Rapidly Locating and Understanding Errors using Runtime Monitoring of Architecture-Carrying Code

Marco Konersmann
paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen
Essen, Germany
marco.konersmann@paluno.uni-due.de

ABSTRACT

Finding and understanding errors in software that is deployed in the field and tested by beta testers or used in production is a difficult and often time consuming task. If the feedback to the developers is naïvely composed of log messages, stack traces, and informal user feedback, it might not contain enough information to locate the erroneous fragments efficiently. It is thus slowing the dynamics in rapid continuous software engineering scenarios. These errors can more easily be found when the code is explicitly designed to carry architectural specifications by intertwining structure and behavior models with the rest of the code. With architecture-carrying code, (1) the steps leading to single errors can be replayed for understanding and debugging; (2) statistical information about errors can be related to structure elements and behavior branches. In this paper we present our idea of architecture-carrying code and its application for error understanding to support rapid continuous software engineering.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software Management, Software Development*

General Terms

Design

Keywords

Runtime Monitoring, Rapid Continuous Software Engineering, Architecture-Carrying Code

1. MOTIVATION

Time-to-market is a relevant topic in software engineering. As software is often a critical success factor for businesses, the software is better ready earlier than later. That said, every non-trivial software has a high probability to contain bugs. One strategy to find bugs in beta state software is to deploy it, and let beta testers use it in the wild to get error reports[1].

For analyzing the results of such tests, statistical data about the fault frequency related to structural and behavioral parts of the software can be helpful, as well as detailed information about single instances of incorrect behavior.

Typical error reporting strategies comprise semi-structured textual logs and corresponding log analyzer tools[2]. Such log files often include information for localizing the source of the failure, e.g. by stating the name of the failed operation and the name of an exception. More tricky faults, that do not result in exceptions but false results are harder to find, and require intensive logging. The effort for extracting single instances of incorrect behavior from such log files depends heavily on the appropriateness of the log messages. This can result in a high effort, and thus be slow and expensive. That is unwished, especially in rapid continuous software engineering (RCSE) scenarios. Additionally, it is hard to extract meaningful statistical data from these log files.

For a better analyzability of such scenarios, we propose a technique for systematically storing model-based usage scenarios. The data is available for replaying the scenario, as well as providing a good basis for visualizing statistical data.

2. FOUNDATIONS

2.1 Architecture-Carrying Code

The technique is based on architecture-carrying code [4]. The idea of architecture-carrying code is to represent architectural models in source code with sophisticated code structures. These code structures are not directly changed with source code editors, but with model editors. These model editors allow to edit the architecture in a representation that software architects are comfortable with, e.g. UML or formal specification languages. The editor extracts the architecture in the underlying code base and presents a model to interact with. Changes to the model are reflected by changes to the underlying code base. The model view is volatile. It only exists as long as the model editor is in use. With architecture-carrying code, the architecture model is

RCoSE'14, June 3, 2014, Hyderabad, India
ACM 978-1-4503-2856-2/14/06
http://dx.doi.org/10.1145/2593812.2593814

[1] e.g. Blackberry Beta Zone (www.blackberry.com/beta), Swype Beta For Android (http://beta.swype.com/)

[2] for example Graylog2 (http://graylog2.org/) or logstash (http://logstash.net/)

available at compile time as source code structures and at run time via reflection mechanisms.

2.2 State Machines

The definition of state machines used in this paper are based on the definition by Balz in [1]. The definition, as well as the implementation of the state machines, is simplified for this paper, because not all features are necessary for this example.

A state machine consists of states, variables, and transitions between states. One state is the initial state. Variables are typed and have initial values. Transitions may have zero or more action labels that describe the activities that happens between these states. These activities change the variables. The actual change of variables is not part of the model. Transitions also have guards. Guards are propositional expressions over the variables. A transition can only be fired if its guard evaluates to true.

2.3 OSGi Services

Our example is based on the use of OSGi[3]. OSGi is an industry-accepted java modularization framework. OSGi includes a so-called service layer, which allows to create components that offer and require services. Service components in this context are named java objects that provide or require java interfaces. They can be dynamically started and stopped at run time. Services in this context are the java interfaces that are provided or required by service components. The type of a service is defined by its interface. Required or provided services are registered in a central registry within the runtime environment. A service type can be provided or required by multiple service components. When a service component requires a service type, it is informed about registered service instances of that type using callback methods. This technology can be used to build a component-based system. In this paper we use OSGi service components to represent components, their registered services as provided interfaces, and their required services as required interfaces. Component types define which interfaces are provided and required. Multiple instances of one component type exist, when multiple components provide and require the same services.

