Proceedings of the 2nd Workshop on Self-Managed Systems, co-located with FSE 2004, Newport Beach, CA, November 2004

Architecture Differencing for Self Management

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
+1 310 578 5350
aegyed@acm.org

ABSTRACT
Traditionally, software models are associated with development and maintenance related activities. This paper demonstrates that models also serve a vital purpose in supporting the self awareness and management of software systems in their deployed environments. We use probes to observe the executing software system by extracting the outside stimuli the system is subjected to. We use this information to concurrently simulate the model-behavior of the software system. The state of the simulating system then mirrors the state of the executing system. In this setting, the simulation serves as a foundation to self awareness through which differences among simulated behavior and real behavior are investigated. The simulation also serves as a guide to self management (i.e., self healing, self configuration) where the system uses additional information provided in the simulating model to manage itself.

Categories and Subject Descriptors
I.6 [**Simulation and Modeling**].

Keywords
Architecture differencing, simulation, and reflection.

1. INTRODUCTION
Software models have gained widespread acceptance due to their ability to support many aspects of software development and maintenance (e.g., Unified Modeling Language [12]). Yet models are rarely associated with software deployment. Indeed, we perceive it as atypically to deploy models with their software systems. But why shouldn't we?

Models are a reflection of the software system. They are generally easier to understand and easier to analyze than the system. Yet when software systems are deployed in their environment, we do not leverage from this benefit. This paper describes the use of model simulation to mirror the execution of the software system in the deployed environment.

Simulation is the ability to mimic the execution of a software system. Typically, simulation will not mimic every detail of the system but instead focus on some technical aspect of relevance. This paper investigates how software understanding benefits from simulation and how the system can piggyback from the simulation to support its own self-awareness and self management.

WOSS'04 Oct 31-Nov 1, 2004 Newport Beach, CA, USA

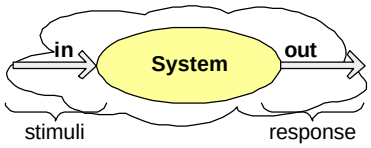

Figure 1. The Stimuli and Responses of Software Systems

In the following, we presume that the software system is a predominantly reactive system that responds to external stimuli. Most applications built today are reactive systems that consume external stimuli to trigger internal activities. For instance, applications with graphical user interfaces are reactive systems that consume mouse clicks or keyboard strokes. Reactive systems may be understood in terms of the stimuli they accept and the responses they generate (Figure 1).

We present an approach to architecture differencing for self-awareness and self-management. We need a simulatable model that can be used to mirror the deployed system. Real stimuli (obtained by instrumenting the actual system and abstracting and filtering the information) are supplied to guide the simulation. The model simulation is then a reflection of the state and behavior of the executing system and they are both expected to respond equivalently given the same input stimuli. If they do not respond equivalently then the model behavior differs from the system behavior. During design time, this knowledge is used to reason about the consistency between the model and system. Once the "correctness" of the model is established then our approach is deployed with the software system to determine disallowed behavioral differences between the model and the system in the deployed environment. Since the model is then presumed correct and complete, a difference indicates abnormal system behavior caused by fault, attack, or improper use. Moreover, once a difference between system and model is detected, a self healing/management mechanism may use the model as a guide to elicit what should have been the correct behavior and then use the discrepancy between system behavior and model behavior to reason about a proper response (i.e., healing, re-configuration).

Section 2 discusses model reflection to mirror the software system with a simulating model. Section 3 discusses model differencing to identify inconsistencies between the responses of the system and the model. Section 4 discusses the use of reflection and differencing for self awareness and self management.

2. VIDEO-ON-DEMAND CASE STUDY
We demonstrate our approach on a video-on-demand player (VOD) developed by [3]. The VOD player is a movie player that searches for movies on a server, downloads them, and plays them.

The "on-demand" feature of the player allows the playing of a movie concurrently while downloading its data from a remote site. The VOD system was modeled in SDSL [4] which is a statechart-like language [6,9] we developed previously (details of the language are omitted here for brevity).

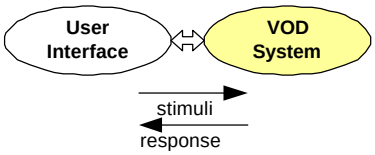

Figure 2. VOD Player (system) interacting with UI

The VOD Player receives external stimuli through the user interface and it responds by displaying movies to the user via the same user interface (UI) – see Figure 2.

