Towards an Integrated Approach for Validating Qualities of Self-Adaptive Systems

Danny Weyns
Department of Computer Science
Linnaeus University, Växjö, Sweden
danny.weyns@lnu.se

ABSTRACT

Self-adaptation has been widely recognized as an effective approach to deal with the increasing complexity and dynamicity of modern software systems. One major challenge in self-adaptive systems is to provide guarantees about the required runtime qualities, such as performance and reliability. Existing research employs formal methods either to provide guarantees about the design of a self-adaptive systems, or to perform runtime analysis supporting adaptations for particular quality goals. Yet, work products of formalization are not exploited over different phases of the software life cycle. In this position paper, we argue for an integrated formally founded approach to validate the required software qualities of self-adaptive systems. This approach integrates three activities: (1) model checking of the behavior of a self-adaptive system during design, (2) model-based testing of the concrete implementation during development, and (3) runtime diagnosis after system deployment. We illustrate the approach with excerpts of an initial study and discuss for each activity research challenges ahead.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Software Quality—*methods for software quality and verification and validation*

Keywords

Self-adaptation, model checking, model based testing, runtime diagnosis

1. INTRODUCTION

Society extensively relies on the qualities of software systems. Examples are the robustness of software for media and the availability of software for business collaborations. Due to the increasing complexity and dynamicity of software systems, assuring and maintaining the required qualities of software constitutes a tremendous challenge. Self-adaptation has been widely recognized as an effective approach to deal with the increasing complexity and dynamicity of software systems [10]. A self-adaptive system comprises two parts: the managed system that deals with the domain functionality and the managing system that monitors the managed system and adapts it to achieve particular quality objectives [13, 27]. The key underlying idea of self-adaptation is complexity management through separation of concerns.

One major challenge in self-adaptive systems is to provide guarantees about the required runtime qualities [10]. With runtime qualities, we mean non-functional qualities that directly relate to a system's runtime behavior, such as availability, reliability, performance, etc. Formal methods provide the means to rigorously specify and verify the behavior of self-adaptive systems both at design time and runtime. However, a recent study of the research results of a representative sample of literature revealed that few researchers are concerned with systematic verification of quality properties of self-adaptive systems [26]. This trend is confirmed in a systematic literature review we just completed focusing on the use of formal methods in self-adaptive systems [28]. These reviews show that most researchers consider verification of quality properties during system design. However, to assure the verified properties, the implementation should conform to the verified architectural models. Moreover, as self-adaptive systems are usually complex systems that operate in dynamic uncertain environments, it is often desirable to monitor the system after deployment and determine incorrect behavior.

In this position paper, we argue for an integrated formally founded approach to guarantee the required software quality properties in self-adaptive systems. This approach integrates 3 activities: (1) static verification of the behavior of self- adaptive systems during architectural design, with (2) model-based testing of concrete implementations during system development, and (3) runtime diagnosis after system deployment. We illustrate the approach with excerpts of an initial study and discuss challenges ahead.

Overview. The remainder of this paper is structured as follows. Section 2 introduces a self-adaptive system that we use for illustration in the paper. In section 3, we underpin our position by reviewing a representative sample of studies that employ formal methods for analyzing properties of self-adaptive systems. Section 4 gives a high-level overview of the integrated approach we propose for validating qualities of self-adaptive systems. In section 5, we elaborate on each of the activities of the approach. We conclude with challenges ahead to realize the proposed approach in section 6.

2. TRAFFIC MONITORING SYSTEM

In this section, we give a brief introduction of a self-adaptive system that we use for illustration purposes. The system consists of a set of intelligent cameras, which are distributed evenly along the road. An example of a highway is shown in Fig. 1. Each camera has a limited viewing range and the task of the cameras is to detect and monitor traffic jams on the highway to assist for example traffic

WODA '12, July 15, 2012, Minneapolis, MN, USA

light controllers or driver assistance systems. Doing this in a decentralized way avoids the bottleneck of a centralized control center. To that end, a software agent is deployed on each camera that monitors the local traffic. If a traffic jam spans the viewing range of multiple cameras, the agents form an organization. Organizations have a master/slave structure; the master provides information to clients that have an interest in traffic jams. The master/slave structure creates dynamic dependencies among the cameras, e.g., a master needs to keep track of its slaves.