3. RUNNING EXAMPLE

Figure 1 shows the most abstract level of the architecture description of a fictional massively multiplayer online game (MMOG) designed for Android smartphones, called "Doing Good and Evil Things (DoGET)" which will serve as a running example in this paper. DoGET is a game in which multiple players can perform quests in a common virtual world with their heroes. The figure shows the static structure of the system. The system's architectural structure is modeled in UML.

On this level the system consists of seven components. Please note that in this paper we use the term "component" to describe a component type that is instantiated exactly once if not stated otherwise. The components *Input*, *Sound*, and *Rendering* handle input and output to the user. *NetworkMgmt* provides means to synchronize the common information with other players, including the state of the common quest. The component *WorldMgmt* handles the position and

[3]http://www.osgi.org

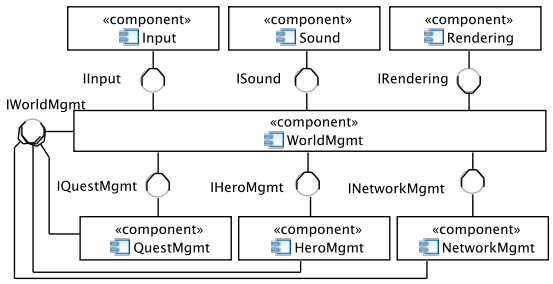


Figure 1: The highest level of the architectural static structure of the massively multiplayer online game DoGET

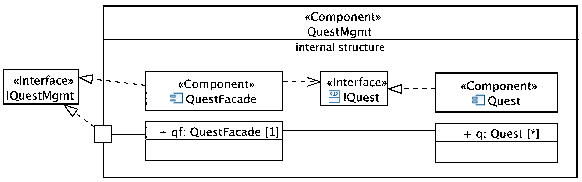


Figure 2: The inner architectural structure of the QuestMgmt component

actions of all entities in the game. *HeroMgmt* has operations to manage the state of the player's hero.

In this paper we will focus on the component *QuestMgmt*, which manages multiple quests. Figure 2 shows the inner architecture of the *QuestMgmt* component. This component comprises the component *QuestFacade* and multiple *Quest* components. The *QuestFacade* requires zero or more implementations of the interface *IQuest*. The component type *Quest* is defined to provide the interface *IQuest*. Several instances of the component type *Quest* can exist in parallel. A *Quest* instance defines a behavior in terms of a state machine, as they are defined in section 2.2. Figure 3 shows the state machine of the *Quest* instance that will be our example. The example program is implemented with the OSGi framework, which is available for the Android operating systems for mobile devices.

4. RUN TIME MONITORING

For systematically storing model-based usage scenarios, we extend the use of architecture-carrying code [4]. First, we create a model of the software architecture. The architecture model comprises static and dynamic structure, as well as static and dynamic behavior. Due to the approach of architecture-carrying code, the architecture model is available at compile time as source code structures and at run time via reflection mechanisms.

In our approach, we store the history of behavior models and dynamic structure models. When errors occur, the history can be tagged as erroneous. In the following, we show how the architecture is designed in terms of architecture-carrying code, and how the dynamic behavior and the dynamic structure of this system is stored. This information can be used to locate and understand problems on the architectural level. We will present this technique and its implications using the running example.

One key aspect for architecture-carrying code is the definition of source code structures for architectural concepts.

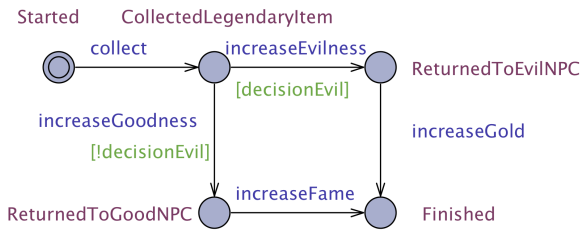


Figure 3: The state machine for the example quest

Thus in this example we have to define source code structures for structure and behaviour model elements.