3. REFLECTION

The VOD Player, like most systems, is reactive by acting and reacting solely on external stimuli (i.e., user input). If the model correctly describes the software system then both, the player's execution (real system) and the model's simulation, behave equivalently if they receive the same external stimuli (user input) at the same time. In other words, both generate the same responses to the same input. We differentiate between model reflection and system reflection where the former captures the behavior of the model simulation and the latter captures the behavior of the system execution (see Figure 3).

Some simulators assume a "closed world" where the simulated stimuli must be pre-defined. This is impractical and we thus rely on "open simulation" where the stimuli are provided externally during simulation. The ideal source of the simulated stimuli is the real stimuli itself but how do we observe these stimuli?

We use instrumentation to intercept the stimuli of the executing system. We previously developed wrapper technology [1] to monitor any system's interaction with the outside world [3]. We consider this and other existing technologies to be "probes" that are deployed within the executing system to monitor it. However, the system's stimuli are typically in a lower level of granularity and abstraction than needed for the simulation. Fortunately, it is much easier to abstract low-level, system stimuli into high level, model stimuli than the other way around because the abstraction is usually a simplification process. We thus filter, aggregate, and otherwise translate real stimuli into simulated stimuli.

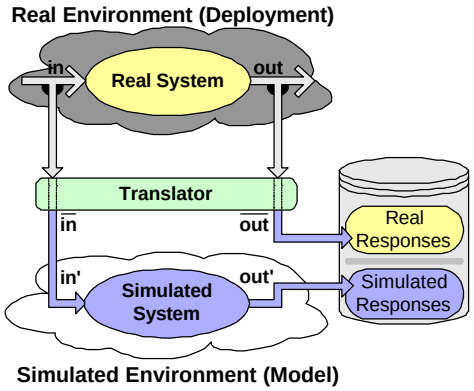

Figure 3. Model Reflection through Probes and Translation

The executed system is thus executed within its environment where it is subjected to external stimuli (i.e., user input). Through probes and translators, the system's stimuli are instantly forwarded to the simulation and its simulating model. The simulating model is thus given the same input the executing system receives. As a consequence, the simulating model mirrors the behavior of the executing system in that their states and responses are similar (we discuss later why their responses are allowed to differ). In itself, this is a useful capability for the understanding of a deployed software system because a human observer can use the simulation as a "window into the system" to better observe the not-so-observable system.

Based on the same instrumentation technology, we also intercept and translate the responses of the real system. Figure 4 (left) depicts the observed stimuli and responses of the real VOD player (note: we use UML sequence diagrams here [9]). In this scenario,