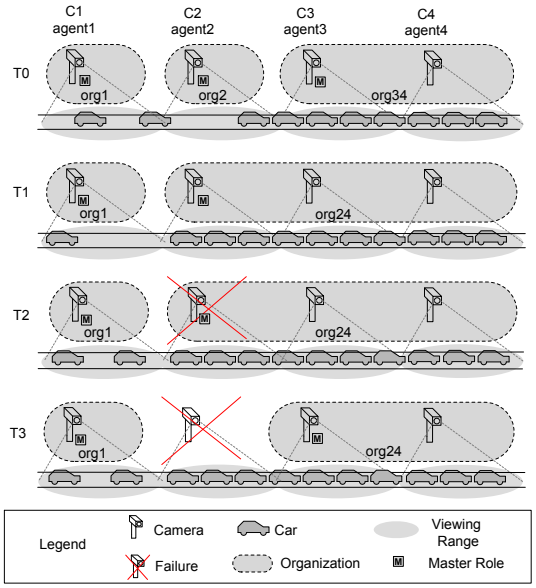

Figure 1: Self-healing scenario

Our particular interest here is on self-healing of the system after a silent node failure, i.e., a failure in which a failing camera becomes unresponsive without sending any incorrect data. This is shown in the scenario from T2 to T3 in Fig. 1. To make the system robust to such failures, we add a self-healing controller to each camera. This controller keeps track of the agent's dependencies. The controllers of cameras with dependencies exchange ping/echo messages to detect failures of other cameras. If a failure is detected, the controllers perform adaptations to bring the agent system back to a consistent state from where it can continue with monitoring traffic jams. For example, if a master fails, a new master will be elected. For a detailed discussion of the traffic monitoring system, we refer the interested reader to [24].

3. FORMAL APPROACHES TO SELF-ADAPTATION

We recently performed a systematic literature review on the use of formal methods in self-adaptive systems that covered 11 main software engineering venues in the field and 5 journals [28]. From the 6353 studies searched, 75 focused on the use of formal methods in self-adaptive systems. From these 75 studies, 51 use formal methods for modeling and reasoning, 13 for model checking, and 11 for theorem proving. We limit our discussion here to the studies that use some form of model checking both offline and at runtime. For more information, we refer the reader to *http://homepage.lnu.se/staff/daweaa/SLR-FMSAS.htm*.

[9] studies an integrated approach for automatic generation of adaptors for Web Service protocols at runtime. The approach combines automata models with rules expressed in linear temporal logic (LTL). The property of interest is conformance with respect to the syntax and order of protocol steps. The authors use "inverse model-based testing" where tests are generated from the WSDL description and validated to check whether the synthesized automaton is a correct data-flow abstraction of the service implementation.

[2] proposes a methodology for the specification and verification of self-adaptive systems, combining communicating sequential processes (CSP) models with properties specified in LTL. Considered properties include correct refinement of the implementation model from the adaptive system model, interference freedom, stability of adaptation behavior, and deadlock freedom.

[19] proposes an approach for constructing autonomous systems that synthesise tasks from high-level goals and adapt their software architecture to perform these tasks reliably in a changing environment. The authors use labeled transition system (LTS) and structural constraints, and focus on verifying reliability properties.

[11] proposes a framework for designing fault-tolerance programs using dynamic program updates triggered by faults. The authors use algebra and set theory for modeling and LTL for expressing rules. The main focus of the work is on failsafe updates and progress during updates.

