Context-Driven Optimization of Mobile Service-Oriented Systems for Improving their Resilience

Deshan Cooray [1] Sam Malek [1] Roshanak Roshandel [2]

[1] Department of Computer Science
George Mason University
Fairfax, VA 22030, USA
{dcooray, smalek}@gmu.edu

[2] Department of Computer Science and Software Eng.
Seattle University
Seattle, WA 98122, USA
roshanak@seattleu.edu

Abstract— **Mobile software systems are characterized by their highly dynamic and unpredictable execution context. Such systems are permeating a number of domains where the systems operate in constantly changing conditions. We refer to such systems as Situated Software Systems. These systems are often deployed in mission-critical settings with stringent reliability requirements. Existing approaches to performing reliability analysis are insufficient in meeting the demands of situated software systems. We propose an approach aimed at such systems and present it in the form of a framework and tool suite known as REsilient SItuated SofTware system (RESIST). The framework utilizes information from the system's context to produce reliability predictions, and places the system in the optimally reliable configuration with respect to other competing quality attributes.**

***Keywords*-Context Awareness; Software Architecture; Self-Adaptive Systems; Service-Oriented Architecture; Mobility**

I. Introduction

Software systems are increasingly deployed in mission critical domains, including emergency response, industrial automation, navigation, and defense. The majority of such systems are mobile, embedded, and pervasive. They are characterized by their highly dynamic configuration, unknown operational profile, and fluctuating execution context. We refer to this class of software systems as *situated software systems*, since the software in this setting is expected to operate under constantly changing situations and conditions. Given the mission-critical nature of the domains in which situated software systems are deployed, their ability to meet the Service Level Agreement (SLA), especially stringent reliability requirements, is a significant concern.

Determining an architectural configuration for a situated software system that meets its SLA is a challenging task. For example, in a situation where services are provisioned by a mobile platform, deciding the optimum architecture in terms of its software components and their composition requires engineers to perform trade-off analysis between competing Quality of Service (QoS) attributes such as efficiency and reliability. It is clear that the overall reliability of such systems depends on problems both internal (e.g., software bugs) and external (e.g., network disconnection, hardware failure) to the software. The key underlying insight in the proposed research is that some internal software problems may manifest themselves only under certain dynamic characteristics external to the software (e.g., physical location), which is traditionally referred to as *context* [1].

Given that the execution context of situated systems cannot be accurately predicted at the time of design and development, the optimal configuration for such systems cannot be determined prior to its deployment. Moreover, due to variability in the context, no particular configuration of a situated system is optimal for its entire operational lifetime, and hence run-time reconfiguration of the system may be necessary. In situated software systems, the optimal configuration is one that first and foremost provides the required level of reliability as per the SLA, while taking into consideration other quality concerns (e.g., efficiency).

In this paper, we present an approach called *REsilient SItuated SofTware systems (RESIST)* that consists of a framework and automated tool-suite and offers reliability-aware run-time adaptation of situated software systems to address the aforementioned challenges. Our approach is architecture-centric and furnishes reliability predictions at the level of the system's architectural components. RESIST utilizes information from the system's operational *context* to predict the reliability of the system in its near future operation.

The reliability predictions are then used *proactively* to find the most suitable configuration, which is effected through run-time adaptation provided by the infrastructure, thus enabling the system's SLA to be maintained. The most suitable configuration is one that provides the appropriate level of resilience to failure by taking into consideration other quality constraints (e.g., efficiency, security). Unlike the traditional reactive models of adaptation, the proposed adaptation will occur proactively, and before the system's reliability degrades. To make RESIST readily adoptable and widely applicable in a variety of existing situated software systems, we have developed the approach on top of Service-Oriented Architecture (SOA) technology standards [13]. This allows RESIST's SOA-compliant infrastructure to be deployed in a variety of domains to provide the most dependable services possible in the face of unanticipated changes, such as network disruptions, and platform faults, by seamlessly morphing the running software.

The remainder of the paper is organized as follows. Section II describes background and related work. Section III illustrates impact of context on the architecture of a system. Section IV describes the RESIST framework and Section V

describes the infrastructure and tool support and we conclude with an outline of our future work.

II. Background and Related Work

Software reliability is defined as the probability that a system performs its intended functionality correctly under specified conditions [2]. A system's software architecture provides an appropriate level of abstraction to reason about its quality attributes, including reliability. While a system's architectural models can facilitate this process, performing architecture-aware reliability analysis enables architecture-based adaptation techniques to be utilized in order to improve or maintain the system's reliability. An appropriate approach must be able to offer fine-grained analysis based on reliability of constituent elements of the system resulting in the ability to analyze the reliability of alternative configurations. Furthermore, such an approach must be able to offer predictive analysis by accommodating uncertainties associated with the system's operation and context.