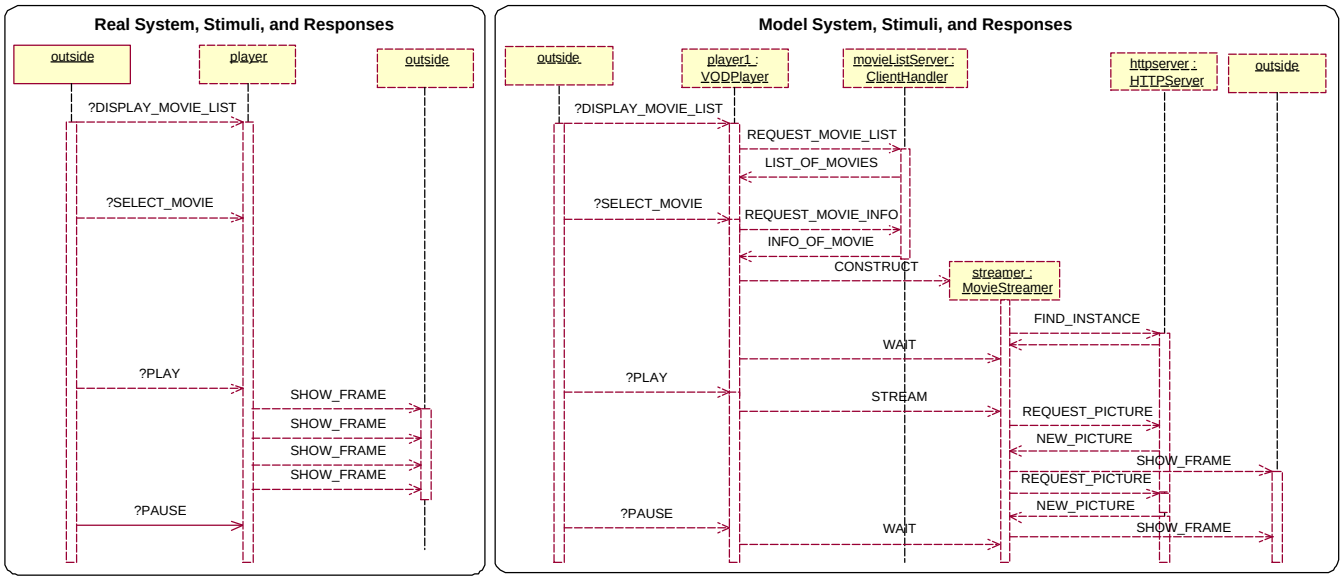

Figure 4. Real System, Stimuli, and Responses (left) and Model System, Stimuli, and Responses (right)

the user initially asks for a list of all movies, then selects a movie from the list, plays it, and pauses it. Figure 4 (left) depicts these stimuli and the corresponding responses after translation. For example, the user click on the button "Movies" is translated into the event "?DISPLAY_MOVIE_LIS T. Figure 4 (left) also depicts the response of the player in that only after successful movie selection the play button will result in periodic "SHOW_FRAME" events until the pause button is pressed.

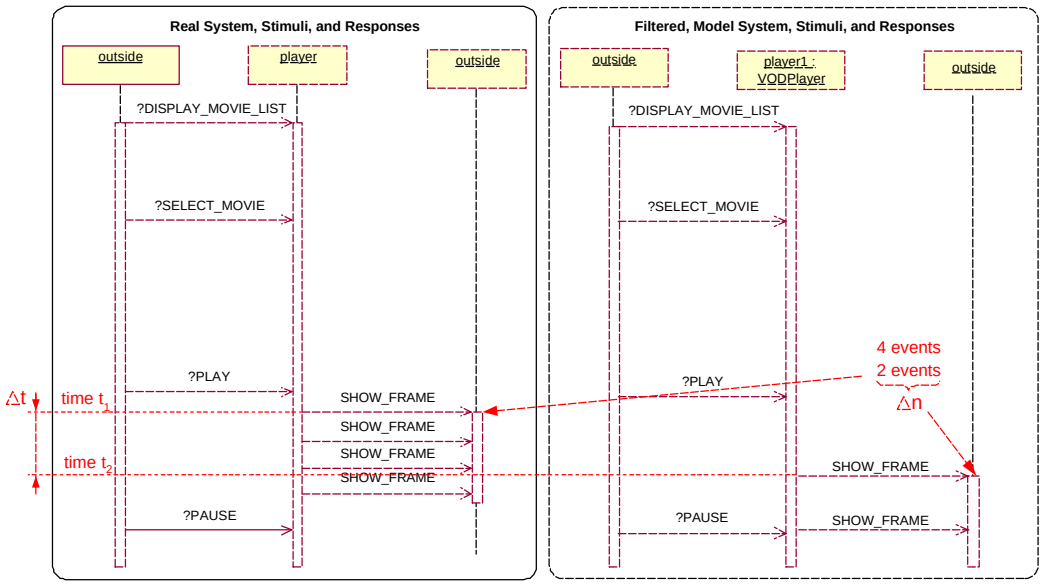

Figure 6. Comparable Real and Model System, Stimuli, and Responses after Aggregation

Figure 4 (right) depicts the corresponding stimuli and responses of the simulated system. Of course, the stimuli are identical with the real system because they where generated from it but we observe that the simulation is much more detailed in terms of what happens inside the VOD player. The VOD player consists of some 20 classes and its model identifies six components. Since we did not instrument all classes, we do not know what happened inside the real system yet the simulator is readily observable. Figure 4 (right) thus depicts all interactions among four of the six simulating components in this scenario.

4. MODEL DIFFERENCING

The goal of the model differencer is to detect whether the behavior of the real system differs from the behavior of the model simulation. To that end, the differencer compares their responses. Model differencing is simply the comparison of the observed system responses with the generated simulation responses (i.e., the model and system stimuli need not be compared because the one is derived from the other). However, to ease the comparison, we translate the system responses into model responses much like we translated the system stimuli into model stimuli (see Figure 5).

