts used and generated in our work. Fig. 2 shows the artifacts of this work and their relations. The state machine elements must be mapped to corresponding code templates that implement the execution semantics. Based on the templates, code can be generated, that implements two artifacts: code that contains the business logic and the instrumentation code for logging. We deliberately chose to separate both artifacts. This allows, e.g. a programmer, to only focus on the business logic. The instrumentation code is later weaved together with the business logic. The resulting artifact is instrumented code, which is an executable representation of the model. Executing the instrumented code will generate events that can be captured and analyzed by monitoring logic. We created a list of events to capture during the state machine execution from systematically analyzing the semantics of each element in the meta model. The monitoring

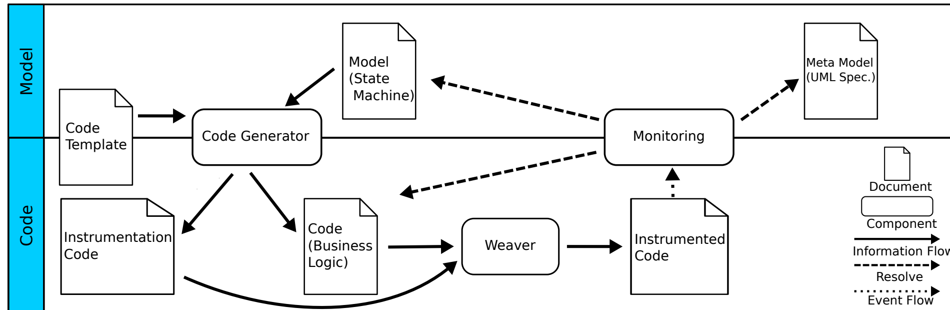

Fig. 2: The artifacts of this work and their relations.

logic takes events and resolves code, model, and meta model information that are related to the observed event.

3.1 Connecting model and code

Modeling is not new to software development. Models have long been used to document the inner structure of software. However, models do not always have the same status as code. Manually updating the model is time-consuming and therefore expensive. This leads—especially when deadlines have to be met—to neglecting to update the model. As a result the models become outdated and less useful [Vö06]. MBSE gives the model the same meaning as the source code. The model is not only used as documentation, but as a part of the software that can increase the quality and the speed of software development [Vö06].

In this work, we build on the model integration concept (MIC) of Konersmann [Ko18, KG20], which defines mappings and transformations between model elements and model-representing code, taking non-generated code into account. With the MIC, modeling languages can be used to describe abstract design aspects, while code describes implementation details. However, the MIC focuses on models at design time only and does not consider runtime information and monitoring. We propose a mapping between UML state machines and Java that is suitable for logging purposes. The resulting code is made be embedded into a contextual code and allows for embedding implanted code.

Existing patterns for implementing state machines (e.g., [Ga96, Fo04, Go15, HJ17]) do not consider integrated logging on the model and code level. In Tab. 1 we give a high-level overview of our mapping. A detailed definition can be seen in [Eh19]. We represent states as classes, similarly to the *state pattern* from Gamma et al. [Ga96]. While the state pattern gives hints to structure the code template, it leaves multiple challenges. The pattern does not account for nesting or orthogonality, since it uses a much simpler meta model that has no concept of regions. We use packages in Java to represent regions, which helps us to group nested states. Further, the state pattern raises questions on where the transitions are defined.

Meta Model Element	Representation in Code
StateMachine	Class
Region	Package
Vertex (Abstract)	Only concrete subclasses are represented in code
State (Subclass of Vertex)	Class
FinalState (Subclass of State)	Class
Pseudostate (Subclass of Vertex)	Class
Transition	Method in source Vertex Class

Tab. 1: Table of the main elements of the meta model and their representation in code.

In our solution transitions are located in and handled by the source vertex. One condition to fire a transition is that the source vertex has to be active. Since we maintain the active vertices in a tree structure (state configuration) we can efficiently evaluate guards to check if a transition can fire. No check is required to test if a vertex is active.