4.1 Structure Modeling

The system's architecture is modeled in UML. For making these structure models available in terms of architecture-carrying code, we have to define source code structures for (1) components, (2) their interconnection, and (3) interfaces. The source code structures presented here are specifically designed for the execution framework in use: OSGi.

The source code structure of a component comprises a Java class and a component descriptor. The class name reflects the component name. The component descriptor is an XML file, including the name, and the implementation class name. Figure 4 shows the representation of the QuestFacade component in UML in the upper part, and the representation as Java class the the corresponding OSGi service component descriptor in the lower part.

An interface with operations in UML is represented as a Java interface with the corresponding Java methods. A component can provide interfaces. In UML this is represented with an *Realization* edge to an interface. In the source code the components class implements the corresponding interface class and the corresponding methods. The body of these methods is irrelevant for the architecture and thus not part of the source code structure. Additionally, OSGi requires the component descriptor to state the provisions. Figure 5 shows these representations. For simplicity reasons the methods are not shown in this figure.

A component requiring interfaces is modeled in UML using the *Usage* edge from a component to an interface. Required interfaces are defined in the source code using an attribute, a setter, and an unsetter method. The concrete representation of the relationship depends on the cardinality. For *1..1* relationships, a simple attribute is set and unset in the corresponding methods. In *1..n* and *n..m* relationships, references to the bound instances are added and removed from a collection. OSGi also requires a corresponding component descriptor. Figure 6 shows these representations. The cardinality is not shown in the UML diagram.

Connections between required and provided interfaces are automatically managed by the OSGi runtime. Thus the source code structures described above also include the binding of component instances to the required interfaces.

4.2 Storing Structure History

The structure of our example system is dynamic. Quests can be registered and unregistered at runtime. The history of this coming and going of service components is stored in our approach. Conceptually, this is performed by an aspect, that executes when a service component is registered, unregistered, or modified in the framework. These events


```
public class QuestFacade { }
```

```
<scr:component
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    name="QuestFacade">
  <implementation class="QuestFacade"/>
</scr:component>
```

Figure 4: The representation of the QuestFacade component as UML, and as Java source code with the corresponding OSGi component descriptor


```
public class ExampleQuest implements IQuest { }
```

```
<scr:component
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    name="ExampleQuest">
  <implementation class="ExampleQuest"/>
  <service><provide interface="IQuest"/></service>
</scr:component>
```

Figure 5: The representation of provided interfaces as UML, and as Java source code with the corresponding OSGi component descriptor. For simplicity reasons, the signatures of the implemented methods is not shown.

are stored in sequence with a timestamp. Technically, this is performed by an OSGi *EventListenerHook* that is informed about such events by the OSGi framework.

4.3 Behavior Modeling

For developing architecture-carrying code in terms of state machines as defined in section 2.2, we need to define source code structures for states, transitions, variables, guards, and action labels. For this we use the structures that have already been defined by Balz in [1].

Balz defines a state as a Java class implementing an interface *IState*. That interface does not define any operation, but is just for marking a class as a state definition. The name of the state is defined by the class name.

A state machine contains variables that are queried by guards and manipulated by actions when transitions are executed. These variables are represented as get (for queries) and set (for manipulation) methods in a java class (the *variable class*). An instance of the state machine needs an instance of this class to query and manipulate actual variables during its execution.

The source code structure of a transition is an operation within a state class, that has a *Transition* annotation. Transition operations have one parameter, the actor. The actor is a reference to the underlying source code that is not part of the behavior model. I.e. these details of the behavior is irrelevant on the architecture level. At run time an instance of the actor class is provided by an execution framework. The action labels that are associated with transitions are represented by method calls to this actor in the transition method body. The actor has a reference to the variable class instance that is managed by the state machine. The execution of these methods manipulate the variables in the state machine using the set methods of the variable class.


```
<scr:component
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    name="IQuest">
  <implementation class="IQuest"/>
  <reference bind="setQuest" unbind="unsetQuest"
    cardinality="0..n" interface="IQuest"
    name="IQuest" policy="dynamic"/>
</scr:component>
```

```
public class QuestFacade {
  LinkedList<IQuest> quests = new LinkedList<IQuest>();

  public synchronized void setQuest(IQuest quest) {
    quests.add(quest);
  }

  public synchronized void unsetQuest(IQuest quest){
    quests.remove(quest);
  }
}
```

Figure 6: The representation of required interfaces as UML, and as Java source code with the corresponding OSGi component descriptor. The cardinality is not shown in this UML diagram.