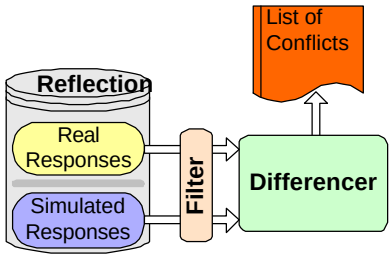

Figure 5. Model Differencing through Observed and Simulated Responses

Ideally, the system responses (in model form) and the simulation responses should be identical. However, we found that the simulation often is in more detail than what is generally observed from the system. In this case, it is necessary to filter the responses. For example, in Figure 4 (left), the real system is subjected to the "?DISPLAY_MOVIE_LIST" stimuli with no response from the system. However, looking at Figure 4 (right), we find that the simulated model engages in a series of interactions in response to the stimuli. These interactions are internal to the system and observable in the simulation but not in the system.

Of course, the executing system also undergoes these internal activities but in this case the internal activities are not observable (e.g., no probes were implemented). Therefore, we have to be careful to only compare responses that are observed by both the simulation and the execution. Therefore, the first task of differencing is to filter the observed responses such that the:

1) interactions between internal components are ignored
2) interactions between component and outside are preserved

Figure 6 (right) depicts the filtered stimuli and responses of the model simulation. It is now readily comparable with the stimuli and responses of the real system (again depicted in Figure 6 (left)). Ideally, the responses of the filtered model and the real system should be identical. For example, the system responds to the PLAY event with a SHOW_FRAME event. We observe that the simulation behaves consistently. However, we also observe that the number of SHOW_FRAME events differs. This difference does not imply an error as there are several permissible exceptions:

- timing difference: simulation and execution likely differ in speed and thus responses are at different times (e.g., first SHOW_FRAME event occurs at a slightly different time).
- count difference: the number of responses may differ in case of periodic activity because of different simulation/execution speed (e.g., four SHOW_FRAME events in the system instead of two events in the simulation).
- ordering difference: concurrently executing components may produce certain events in a slightly different order (no example here) because of execution/simulation speed differences among the components.

In summary, there are cases where the simulation generates responses quicker than the executing system and vice versa. The

difference in the speed of the execution and the simulation has to be taken under consideration during differencing to not identify false differences.

While there are acceptable differences, other difference are signs of error (e.g., out of time, out of order, count violation). These unacceptable differences either imply an incorrect model (e.g., if differencing was used to test the correctness of the model) or an incorrect system (e.g., the system does not behave as predicted in the model). Either way, these differences require action in correcting the model or the system (or both).

If only count and timing differences have to be considered then the algorithm for differencing is straightforward. Dealing with the ordering difference is more difficult. We have not yet developed a generic solution for the latter.

5. SELF MANAGEMENT

During design time, the purpose of differencing is in detecting inconsistencies between the model and the system and correcting the inconsistencies (either by updating the model or the system). This obviously benefits the needs of the software developer and maintainer. However, reflection and differencing are also useful to a system's self management during run time. In a deployed environment we presume that the system completely and consistency implements the model. In this case, a difference between the model behavior and the system behavior indicates system abnormality (e.g., fault, attack, misuse).

WOSS defines the self management as systems with the "ability to adapt themselves at run time to handle such things as resource variability, changing user needs, and system faults." Such ability requires a self reflection in that the system must understand itself and, perhaps, its environment. Model reflection provides the foundation for this self awareness because, at any given time, the system can consult the simulating model about its current state. The simulating model also reflects the system from a perspective that may be more suitable for self awareness as it is in a model form (or at the very least it provides an alternative perspective in addition to its own system reflection).

For self management, the model reflection is also a beacon of correctness when the system fails. If the system behavior differs from the model behavior due to error or attack then the model reflection reveals the presumed correct system state. The system can then engage in corrective actions (i.e., self healing) with the benefit of this additional knowledge (i.e., the actual difference between "is" and "should be"). Of course, if the model is incorrect then our approach fails as it may provide incorrect information about self management. Thus, the consistency and completeness of the model must be validated beforehand which can be done through differencing and exhaustive testing.