Fig. 3 shows a visual representation of our code template for a transition. The left column shows the meta model for state machines, the center column shows the model and the right column shows the code. The solid boxes group elements belonging to a transition, the dashed boxes highlight the constraint (guard), the dotted boxes highlight the behavior (effect). Monitoring a guard is part of our running example.

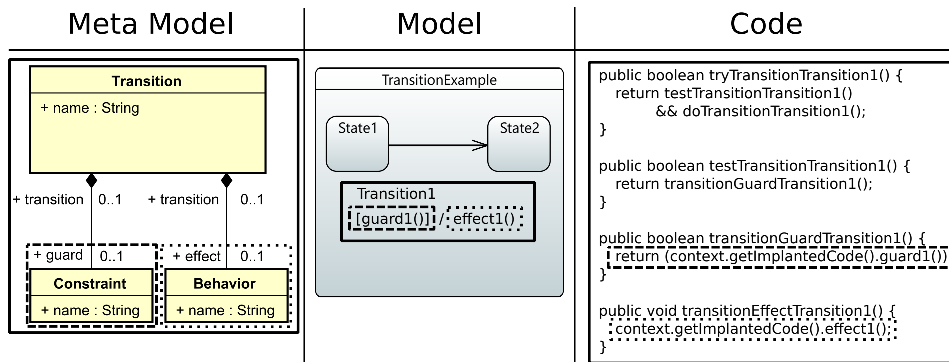

Fig. 3: Code template of the UML transition element.

3.2 Connecting code and monitoring

Analyzing the semantics of the meta model gives us insights into critical points in the state machine execution. For each meta model element, we created a list of events that cover these critical points. Tab. 2 shows an excerpt of the list of events. The first column shows the name of the event, the second column shows when the event is registered, and the last column shows the point in the code where the event is registered in form of an AspectJ [CCH04] notation in relation to the code representation in Tab. 1.

Monitoring Event	Position	Pointcut signature
TransitionCompletionEvent	after	public boolean Vertex+.doTransition*()
TransitionStartEvent	before	public boolean Vertex+.doTransition*()
TransitionEffectBehaviorFinishedEvent	after	public void Vertex+.transitionEffect*()
TransitionEffectBehaviorStartEvent	before	public void Vertex+.transitionEffect*()
TransitionGuardEvaluationEvent	around	public boolean Vertex+.evaluateGuard*()
TransitionTestedEvent	after	public boolean Vertex+.testTransition*()

Tab. 2: An excerpt of the captured events that concern transitions.

For each event, we create a description and a list of information that helps to understand the event. We distinguish the information by its origin. Runtime information is captured during runtime, while code, model, and meta model information can be obtained from the respective artifacts. In Fig. 4 we show the *TransitionGuardEvaluationEvent* as an example. The evaluation of the guard is of special interest for monitoring, since this information explains why a transition is executed or not. All monitoring events contain a timestamp of the event's occurrence and the current state configuration. Both are runtime information. Additionally, each event will contain information about the current location in the code and the method signature.

TransitionGuardEvaluationEvent
This event is generated when a transition guard is evaluated.
Attached Information
• **Runtime**: A timestamp of the event occurrence • **Runtime**: The current state configuration • **Runtime**: The boolean value of the guard • **Code**: The location in the source code where the monitoring event is generated • **Code**: The signature of the method where the monitoring event is generated • **Model**: The transition • **Model**: The source vertex of the transition • **Model**: The target vertex of the transition • **Meta-Model**: The type of the element

Fig. 4: An example of an event captured for monitoring, with a description of the event and attached information.

Using a systematic model-based approach to instrument the code has multiple benefits: The first observation is that this step does not require manual interaction with the source code. Rules can be defined for how the monitoring statements are inserted. When the code is automatically instrumented based on rules, the statements are generated in a uniform and systematic way. Defining these rules forces us to think about which events are relevant for monitoring.