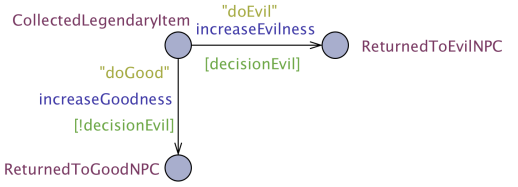
The *Transition* annotation has two parameters: a target state class, and a contract class. The target state class is the state that follows the current state when the transition was executed. The contract class contains the guard. It implements the interface *IContract*. The IContract interface takes a variable class as class parameter, and provides the methods *checkCondition* and *validate*. The method *checkCondition* represents the guard of the transition, and returns whether the guard evaluates to true. For this, it uses calls to the variable class instance. The method *validate* compares the variables before and after the transition execution. The validate method exists to ensure at run time that the method calls representing the action labels updated the state machine's variables as expected.

Figure 7 shows the source code representation of the state *CollectedLegendaryItem*, and one referenced contract class *ReturnToEvilContract*. The class *Heroes* is at the same time the variable class and the actor class.

The operations within the state class are the transitions. In our example (cf. figure 3) the state *LegendaryItemCollected* is the source for two transitions, one named *doEvil*, and one named *doGood*. The first transition has a guard *[decisionEvil]*. The guard is reflected in a call to the actors operation *getDecisionEvil* in the contract in the lower part of figure 7. The action *increaseEvilness* increases the evilness of the heroes. This is reflected in the operation call to the actor's *increaseEvilness* operation. The second transition has an action *increaseGoodness*, which increases the goodness of the heroes, and a guard *[!decisionEvil]* (the latter is not shown in the figures). The detailed behavior, how the evilness or goodness of the heroes is increased is considered irrelevant for the architecture.

4.4 Storing Behaviour History

During the beta tests or at production time, we store the history of the dynamic behavior. To achieve this, the source code structure for the transition operations is extended to store information about the fired transition. Aspect oriented programming [3] is used for not having to change the static behavior definition in the source code.




```
public class CollectedLegendaryItem implements IState {
  @Transition(target = ReturnedToEvil.class,
              contract = ReturnToEvilContract.class)
  public void doEvil(Heroes actor) {
    actor.increaseEvilness();
  }

  @Transition(target = ReturnedToGood.class,
              contract = ReturnToGoodContract.class)
  public void doGood(Heroes actor) {
    actor.increaseGoodness();
  }
}
public class ReturnToEvilContract
    implements IContract<Heroes> {
  public boolean checkCondition(Heroes vars) {
    return vars.getDecisionEvil();
  }

  public boolean validate(Heroes before, Heroes after) {
    // Shortened for readability
    return [...];
  }
}
```

Figure 7: The source code structure for the state *LegendaryItemCollected* with a guard.

The aspect is executed around each transition method. It first stores the current variables and their values using the variable class instance by identifying and calling its get methods. It then executes the transition method. Afterwards the new values of the variables are collected. Both variable states are stored together with the executed transition and a time stamp.

4.5 Error Dashboard

Using our approach, the structural and behavioral history are stored at run time. Using an error reporting mechanism, this history can be transmitted — enriched with an informal error description — to the developers to provide feedback. The history transmitted to the developers can be loaded into an analyzing tool. The tool uses the structure and behavior history data to replay the scenarios that lead to errors.

Our idea is that an initial view is the static structure of the system. This structure can be extracted from the source code, because it is encoded in the predefined source code structures. The architectural structure can be represented as a UML component diagram. The behavior can be represented by state machine diagrams.

A single set of history data, enriched with error information, allows the erroneous behavior to be precisely located in the dynamic behavior, because it is related to a transition in a state machine. The state machine in turn is related to a dynamic structure element. The dynamic structure element has a static representative. The dashboard can show the location in this hierarchy, following Shneiderman's visualization mantra [7]: overview, zoom and filter, details on demand. In the static structure the error can be located as an overview, by color coding erroneous components. For big architectures, zooming and filtering can make sense, by

zooming into hierarchical architectures, and filtering erroneous components. Details on demand are provided when the perspective is changed from the static to the dynamic architecture. When the history data is loaded, the developer can step through the fired transitions and structural changes. This is possible because the history is stored with discrete time information.