Most existing techniques aimed at architecture-based reliability analysis rely on assumptions that make them unsuitable for situated software systems [6][7][8][9]. A majority of the approaches are geared towards static design-time assessment and assume that the system's operational profile is known in advance and does not change at runtime. Further, many of the approaches focus on system-level analysis and assume that component reliability values are known apriori—an assumption that is particularly unsuitable for situated software systems. Finally, the approaches do not consider the impact of contextual change on the system's reliability.

Self-adaptive software systems respond to changes in the operational environment and autonomously reconfigure themselves in order to achieve the overall goals of the system [3]. Given that dynamism and unpredictability associated with situated software systems and their mission-critical nature, the system is required to adapt in such a manner that its effectiveness is maintained throughout the mission.

Related to our work are the general purpose architecture-based adaptation frameworks [3][4][5]. These frameworks are primarily *reactive* in their decision making while the mission critical nature of situated software systems requires that the system adapts in *anticipation* to degradation in its QoS (e.g., reliability) beyond the allowable threshold. Nevertheless, these frameworks can form the basis for systems that adapt proactively.

III. Impact of Context on Architecture

Any type of information that characterizes the runtime conditions of the system, and alters its behavior can be considered its context [11]. A system's context may consist of several different aspects of its changing execution environment that could potentially impact the behavior and properties of a system. Among them three main categories of context can be identified [11][12];

- *Computing Environment*, such as the available resources, including CPU, network bandwidth, battery power.
- *User Environment*, such as the user's location, social situation, and an ongoing activity.
- *Physical Environment*, such as near-by objects, the amount of light, and temperature.

A context-aware system uses knowledge about its context to provide relevant information and/or services to the user [11]. While in some systems contextual information is directly used to provide services to the user, in some others contextual information is used to optimize the manner in which services are provided to the user. For example, a GPS enabled mobile phone which displays a map based on the user's location considers the location as an input to the service that is provided. In contrast, a mobile robot engaged in firefighting may need to reconfigure itself depending on its contextual characteristics so that its dependability is optimal with respect to other quality attributes such as resource usage. As described in the next section, RESIST is aimed at the second class of systems. Specifically, RESIST uses the system's context to perform architectural reconfiguration of the system so that it remains resilient in the face of degrading reliability.

Changes to the operational context of a system impact its runtime behavior which in turn could potentially impact the system's quality attributes such as reliability. In architecture-based adaptation the system's software architecture forms the basis for adaptation reasoning. Consequently, we argue that it is important to be able to model the effect of changes in the context on a system's architecture as a first class entity. In our work, we adopt a broad interpretation of system's *architecture*, which simply captures the *knowledge* about the system. This *knowledge* includes many different aspects of the system, including the principle design decisions about the system, its structure and behavioral models, as well as behavioral properties of the system captured in the form of an operational profile model.

To exemplify the effect the context has on a system's architecture, below we present how the mobile nature of a robotic system introduces contextual changes that can impact its operational profile, and in-turn its reliability. Figure 1(a) shows the architectural models of the mobile robot. It receives a command from an external system such as a PDA, and returns the result of executing the command. Upon receiving a command, it uses its Sensors to gather data about its environment, such as near-by obstacles and proximity to heat, and determines a plan and executes it using its Navigator and Actuator components, respectively. Figure 1(b) shows the robot's Controller component's behavioral model in the form of a UML state chart. It includes behavioral states *idle*, *estimating*, *planning* and *moving*, during which the Controller invokes interactions with the other components in the system (i.e., Sensors, Actuator, Navigator, etc.). The *failed* state denotes a common failure state of the component. Transitions O_1 to O_6 denote behavioral transitions resulting from input events such as interface calls on the component. Transitions F_1 to F_3 denote a failure that may arise under some circumstances. Such failures are caused by faults in the software that could lead to a failure. Transition S denotes eventual recovery of the