4 Implementation

In this section, we describe how we implemented the tool prototype for our approach. The input of our tool is a UML state machine model. The outputs are instrumentation code and code that implements the business logic in Java. Both are woven together to get the instrumented code. During execution of the instrumented code, events are submitted to a monitoring system, which enriches the events with model information and allows us to query for specific information. We go on by describing each artifact and component in detail.

We use the Eclipse Papyrus Modeling environment[3] version 4.8.0 to create our models. This allows us to graphically model and save UML state machines. The models are saved using the XML Metadata Interchange (XMI) [OM15], a standardized XML interchange format widely used in the industry for exchanging UML models.

The code template is defined with Xtend[4] template expressions. Xtend is a general purpose programming language related to Java. Since we only use the template expressions of that language for code generation, we will not go further into detail about Xtend in general. When the code is generated, the code template will transform the previously defined state machines model into two artifacts. First, the business logic, which is a set of Java classes grouped in a package, that contains the the model code, including entry points the implanted code. And second, the instrumentation code is generated. The instrumentation code comprises AspectJ pointcuts and advices. The pointcut defines where an advice is applied.

List. 1 shows the pointcut (line 2-4) and advice (line 5-15) for the *TransitionGuardEvaluationEvent*. The purpose of the advice is to generate the event and pass it to the monitoring system. The first part of the advice (line 5-12) collects information that is common for all event types. After that (line 15) information specific to that event type is added, in this case, the return value of the guard evaluation. List. 2 shows the method in the business logic code that is matched by the pointcut shown in List. 1. When the method is executed the *TransitionGuardEvaluationEvent* is generated, populated with the desired information, and passed to the monitoring system.

5 Evaluation

We evaluated the applicability and functional correctness of our approach by creating a model of an artificial ATM system (see running example in Section 2) and applying the approach via the implemented tool prototype. We also evaluated the performance of the integrated monitoring in comparison to a non-monitored solution.

List. 3 shows the output of our monitoring system. This includes the information required to realize a unified view of the system that provides context information to monitoring

[3] https://www.eclipse.org/papyrus/

[4] https://www.eclipse.org/xtend/

```
// TransitionGuardEvaluationEvent
boolean around(sm.bankATM.interfaces.Vertex vertex) :
execution(public boolean sm.bankATM.regionTop.SelfTest.evaluateGuardT3())
&& target(vertex){
        monitoring.events.MonitoringEvent event = generateEvent(
        MonitoringEventTypes.TransitionGuardEvaluationEvent, //Event name (Event)
        System.currentTimeMillis(),             //Timestamp (Runtime)
        vertex.getContext().getStateMachine(),  //State configuration (Runtime)
        "_478fMLfSEeqxh_nJ0zRETQ",              //Model element id (Model)
        "T3",                                   //Model element name (Model)
        thisJoinPoint.getSourceLocation(),      //Source code location (Code)
        thisJoinPointStaticPart.getSignature()); //Method signature (Code)
        [...]
        //Return value (Runtime)
        event.getRuntimeInformation().put(RuntimeInformationKey.ReturnValue, returnValue);
        [...]
}
```

List. 1: Commented instrumentation code for a *TransitionGuardEvaluationEvent*.

```
public boolean evaluateGuardT3() {
        return (getContext().getImplantedCode().selfTestGuard());
}
```

List. 2: A code snippet representing the transition guard that will be matched by the *TransitionGuardEvaluationEvent*.

events. Fig. 5 shows the relationship between a logged event in the running example: the runtime information, the code that is executed, the model that is represented by the code, and the meta model that describes the model type. The arrows indicate how the event log has traces to each view of the system. The meta model information is used to resolve the model elements' type information.