[7] focuses on design time formalization of self-repairing dynamic software architectures. The authors use T-typed Hypergraph Grammars for modeling, and rules that are checked during graph transformations. Properties of interest are completeness (desirable configurations can be reached) and correctness (for reachable configurations that are not desirable there exist repairing productions).

[14] proposes an approach for model-based runtime adaptation for self-healing systems. Architecture models in Acme are checked via Armani, which evaluates first order constraints on the fly as properties of the architecture change. When problems are detected Armani triggers a repair engine to look for a repair strategy. Considered properties are latency and the load of a server.

The authors of [3] argue that testing cannot provide safety guarantees for complex decentralized systems, while current model checking and theorem proving techniques do not scale for such systems. To remedy this problem, they present a verification technique that exploits the local character of structural safety properties. The approach employs graphs for modeling and rules for defining properties, including system invariants and safety (e.g, hazards).

[29] proposes a model-based approach for developing dynamically adaptive software. The authors use Petri nets and LTL and focus on verifying invariants, liveness, tolerance, and adaptation integrity. A Java implementation is connected with the Petri nets (modeled in Renew) to test the conformance between the executions of the Java implementation and that of the Petri net models.

[20] proposes a model-based framework for developing self-monitoring embedded programs. The approach derives a model-based monitor from the requirements specification described in temporal logic, and instruments a system model to emit events of interest. The composition of the instrumented model with the monitor forms a self-monitoring model that can be used for verification and generating a program that can monitor its own execution.

[12] proposes a formal orchestration model for dynamically adaptable services. The authors use process algebra with first-order logic expressions (FOL). The focus is on three properties: responsiveness (the service always guarantees an answer to every received service request, unless the user cancels), availability (the service is always capable to accept a request), and reliability (the service request can always succeed).

[17] proposes an approach for on the fly behavioral adaptation of component compositions using process algebra and LTS specifica-

tions. The focus is on the automatic generation of adaptor protocols, and verification of matching and absence of deadlock.

[5] employs different verification techniques to learn an abstract description from the observed behavior of the system. The approach allows the application of formal verification methods and tools in the validation process.

[8] proposes an approach to achieve QoS for service-based systems through dynamical adaptation. Formally specified requirements are automatically analyzed to identify and enforce optimal system configurations using a control loop. The approach uses Markov models and probabilistic computation tree logic (PCTL), and focuses on improving response time and dealing with failures.

From this discussion, it is clear that most researchers employ formal methods either to provide guarantees about the design of a self-adaptive systems, or to perform runtime analysis to support adaptations with particular guarantees. The only studies that directly transfer formalization results over different phases of the software life cycle are [9, 20, 29, 8]. While we notice an increasing attention for the use of formal methods in self-adaptive systems, we believe that there is a dearth of approaches that exploit work products of formalization throughout the software life cycle. Such approaches would enable the transfer of quality assurances of self-adaptive systems obtained during design to the implementation and the running system, enhancing the validity of the required qualities.

4. OVERVIEW OF THE APPROACH

We propose an integrated approach to validate the required software quality properties of self-adaptive systems that spans design, implementation, and deployment. In this section, we give a high-level overview of the approach; each activity is discussed in more detail in the next section. Fig. 2 shows the logical sequence of activities of the approach with their respective work products. Although the figure suggests that the activities are executed in a sequence, in practice, typically feedback is used for revising preceding activities (incl. requirements description).