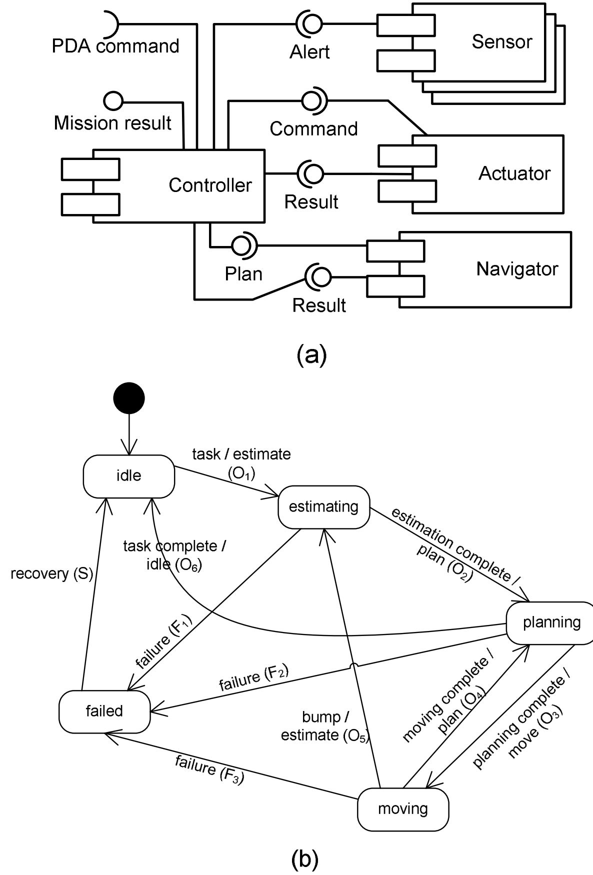

Figure 1. Robot's architecture: (a) robot's structural model, and (b) the behavioral model of the robot's Controller

component as a result of automatic or manual re-initialization of the component.

This behavioral model depicts both the robot's internal behavior as well as interactions with the external environment. For example, O_1 corresponds to an input task from the user, and O_5 corresponds to bump events triggered from the physical environment as a result of colliding with, or being within close proximity of an obstacle. Changes in the contextual environment may impact the frequency of these input events, which in turn alters the frequency of these two state transitions O_1 and O_5. The resulting changes in the execution frequency of the states in turn change the frequency of failures as well. For example, if the estimating state happens to be a state from which failures happen frequently, situations in which robot navigates through a dense terrain can increase bump events, which consequently increases the frequency of transition to the estimating state, and thus the probability of component failure. Thus in this example, the contextual changes resulting from the robot's mobility, in turn impacts the component's reliability.

The impact of the system's context is not limited to internal changes in the component behavior, as they may also change the manner in which components interact, and thus influence the system's reliability. For example, the Controller interacts with the Sensors in order to perform estimations prior to planning its navigation route. However, if the number of bump events increases, the Controller interacts with the Sensors with a higher frequency in order to perform re-estimations. Thus, the impact of the Sensor components' reliability on system's reliability depends on how frequently the Controller needs to interact with the Sensors, which is in turn determined by location dependent contextual information such as the complexity of the terrain (i.e. the probability of bumps).

Therefore the changes in context and its effect on the system's architecture can be modeled as follows:

- A set of contextual parameters $C = \{C_1, \dots, C_l\}$, which includes any information about a system's context that impacts the system
- A set of architectural parameters $A = \{A_1, \dots, A_m\}$, which includes architectural properties that change as a result of the system's context
- A set of interactions $I = \{I_1, \dots, I_n\}$ between contextual and architectural parameters where in each interaction, one or more contextual parameters cause a change in an architectural parameter
- A set of functions that captures the effect of the above interactions on architectural parameters. $\forall i \in I$, these functions are of the form;

$$\mu_i: \mathcal{P}(C) \rightarrow A \quad (1)$$

where $\mathcal{P}(C)$ denotes the power set of contextual parameters. In the case of the robotic system above, the probability of the robot encountering an obstacle on its path (a contextual parameter which changes as a result of its mobility), has an effect on two architectural parameters: the transition probability from *moving* to *estimating* state in the Controller, and the probability that the Controller interacts with the Sensor components. In this example, we have described two points of interaction between the contextual parameter and architectural parameters, but in any sizable system one could expect multiple points of interaction, which further highlight the importance of properly modeling and incorporating context in engineering mobile systems.

In the next section, we present an overview of the RESIST framework, and how the context information is used in predicting system's reliability and optimizing system's architecture.

IV. OVERVIEW OF THE RESIST FRAMEWORK

RESIST consists of a comprehensive and integrated approach to monitoring, assessing, and adapting of situated software systems. An overview of RESIST framework is depicted in Figure 2. The process is organized as a feedback-control loop that continuously monitors, analyzes, and adapts the system at run-time. RESIST consists of three conceptual software components, implemented as meta-level components.