When multiple histories are loaded into the dashboard, a heatmap can be shown in all parts of the dashboard. The static structure can show on various levels where the most errors or, depending on the error meta data, where the most severe errors happen. The dynamic structure can give further details for multi-instance components. The static behavior can show behavior branches with high error rates.

5. RESULTS

The technique shown in this paper can be used in the beta testing and the production phase for processing testers' and users' feedback. Single history sets can be used for debugging. The replay mechanism allows an understanding of the system's dynamics on the architectural level when the error occurred. The dashboard can show the state of the program when the error occurred, and developers can jump through the states the program went through, before the error happened.

In contrast to typical debugging tools, with our approach the software can be analyzed a posteriori, when the error actually happened. When software is debugged using current tools, it is sometimes hard to create the environment that led to the error. Thus the history can ease the error understanding. This is important for RCSE, because a faster understanding allows faster fixes, and thus shorter release cycles.

Greater sets of history can be used for creating heat maps, that show fault frequency related to structural and behavioral parts. This can be used as a basis for planning error fixing resources, or for developing an error fixing road map. This overview can also be a benefit for RCSE scenarios, because it allows to see where the most errors happen.

6. TOOLING

The tooling for the idea presented in this paper is not fully implemented yet. The source code and run time representation of static behavior models in terms of state machines in java source code has already been implemented [1]. An execution environment and a code editor that shows the code as a state machine diagram and allows for arbitrary editing are already available and are used in productive applications in our research group. For the approach presented in this paper, we extended this technology with the collection of history information (cf. section 4.4).

An editor for showing OSGi structures using UML does not exist yet. We are currently defining source code structures and developing editors for a variety of component frameworks like OSGi. However, a starting set of source code structures for OSGi has already been developed (cf. section 4.1). For this set of structures, a history collection mechanism has already been implemented (cf. section 4.2). The major next steps of tool development is to complete the structure editor and to develop the dashboard application.

7. DISCUSSION

The technique we propose has implications to the projects that use them. First of all, the architectural source code of the application is not completely in the hand of the programmers, but has to follow certain structures. While this seems to be a considerable constraint to some programmers, we believe that in systematic software engineering projects, most of the code already follows very specific structures. When software is developed based on an execution framework, like OSGi, the code has to follow its rules to be executable. The specified architecture can hardly be developed in another way than the one our source code structures enforce, if the framework is used. Deviance from such structures often result in code that is worse to maintain, because the same concept within an application (e.g. a component) has different types of implementation styles. The same holds for the behavior models. When the behavior is specified by a state machine, the code should systematically follow specific structures for implementing this state machine. The approach used in this paper explicitly states these structures. We expect the code structures to rarely be changed manually in code, because it can be opened and fully edited by opening it in a diagram view.

Our approach is focused on the architectural level. Software is often specified on this level in some notation, e.g. in box-and-arrow charts or UML. Sometimes behavior is specified in a formal notation, e.g. as state machines. On lower levels of the system such specifications are rare for many applications. This is especially true for mobile applications. According to our experience in this domain these are often parts of the system that can be developed without formal behavior models. At these levels, we think the programming language is sufficient for understanding the structure and the behavior, so that our approach is not helpful there.

Currently the approach supports two ways of describing formal behavior: state machines and process models [1]. Other behavior model types are in use. The requirements for developing source code structures and run time environments for such models is described in [1]. More model types can be used if the corresponding source code structures and runtime environments are developed.

With our approach errors on the architectural level can be located and debugged on the abstraction level of the architectural model instead of the level of programming languages. This probably eases the understanding of errors, as long as the errors are on the architectural level. Error location and debugging on a lower level is explicitly not addressed in our approach. However, our approach might still be helpful in this case, because it requires the code to be relatively systematically structured, and assists the developers in doing so.

Our approach is based on the idea to store enough information for efficient error location and understanding. The latter implies an overhead in terms of time and storage demand. The main overhead is the stored history data, especially for behavior models, because each transition execution is stored with all variable values before and after the transition, and the execution comprises method calls to the guard, for the actions, and for validation. With our approach we aim at a high abstraction level of the system specification. Due to this abstraction level, the history is effectively traceable and practical to be stored. However, this is strongly depends on the actual system, and what is understood as the *architecture level* in that system. The main driver for storage overhead is

the granularity of transitions and states, and the size of the stored variables. The overhead of the structure history seems to be negligible. We are planning to validate the overhead in a systematic study.