While we believe that it is not easy to re-configure a system to a state that is consistent with the model (the presumed correct state), we believe that it is easier to so than trying to recover the system without the knowledge of the presumed correct state (i.e., during

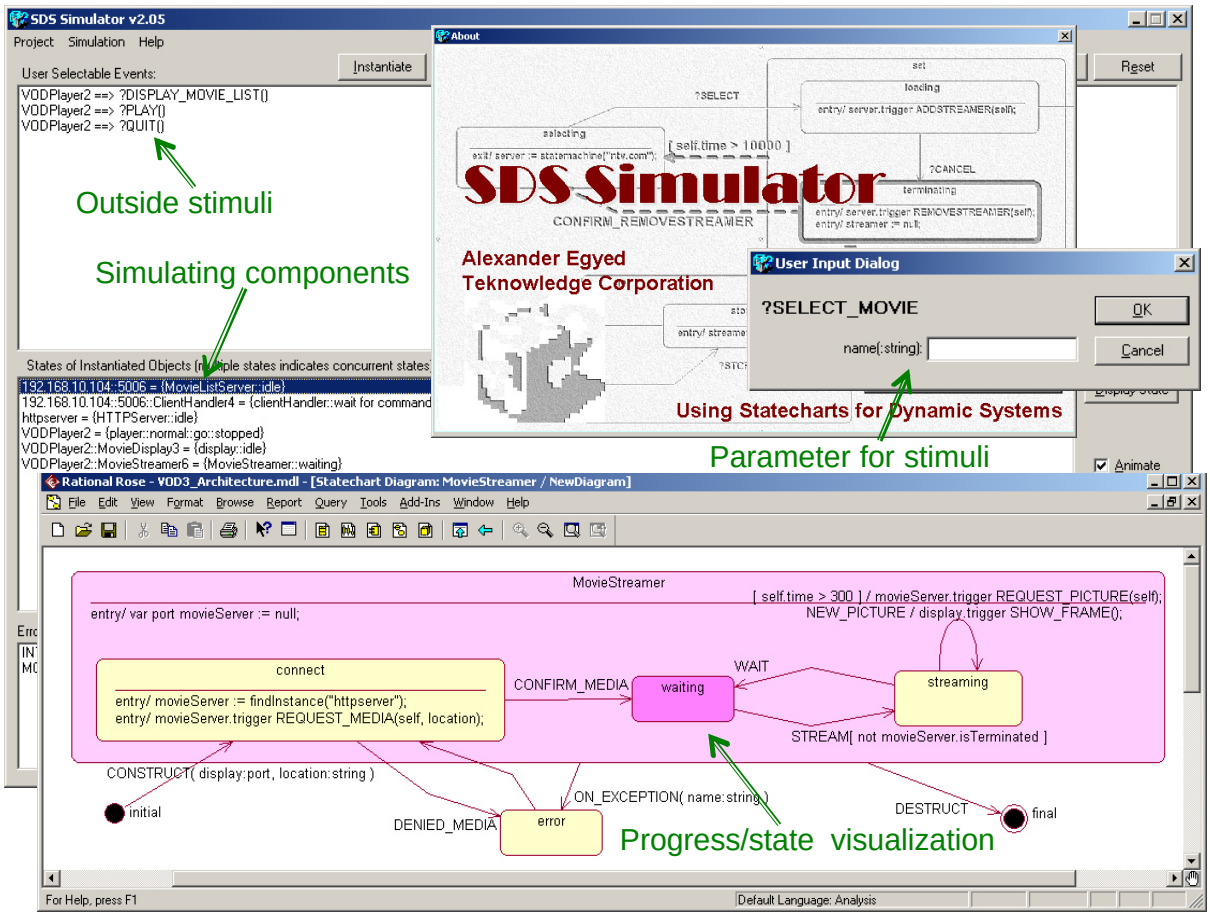

Figure 7. SDSL Langauge (bottom) visualized in IBM Rational Rose and Simulator (top)

rollback and rollforward). Yet, we have not conducted analyses to support this claim. We do believe, however, that "model-driven self management" may complement other self management activities that, perhaps, include system information. For example, a combination of model-driven and system-driven self management may be a reasonable approach where the model reflection is used to guide the self healing of a system and necessary low-level information is recovered from the system itself. For example, MS Word has a recovery method that saves all document data during a crash and re-initiates a new instance of MS Word with the recovered data immediately thereafter. While the recovery action could be model guided, the data needed to do so could be recovered from the system as is done in MS Word. We intent to investigate this and related issues.