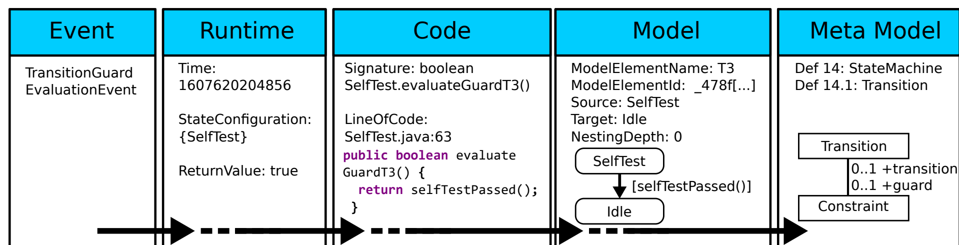

Fig. 5: Integrated view of model-based monitoring with linked views.

In our performance evaluation, we compared the execution times of multiple state machines without any monitoring; with collecting monitoring events, but without resolving model information; and with monitoring and resolving model information from the model. *Without monitoring* means that the code is not instrumented. This is done by removing the AspectJ

```
1  ----------------------------------------
2  [Event] TransitionGuardEvaluationEvent
3  ----------------------------------------
4  [Runtime] Time: 1607620204856
5  [Runtime] StateConfiguration: StateMachine: {SelfTest}
6  [Runtime] ReturnValue: true
7  [Code] Signature: boolean sm.bankATM.regionTop.SelfTest.evaluateGuardT3()
8  [Code] LineOfCode: SelfTest.java:63
9  [Model] ModelElementId: _478fMLfSEeqxh_nJ0zRETQ
10 [Model] Source: SelfTest
11 [Model] Target: Idle
12 [Model] ModelElementName: T3
13 [Model] NestingDepth: 0
14 [Meta Model] Type: org.eclipse.uml2.uml.Transition
```

List. 3: The monitoring output of the *TransitionGuardEvaluationEvent* of the Transition *T3*.

file. I.e., no monitoring statements are woven into the model representing code. *With monitoring* means, that the code is instrumented via AspectJ. I.e., monitoring statements are woven into the model representing code. *With monitoring and model information resolution* mean, that the code is instrumented and the model is used to resolve model information when the monitoring event is created. The model used for the performance test consists of an initial pseudostate, the displayed number states in between, and a final state. All the mentioned vertices are connected to the next vertex by a single transition.

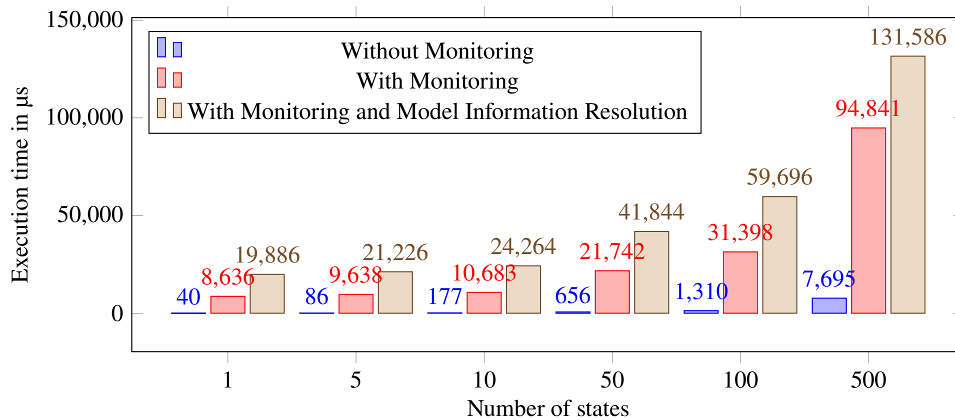

Fig. 6: The results of the performance test.

The performance test (see results in Fig. 6) shows that the monitoring system has a considerable performance impact. With monitoring turned on, the execution time is about 200 times higher for a state machine with one state and about 12 times higher for a state machine with 500 states. The infrastructure introduces a constant delay of about 8000 µs (see the difference without and with monitoring for one state). Resolving model

information introduces a performance overhead. This can be managed by resolving the model information only when needed. The monitoring system in the tool prototype was not optimized for performance. It was developed to show the advantages of model-based monitoring. Therefore, we suggest in future work to incorporate a monitoring framework that is optimized for performance.