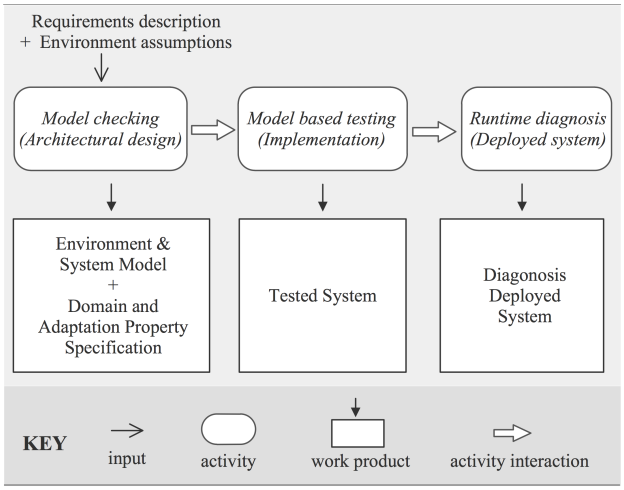

Figure 2: Overview of the integrated approach

The approach starts from a description of requirements and environment assumptions, which are often written in natural language. During architectural design, the requirements are used to create formal models of the self-adaptive system and the relevant parts of the environment. Models can be defined for different parts of the system and at different levels of abstraction. The mixture of abstraction and precision allows us to accurately model what is needed. In addition, domain properties and adaptation properties are defined that define rules or constraints over the models of the self-adaptive system. Domain properties relate to the functionality of the self-adaptive system, adaptation properties relate to the qualities that are subject of adaptation. During model checking the models are verified against the properties, and in case of violations, the feedback (counterexamples) is used to correct the models (or alternatively, the properties may be revised).

While formal verification intends to show that a system has some desired properties by proving that a model of that system satisfies these properties, model based testing starts with a verified model and a set of required properties, and then intends to show that the actual implementation of the system behaves compliant with this model. Model based testing supports automatic generation of tests (supported by user guidance) and execution of the tests. Different setups can be used for unit and integration tests, using corresponding models and properties. The work product of model based testing is an implementation that complies with the verified model that resulted from model checking.

Once the system has passed the necessary tests, it can be deployed. As self-adaptive systems are usually complex systems that have to operate under highly dynamic operating conditions, it is often desirable to diagnosis the behavior of the system after deployment. To that end, the system has to be instrumented and monitored at runtime. The collected data can be used to analyze the system behavior based on the verified models and required system properties. In case of violations, feedback is provided to the stakeholders which can be used for various purposes, from providing statistical information, up to evolving the systems or revising strategies.

5. ACTIVITIES OF THE APPROACH FOR VALIDATING QUALITIES

We now zoom in on the different activities of the approach. We start with model checking, followed by model based testing, and finally runtime diagnosis.

5.1 Model Checking

Given a model of the system and a formal property, model checking enables one to systematically check whether this property holds for the model. Model checking is a relative mature domain [1, 4] and numerous tools are available to verify a variety of different types of models. Fig. 3 shows an overview of model checking for self-adaptive systems.

The work products of model checking are verified models for a set of properties. These work products are central to the integrated approach for validating qualities of self-adaptive systems. We illustrate the different models and properties with examples of the traffic monitoring system described in section 2

Environment Model. One of the abstract models we use to model the environment is that of a car. Fig. 4 shows a timed automaton for a car, designed with Uppaal.[1] Car waits for the $startCar$ signal from the release traffic process (this process releases cars in the environment). Once started, the car moves along the subsequent viewing ranges of the cameras. Whenever a car enters/leaves the viewing range of a particular camera it emits a signal that can be used by the camera agents to monitor traffic congestion.

To reduce the state space during verification of robustness properties, we use an environment model that creates the sequence of traffic conditions as described in Fig. 1, including the failure of camera 1 at T2. This model is shown in Fig. 5.