At design-time and before the system's implementation is complete, an initial set of architecture-based reliability models are developed which are later evolved during implementation. At runtime these models are used to assess a variety of configuration choices. Furthermore, they serve as

the basis for building predictive models to assess future reliability of the system. Unlike the traditional architectural models, they embody contextual properties necessary for reliability analysis of situated systems. As described below, these models are expected to be updated and refined at runtime.

Architecture-based reliability models along with contextual and monitoring information obtained from the system are used by the *Component-Level Reliability Analyzer* to predict the reliability of system's components in their near future operation. These fine-grained reliability estimates are used by the *Configuration Reliability Analyzer* to determine the reliability of alternative configurations for the system. The *Configuration Selector* is in turn used to select a suitable configuration for the near future operation of the system. The configuration selector may use other quality attributes, such as performance, in the selection process. The process for obtaining and estimating these properties is beyond the scope of this paper, which is focused on reliability concerns.

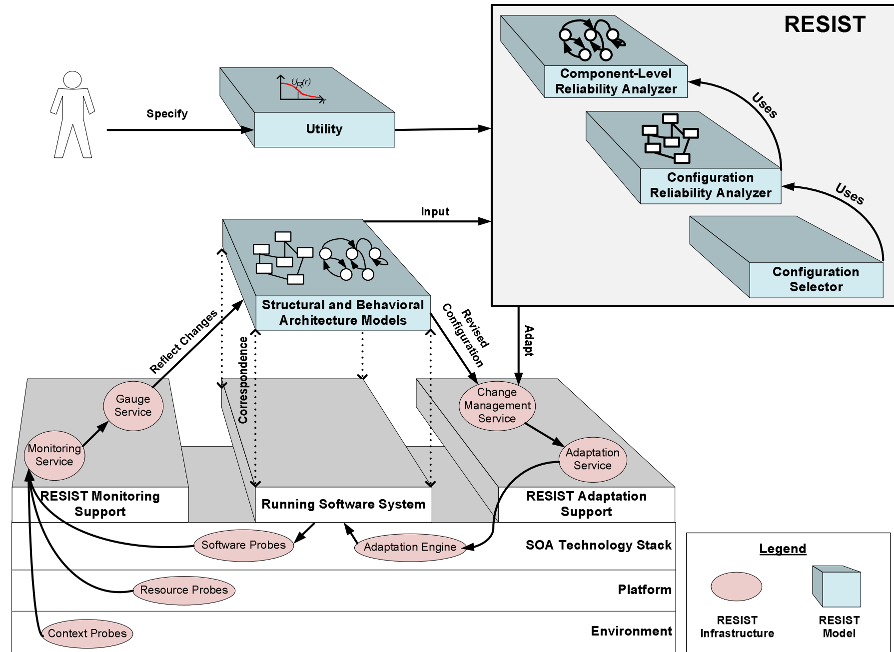

Figure 2. RESIST Framework and its Context-Aware SOA Infrastructure Support

Once a new configuration is selected, RESIST uses its context-aware middleware infrastructure to adapt the system (details are described in Section V). This middleware adapts the system at runtime to reflect changes in the configuration. The middleware provides support for execution, monitoring, and adaptation of a software system in terms of its architectural constructs (e.g., components, connectors, and configuration). At runtime, the middleware monitors the software system for information that is used to refine the reliability predictions. This information is obtained from multiple sources, such as monitoring internal (e.g., frequency of failures, exceptions, and service requests) and external (e.g., network fluctuations, battery charge) software properties, changes in the structure of the software (e.g., disconnection of components due to network drop outs, off-loading of components due to drained battery), and contextual properties (e.g., physical location). Since the monitored data represents the most recent operational, structural, and contextual profile of the system's execution, it can be used to assess the system reliability more accurately. Note that unlike previous approaches [14][15] we do not rely solely on the monitoring data. Instead, we incorporate architectural knowledge, monitoring data, and contextual changes at runtime in a complementary fashion to produce more accurate results.

A. Reliability Analysis

As shown in Figure 2, RESIST performs the reliability analysis at two levels: at component level, and at configuration level. At both levels, architecture-based reliability techniques are used in conjunction with monitoring information obtained from the system and its context. Since context impacts both the internal behavior of components and the interactions among them, the context information is incorporated into the reliability analysis at both component and configuration level.

In order to perform reliability analysis and prediction, we need to consider the Software Operational Profile (SOP) of the system [2]. SOP represents the set of executions that take place in a software program along with the probabilities with which they will occur in a given environment. As described in Section III, these probabilities may be affected by changes in the system's context. In this case, we model these probabilities as relevant architectural parameters.