There is also a benefit in describing self management in terms of the model where self management activities are included in the model. For example, it is possible to create a mapping table that describes in advance the differences that can occur between the model and the system. This mapping table then recommends self management steps to correct differences. Or, the model simply includes information about self management that are triggered during the simulation. The system then not only looks at the model for its reflection and differencing but also for guidance to its self management. We have implemented some of these capabilities [10] but more research is needed.

6. ISSUES AND CONCLUSIONS

Differencing and self management requires a simulatable, behavioral model. We believe that our approach is open to a wide range of architecture and design models [5,7,8] but, thus far, we only used a language called SDSL (Statecharts for Dynamic Systems Language) [4]. SDSL retains the look-and-feel of Harel's statecharts but unlike Statecharts, SDSL can model dynamic constructs such as remote method invocation, component construction & destruction, late binding, introspection, or instance localization.

SDSL was built by us with the primary goal of providing a dynamic language that can be simulated. We therefore built a simulation tool that takes SDSL specifications (as created graphically in IBM Rational Rose) to animate them. We then extended the simulator, called SDS Simulator, to accept external stimuli of the kind a real system is able to accept (i.e., user input). Figure 7 depicts the simulator during the simulation of the VOD.

To test the simulation language we validated it on several real software systems (e.g., video-on-demand software system [2], OASIS). We then coupled the simulator with the execution of the real systems. In most cases, instrumenting the source code of the real systems was sufficient. The probes intercepted the external stimuli of the executing systems, translated them, and projected those stimuli onto the simulation of their models. We found it highly beneficial to use the simulation to 'spy' into the executing system to observe its behavior. Even without the differencer, we found it very useful to detect inconsistencies.

The reflection and the differencer are not implemented yet although a crude prototype exists. The major issue we foresee is how to model the allowed differences (timing differences, count differences, and ordering differences) that may vary across applications and, even, within applications. We believe that we need to augment SDSL to include allowed differences into the model. The differencer would then use this information.

We have applied the SDSL modeling and simulation capability (an earlier version) onto a self healing project [10] where the simulation modeled the system's behavior and its healing activities. There we demonstrated that it is feasible to use a model as a decision maker for a deployed system. Our findings are preliminary but encouraging.

It is future work to investigate how to incorporate the environment (context) into the model. Also, we intend to investigate how well a system can manage itself based on model information (the model is an abstraction and may not provide all needed information for self management).

7. ACKNOWLEDGMENTS

Our thanks Bob Balzer, Dave Wile, and the anonymous reviewers for insightful discussions. This work was funded by DARPA AWDRAT under the subaward agreement No. 5710001719, AFRL Cooperative Agreement No. FA8750-04-2-0240.

8. REFERENCES

[1] Balzer, R. and Goldman, N.: "Mediating Connectors: A Non-ByPassable Process Wrapping Technology," Proceedings of the DARPA DISCEX Conference, Hilton Head, South Carolina, January 2000, pp.361-368.

[2] Dohyung, K.: "Java MPEG Player," http://peace.snu.ac.kr/dhkim/java/MPEG/, 1999.

[3] Egyed, A. and Balzer, R.: "Unfriendly COTS Integration – Instrumentation and Interfaces for Improved Plugability," Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE), San Diego, USA, November 2001.

[4] Egyed, A. and Wile, D.: "Statechart Simulator for Modeling Architectural Dynamics," Proceedings of the 2nd Working International Conference on Software Architecture (WICSA), August 2001, pp.87-96.

[5] Garlan, D., Monroe, R., Wile, D.: Architectural Descriptions of Component-Based Systems, In Foundations of Component-Based Systems by Gary Leavens and Murali Sitaramam, eds. Kluwer, 2000.

[6] Harel D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 1987.

[7] Magee, J. and Kramer, J.: "Dynamic Structure in Software Architectures," Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 1996.

[8] Oriezy P. and Taylor R. N.: On the role of software architectures in runtime system reconfiguration. IEE Proceedings – Software 145(5), 1998, 137-145 .

[9] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.

[10] Wile, D. S. and Egyed, A.: "An Architectural Style for Self Healing Systems," Proceedings of the 4th Working IEEE / IFIP Conference on Software Architecture (WICSA), Oslo, Norway, June 2004, pp.285-290.