6 Related Work

Monitoring of state machines has been subject to related work for many years. To the best of our knowledge, none provides an integrated view of runtime information on the model and code level. In this section we briefly discuss representatives of related approaches.

Balz [Ba11] describes the integration of non-hierarchical state machines with code, without focusing on the integration of monitoring and debugging. The MIC underlying our approach can be considered an evolution of Balz' work. Tan et al. [Ta08] describe an approach to analyze existing log files by deriving a state machine. This derived state machine is used to construct high-level analysis on the log data. Their approach shows that higher-level model information can be used to put log information into context, while it does not answer how state machines can be implemented or logged. Amar et al. [Am18] extract state machines from log files to spot differences in different versions or different deployment context of one piece of software. Jung et al. [JHS13] create an instrumentation aspect language (IAL) to specify monitoring probes based on model information. Using the IAL in our context could simplify the instrumentation. Heinrich et al. [He15] proposes an extensions to the iObserve [Ha13] monitoring and analysis approach, that integrates design time and run time views for cloud-based software. They focus on architectural models while our approach focuses on UML state machines. Bošković et al. [BH09] suggest with MoDePeMART a declarative specification of performance metrics in a domain specific modeling language. They focus on performance metrics while our approach focuses on explaining behavior. Using a domain specific modeling language to configure the instrumentation can facilitate adapting our approach because implementation details, like the use of AspectJ, can be hidden.

7 Conclusion

This research aimed to improve debugging and monitoring of state machines with the unique information only the model can deliver, to the benefit of model-based long-living software systems. As challenges, we identified that (C1) logging is usually not systematic for a project, and (C2) code-based log events describe observed behavior, but it is difficult to derive the intended behavior when the code was generated from a model and extended manually.

To overcome these challenges we developed a method for monitoring state machine executions of integrated state machine models following the model integration concept of

Konersmann [Ko18]. For this, we developed model/code mappings that are specifically suited for logging purposes, for embedding them into contextual code, and for embedding implanted code. These mappings were extended with systematic logging, which resolves code, model, and meta model information at the given program state. We evaluated our prototype using an artificial use case and a performance evaluation. The approach was applicable in the example use case. The performance is promising, yet requires optimization for productive use.

The use of systematic model/code mappings enables us to systematically weave instrumentation code into the system (C1). In contrast to traditional logging that is focused on either the code level or the model level, our approach presents the data in an integrated fashion. It therefore provides the runtime information of the model-based system in the context of a model and the code (C2). In future work we plan to extend monitoring and debugging views in IDEs like Eclipse with the now technically joint information and employ user studies to evaluate how the integrated views affect debugging processes.

Acknowledgements: The work presented in this paper is partially funded by the German Federal Ministry of Education and Research (BMBF) under the grant number 01IS19084D.

Bibliography

[Am18] Amar, Hen; Bao, Lingfeng; Busany, Nimrod; Lo, David; Maoz, Shahar: Using Finite-State Models for Log Differencing. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, p. 49–59, 2018.

[Ba11] Balz, Moritz: Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-based Software Development. PhD thesis, Universität Duisburg-Essen, May 2011.

[BH09] Bošković, Marko; Hasselbring, Wilhelm: Model Driven Performance Measurement and Assessment with MoDePeMART. In (Schürr, Andy; Selic, Bran, eds): Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 62–76, 2009.

[CCH04] Colyer, Adrian; Clement, Andy; Harley, George: Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. ADDISON WESLEY PUB CO INC, 2004.

[Eh19] Ehl, Marco: Model-based Monitoring of Integrated State Machines. Master's thesis, University of Koblenz and Landau, Germany, 2019.

[Fo04] Fowler, Martin: UML Distilled: a Brief Guide to the Standard Object Modeling Language. Addison-Wesley, 3rd edition, 2004.