[1] http://www.uppaal.com/

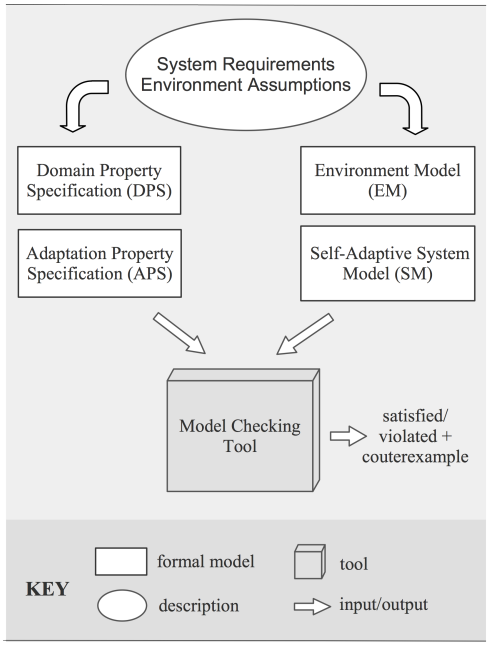

Figure 3: Overview model checking

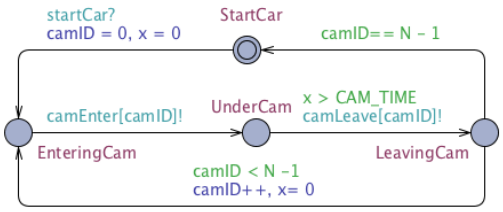

Figure 4: Car

Self-Adaptive System Model. The central component of the self-adaptive system is the self-healing controller. Fig. 6 shows the automaton for the self-healing controller. The controller sends periodically $isAlive[ping]$ signals (based on $WAIT\_TIME$) to the self-healing controllers of the cameras on which the local camera depends (e.g., master/slave and neighbor dependencies). If a camera does not respond in a certain time ($ALIVE\_TIME$) it adapts the organizational controller (i.e., a part of the regular functionality that supports agents with managing organizations), either by removing a dependency in case a slave failed, or by restructuring the organization in case the master of the organization failed.

Domain Propery. One of the required domain properties is that all cameras cannot be slave at the same time (i.e., the system would not provide its function, that is, there are no masters that inform clients about traffic congestion):

```
A[] not forall(i: cam_id) Camera(i).Slave
```

Self-Adaptation Property. We illustrate two self-adaptation properties related to robustness. First, when camera 1 fails then eventually camera 2 and 3 detect the failure.

```
A<> SelfHealingController(1).Failed imply
    SelfHealingController(2).FailureDetected
 && SelfHealingController(3).FailureDetected
```

Second, when camera 1 fails then eventually camera 2 and 3 will

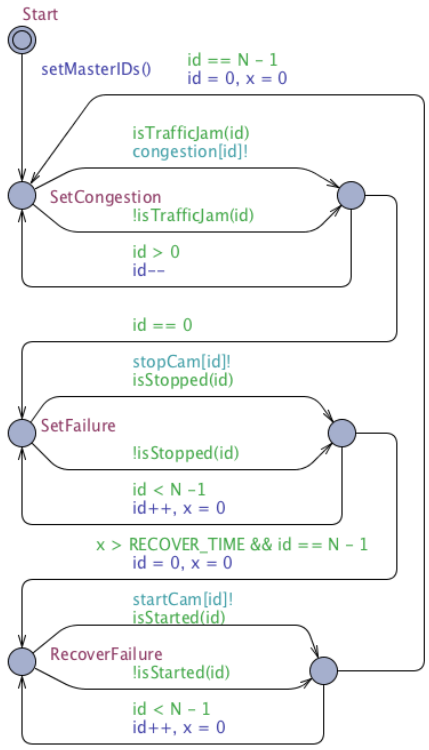

Figure 5: Virtual Agent Environment

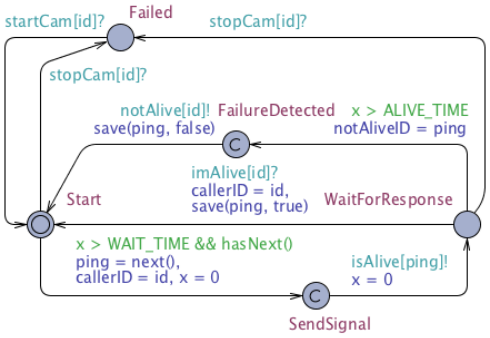

Figure 6: Self-Healing Controller

form an organization (with either one of them as master and the other slave).

```
A<> Camera(1).Failed imply
    ((Camera(2).MasterWithSlaves
      && camera[2].slaves[3])
 ||  (Camera(3).MasterWithSlaves
      && camera[3].slaves[2]))
```

The work products of model checking are the verified models of the self-adaptive systems, together with a set of properties. The models provide the input for the following activities.