8. RELATED WORK

Other work that is related to our approach can be found in different directions. Architecture dashboards show the runtime architecture of systems. Monitoring frameworks and log analyzers generate events during the system runtime that can be monitored and analyzed. Work related to architecture-carrying-code is found in the communities of runtime models and model-driven development.

Runtime models of software architecture are used often in the context of self-adaptive and self-healing systems (e.g. [8]). These approaches are not designed to help locating and identifying runtime errors, but to automatically manipulate the architecture based on predefined rules. As such, the do not explicitly target RCSE scenarios.

Software monitoring frameworks (e.g. Kieker [9]) generate detailed information about the software at run time. This is of benefit for detailed debugging and for an overview about the general health of the software. These frameworks are targeted at detailed information, and typically do not bridge the semantic gap between the running system and its specified architecture.

Managing multiple representations of software design and specifically architecture has been subject to other fields of research. Related to the paper at hand is the field of Model-Driven Software Development (MDSD) (e.g. [2]), model execution (e.g. [5]), and round trip engineering (e.g. [6]).

MDSD concentrates on deriving code from models. The models and the code are two representations of the architecture that are independently subject to evolution and maintenance. Changes in the specification can be taken over automatically in the implementation. When the architecture changes in the implementation, these changes cannot be automatically taken over in the specification.

Model execution reduces the representations to the models only. The specifying model is enriched with clear semantics. Thus the models can be executed. These models are typically interpreted and thus have probably a weaker performance than our approach. However this is still subject to a systematic study.

Round trip engineering (RTE) describes techniques to synchronize models and code. The models used in RTE are very detailed and technical, e.g. UML class diagrams. RTE thus allows for two-way synchronization, but does not bridge the gap between abstraction levels.

9. SUMMARY

In this paper we presented our idea for rapidly locating and understanding errors using runtime monitoring of architecture-carrying code. Our idea is to shorten release cycles in RCSE scenarios, by providing a technique to easier locate and understand errors that come up while beta testing or in production. By shortening the time to locate and understand an error, systems can be more rapidly fixed. We also provide a dashboard for the developers to visualize the hot spots in the program in terms of error frequencies.

The technique we present is based on the approach of architecture-carrying code. In this approach, the source code has to follow certain structures, so that architectural information, that is usually only implicitly available in source code, is explicitly available at compile time and at run time.

Ongoing and future work is to finish the implementation of the prototype. When the prototype is available, we plan to evaluate the performance of the approach.

When the live presentation of remote structure and behaviour is working, we consider extending the technique to remotely change the behavior or structure of the software running on mobile devices. This way, we could provide short-term help for users that ran into malicious systems states. When the user is happy, in the background the root cause can be fixed, and the patch uploaded to all users at run time.

10. REFERENCES

[1] M. Balz. *Embedding Model Specifications in Object-Oriented Program Code – A Bottom-Up Approach for Model-Based Software Development*. PhD thesis, Univeristy of Duisburg-Essen, 2011.

[2] A. Brown, J. Conallen, and D. Tropeano. Introduction: Models, Modeling, and Model-Driven Architecture (MDA) Model-Driven Software Development. In S. Beydeda, M. Book, and V. Gruhn, editors, *Model-Driven Software Development*, chapter 1. Springer, Berlin/Heidelberg, 2005.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[4] M. Konersmann and M. Goedicke. A Conceptual Framework and Experimental Workbench for Architectures. In M. Heisel, editor, *Software Service and Application Engineering*, volume 7365 of *Lecture Notes in Computer Science*, pages 36–52. Springer Berlin Heidelberg, 2012.

[5] M. P. Luz and A. R. da Silva. Executing UML Models. In *3rd Workshop in Software Model Engineering (WiSME 2004)*, 2004.

[6] U. A. Nickel, J. Niere, J. P. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In *Proc of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*, 2000.

[7] B. Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 336–343, Sep 1996.

[8] S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management: The self-repair case. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 101–110, New York, NY, USA, 2008. ACM.

[9] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, April 2012.