[Ga96] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, 6th edition, 1996.

[Go15] Gonzalez Moctezuma, L. E.; Ferrer, B. R.; Xu, X.; Lobov, A.; Martinez Lastra, J. L.: Knowledge-driven finite-state machines. Study case in monitoring industrial equipment. In: 2015 IEEE 13th International Conference on Industrial Informatics (INDIN). pp. 1056–1062, 2015.

[Ha87] Harel, David: Statecharts: a visual formalism for complex systems. Science of Computer Programming, 8(3):231–274, 1987.

[Ha13] Hasselbring, Wilhelm; Heinrich, Robert; Jung, Reiner; Metzger, Andreas; Pohl, Klaus; Reussner, Ralf; Schmieders, Eric: iObserve: Integrated Observation and Modeling Techniques to Support Adaptation and Evolution of Software Systems. Forschungsbericht, Christian-Albrechts-Universität Kiel, Kiel, Germany, Oktober 2013.

[He15] Heinrich, Robert; Jung, Reiner; Schmieders, Eric; Metzger, Andreas; Hasselbring, Wilhelm; Reussner, Ralf; Pohl, Klaus: Architectural run-time models for operator-in-the-loop adaptation of cloud applications. In: 2015 IEEE 9th International Symposium on the Maintenance and Evolution of Service-Oriented Systems and Cloud-Based Environments, MESOCA 2015 - Proceedings. IEEE, pp. 36–40, 2015.

[HJ17] Havelund, Klaus; Joshi, Rajeev: Modeling and Monitoring of Hierarchical State Machines in Scala. In (Romanovsky, Alexander; Troubitsyna, Elena A., eds): Software Engineering for Resilient Systems. Springer International Publishing, Cham, pp. 21–36, 2017.

[JHS13] Jung, Reiner; Heinrich, Robert; Schmieders, Eric: Model-driven Instrumentation with Kieker and Palladio to forecast Dynamic Applications. In: Proceedings Symposium on Software Performance: Joint Kieker/Palladio Days 2013 (KPDAYS 2013). volume 1083 of CEUR Workshop Proceedings. CEUR, pp. 99–108, November 2013.

[KG20] Konersmann, Marco; Goedicke, Michael: Same but Different: Consistently Developing and Evolving Software Architecture Models and their Implementation. In (Felderer, Michael; Hasselbring, Wilhelm; Koziolek, Heiko; Matthes, Florian; Prechelt, Lutz; Reussner, Ralf; Rumpe, Bernhard; Schaefer, Ina, eds): Ernst Denert Award for Software Engineering 2019 - Practice meets Foundations. Springer International Publishing, 2020.

[Ko18] Konersmann, Marco: Explicitly Integrated Architecture - An Approach for Integrating Software Architecture Model Information with Program Code. PhD thesis, 2018.

[OM15] OMG XML Metadata Interchange (XMI) Specification, Version 2.5.1, 2015.

[OM17] OMG Unified Modeling Language Specification, Version 2.5.1, 2017.

[Ta08] Tan, Jiaqi; Pan, Xinghao; Kavulya, Soila; Gandhi, Rajeev; Narasimhan, Priya: SALSA: Analyzing Logs as State Machines. In: Proceedings of the First USENIX Conference on Analysis of System Logs. WASL'08, USENIX Association, USA, p. 6, 2008.

[vHWH12] van Hoorn, André; Waller, Jan; Hasselbring, Wilhelm: Kieker: a framework for application performance monitoring and dynamic software analysis. In (Kaeli, David R.; Rolia, Jerry; John, Lizy K.; Krishnamurthy, Diwakar, eds): Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12, Boston, MA, USA - April 22 - 25, 2012. ACM, pp. 247–248, 2012.

[Vö06] Völter, Markus; Stahl, Thomas; Bettin, Jorn; Haase, Arno; Helsen, Simon: Model-Driven Software Development - Technology, Engineering, Management. John Wiley & Sons, New York, 2006.