5.2 Model Based Testing

[22] characterizes model based testing as automation of black-box test design. Model based testing uses a concise behavioral model of the system under test, and automatically generates test cases from the model. The goal of model based testing is to show that the implementation of the system behaves compliant with this model. Fig. 7 shows an overview of model based testing.

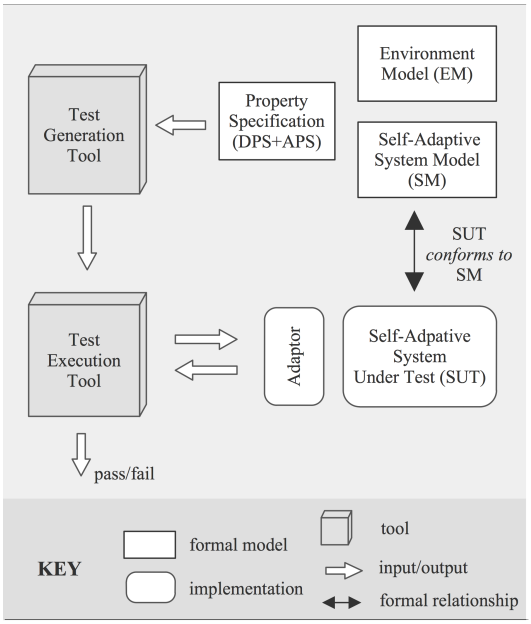

Figure 7: Overview model-based testing

The central aspect of model based testing is the implementation relation that defines the conformance between the system model (SM) and the implementation (SUT). [21] defines the *ioco* (input-output conformance) as follows: any experiment derived from the specification and executed on the implementation leads to an output that is foreseen by the specification. [15] defines relativized timed input/output conformance as the formal implementation relation. Relativized i/o conformance allows checking whether the behavior of an implementation is correct (conforms) to its specification when operating under assumptions about the environment.

An important aspect of model based testing is test selection. As exhaustive testing of realistic systems is typically not feasible, the tester needs to steer test selection. A typical approach is to annotate the models with auxillary variables or automata that allow the test purpose or coverage criterion to be formulated as a reachability property that can be issued to the model-checker. For example, in JTorX[2] the test purpose can be specified in an automaton by marking the success state(s) in which the test purpose is reached.

As an example, to test the self-healing scenario described in Fig. 1, we could mark the $RecoverFailure$ state in the automaton shown in Fig. 5 as a success state. We can then formulate a reachability property to check whether the system will always reach the recover failure state after the camera has failed. This property can be issued to the model checker to test whether the implementation conforms to the model with respect to this property.

An alternative approach is described in [15] where test purposes and coverage criteria are formulated as observer automata that can be automatically superimposed on the model in Uppaal. This avoids explicit changes to the model, and allows the user to specify his own coverage criteria relatively easily.

Finally, to enable test execution, the implementation (SUT) has to be connected with the test execution tool using an adapter. The adapter is an implementation-specific hardware/software component that is responsible for translating abstract input test events into real representations, and physical output observations into abstract model outputs. Whereas the development of the adapter may sometimes be laborious, it is not specific for model based testing.

Besides the formally verified models of self-adaptive systems (SM), we also include the environment models (EM) for testing. [16] argues that modeling the environment explicitly and taking this into account during test generation has several advantages: 1) the test generation tool can synthesize only relevant and realistic scenarios for the given type of environment; 2) the engineer can guide the test generator to specific situations of interest; 3) a separate environment model avoids explicit changes to the system model if testing must be done under different assumptions or use patterns.

We add to these arguments that environment models are a sine qua non for model based testing of runtime *qualities*, which is central to self-adaptation. In the example discussed above, it is the environment model that specifies the failure events that have to be tested. An explicit model of the environment allows an engineer to precisely specify the failure scenarios of interest and the conditions under which the failures happens. For example, in the scenario shown in Fig. 5 a camera failure is generated after traffic is congested, which allows to test the correctness of the system when one of the cameras of an organization that monitors a traffic jam fails. Nevertheless, model based testing of qualities is an open issue [6, 18, 23], and so is the question of which environment (or other) models we need for testing particular quality properties.

The work product of model based testing is an implementation that conforms to the verified design models. It is important to note that testing only shows errors of the implementation against the verified models within the test scope based on test selection.

5.3 Runtime Diagnosis

Self-adaptive systems are designed with run-time adaptation in mind, keeping the user's perceived as well as system intrinsic dependability at a satisfactory level. However, the partial coverage inherent to model based testing of complex systems combined with the uncertainty of the operating conditions under which self-adaptive systems have to operate calls for automated runtime diagnosis. Fig. 8 shows an abstract model for runtime diagnosis of a deployed self-adaptive system.

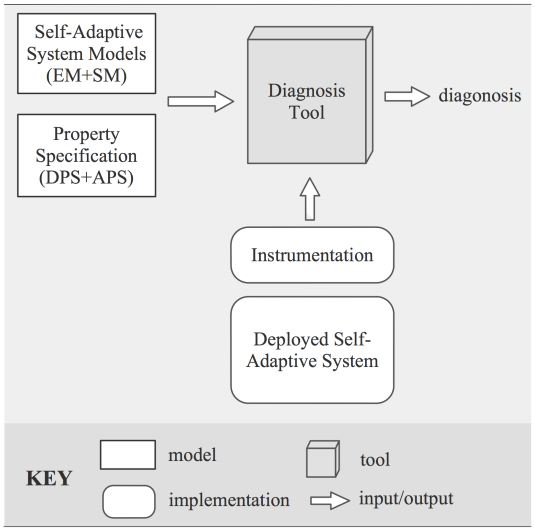

Figure 8: Overview runtime diagnosis

To enable monitoring, the system needs to be instrumented to collect relevant information at runtime. Additionally, probes have to be provided that monitor the relevant aspects of the environment.

[2]https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki/

The data derived from monitoring is used by a diagnosis tool to analyze the system behavior. Analysis interprets the actual system behavior based on the verified behavioral models of the self-adaptive system and the properties of interest. In the example described in section 5.1, we may want to monitor camera failures and verify whether the system recovers within a certain time window.

The results of runtime diagnosis can serve different purposes. At a minimum, violations of properties may be communicated informatively to the stakeholders. Analysis of data over time may be useful for stakeholders to support decision making about a possible evolution of the self-adaptive system. For example, if the data shows that the system systematically does not respond appropriately, this may trigger a system evolution. The runtime models of the self-adaptive system can support integrators with performing online updates of the system. In [25], we have demonstrated that accurate models of the running system are essential to understand dependencies among components and as such to perform correct online updates. The results of diagnosis may also be used for strategic decision making. For example, the failure rate of particular hardware or the resistance of the software to certain security attacks may be used for future decision making.

6. CONCLUSIONS AND CHALLENGES

In this paper, we have argued for an integrated formally founded approach to validate the qualities of self-adaptive systems. The approach is based on the exploitation of the formal work products during subsequent stages of the software life cycle.

We conclude with summarizing the main challenges in each of the activities of the proposed approach. Model checking of self-adaptive systems requires verification of a new class of properties, such as stability of adaptation behavior, failsafe updates, progress during updates, adaptation integrity, mismatch, and interference freedom. Research is needed to get a better understanding of these properties and how they can be verified. The focus of model based testing so far has been on functional correctness of software systems. However, the concerns of self-adaptive systems are primarily of a qualitative nature. Research is needed to study how model based testing can be used to test implementations for quality properties. Finally, formally verified models of self-adaptive systems enable rigorous runtime monitoring and analysis of self-adaptive systems. However, this area is relatively unexplored and further research is needed both for realizing runtime diagnosis and exploring the potential use of it.

In our ongoing work, we explore the use of model based testing for robustness in self-adaptive systems. Our long term goal is to develop a foundation for validating qualities in decentralized self-adaptive systems in line with the proposal outlined in this paper.

7. REFERENCES

[1] C. Baier and J.Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] B. Bartels and Kleine M. A CSP-based framework for the specification, verification, and implementation of adaptive systems. In *SEAMS*, 2011.

[3] B. Becker et al. Symbolic invariant verification for systems with dynamic structural adaptation. In *International Conference on Software Engineering*, 2006.

[4] B. Berard et al. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2010.

[5] R. Borges et al. Integrating model verification and self-adaptation. In *Automated Software Engineering*, 2010.

[6] E. Brinksma et al. Perspectives of model-based testing. In *Dagstuhl Seminar Proceedings 04371*, 2005.

[7] A. Bucchiarone et al. Self-repairing systems modeling and verification using agg. In *WICSA/ECSA*, 2009.

[8] R. Calinescu et al. Dynamic qos management and optimization in service-based systems. *TSE*, 37(3), 2011.

[9] L. Cavallaro et al. Synthesizing adapters for conversational web-services from their wsdl interface. In *Software Engineering for Adaptive and Self-Managing Systems*, 2010.

[10] R. de Lemos et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 2012.

[11] A. Ebnenasir. Designing run-time fault-tolerance using dynamic updates. In *Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2007.

[12] J. Fox. A formal orchestration model for dynamically adaptable services with cows. In *International Conference on Adaptive and Self-Adaptive Systems and Applications*, 2011.

[13] D. Garlan et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37, 2004.

[14] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Self-healing Systems*, 2002.

[15] A. Hessel et al. Formal methods and testing. chapter Testing real-time systems using UPPAAL. Springer, 2008.

[16] K. Larsen et al. Online testing of real-time systems using uppaal: Status and future work. In *Dagstuhl Seminar 04371: Perspectives of Model-Based Testing*, 2005.

[17] R. Mateescu et al. Behavioral adaptation of component compositions based on process algebra encodings. In *Automated Software Engineering*, 2007.

[18] A. Dias Neto et al. A survey on model-based testing approaches: a systematic review. In *Empirical Assessment of Software Engineering Languages and Technologies*, 2007.

[19] D. Sykes et al. From goals to components: a combined approach to self-management. In *Software Engineering for Adaptive and Self-managing Systems*, 2008.

[20] L. Tani. Model-based self-monitoring embedded programs with temporal logic specifications. In *International Conference on Automated Software Engineering*, 2005.

[21] J. Tretmans. Testing concurrent systems: A formal approach. In *International Conference on Concurrency Theory*, 1999.

[22] J. Tretmans. Model based testing with labelled transition systems. LNCS 4949, Springer, 2008.

[23] M. Utting et al. A taxonomy of model-based testing approaches. *Softw. Testing, Verification and Reliability*, 2011.

[24] D. Weyns et al. The macodo middleware for context-driven dynamic agent organizations. *ACM TAAS*, 5(1), 2010.

[25] D. Weyns et al. An architectural approach to support online updates of software product lines. In *9th IEEE Working Conference on Software Architecture*, 2011.

[26] D. Weyns et al. Claims and supporting evidence for self-adaptive systems: A literature study. In *SEAMS*, 2012.

[27] D. Weyns et al. FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM TAAS*, 7(1), 2012.

[28] D. Weyns et al. A survey on formal models in self-adaptive systems. In *First International Workshop on Formal Models in Self-Adaptive Systems, FMSAS*. ACM, 2012.

[29] J. Zhang and B. Cheng. Model-based development of dynamically adaptive software. In *28th International Conference on Software Engineering*. ACM, 2006.