For the purpose of modeling the SOP, we use existing techniques based on Discrete Time Markov Chains (DTMC) at both component and configuration levels. A DTMC is defined as a stochastic process with a set of states $S = \{S_1, S_2, \ldots, S_N\}$ and a transition matrix $A = \{a_{ij}\}$, where a_{ij} is the probability of transitioning from state S_i to state S_j. Once the operational profile of the system in the form of a DTMC has been estimated, the context information is used to update the transition probabilities in the DTMC, so that it reflects the future operational profile.

Our reliability models used for component and configuration reliability prediction rely on Hidden Markov Models (HMMs) [23] to estimate the transition probabilities of the DTMC (i.e., the matrix A mentioned above). As confirmed by our previous results [22], HMMs can be used to learn from runtime data and to obtain transition probabilities.

In the case of component reliability, the states are identified using the component's behavioral model, such as the state chart diagram depicted in Figure 1(b), while runtime data obtained through monitoring the system becomes training data for the HMM. We use the Baum-Welch algorithm [23] to train and solve the HMM. The resulting transition matrix for the component is updated based on the contextual parameters using functions in the form of equation (1) that capture the contextual impact on architectural parameters. The updated transition matrix represents the new operational profile for the component based on the context in which the system is expected to operate in the near future. The reliability prediction is computed by solving for the steady state probability (obtained from standard numerical methods [24]) of not being in the *failed* state (recall Figure 1(b)).

Once the component level reliabilities are computed, we use them to arrive at configuration level reliability predictions using an existing technique [10], where the system reliability is computed in terms of its component reliabilities. Here, the components in the system's structural model such as the one depicted in Figure 1(a) is used to derive the states for the Markov Chain. The transition probabilities are derived in a manner similar to the component-level reliability, except that now the monitoring data used consists of interactions between components. Similar to component reliability prediction, transition probabilities at configuration level consist of architectural parameters affected by context. Hence, functions of the form of equation (1) are used to update relevant transition probabilities in order to capture the effect of the context in the near future operation of the system. Solving the model as per [10] using updated transition probabilities and component reliabilities yields the prediction for configuration reliability.

B. *Architectural Optimization*

The reliability estimation approach presented earlier can be used to determine the most reliable configuration for a situated software system. The optimal configuration in RESIST is defined as one that satisfies the system's reliability requirement, while improving other quality attributes of concern (e.g. efficiency). Consequently, the configuration selection problem becomes one of an optimization problem. Specifically, RESIST's objective is to find an architectural configuration which satisfies quality requirements defined in the SLA.

RESIST provides the capability of specifying preferences for other (potentially conflicting) quality attributes using utility functions. The user indicates these preferences so that they are consistent with the QoS requirements specified in the SLA. In turn, these utility functions are used by the *Configuration Selector* to perform trade-off analysis between conflicting quality concerns when selecting the optimally reliable configuration. When searching for alternative configurations, RESIST considers options such as changing the software architecture (e.g. replication of software components), and changing the deployment (e.g., component to process allocation).

V. INFRASTRUCTURE SUPPORT

Our approach for dynamic monitoring and adaptation builds on our previous research on adaptive software architectures [16][17], context-aware middleware [18][19], and dynamic SOA systems [17][20]. The approach is consistent with the widely accepted three layer model of self-management [3]: (1) *Goal Management Layer*—plans for change; (2) *Change Management Layer*—coordinates the change; and (3) *Component Control Layer*—executes the change. Figure 2 depicts RESIST's infrastructure support. Consistent with our objective of complying with the SOA principles, we expose RESIST's monitoring and adaptation facilities as discoverable services. Details are described below.

The pink ovals in Figure 2 represent the underlying facilities that will be deployed on each RESIST platform. *Context*, *Resource*, and *Software Probes* monitor environmental, system, and software parameters, respectively. For example, a *Context Probe* may correspond to readings provided from a physical sensor, a *Resource Probe* may correspond to an instrumentation of OS for collecting CPU utilization metrics, and a *Software Probe* may provide statistics on the invocation frequency of a service. Since *Context* and *Resource Probes* do not necessarily depend on the SOA technologies [13], they are placed outside of the *SOA Technology Stack*. Adaptation Engine provides a reflective capability that can be leveraged to adapt the services hosted on a given RESIST platform, such as binding a client to a new provider, changing a service provider's operations, instantiating new services, and so on. Both *Software Probes* and *Adaptation Engine* depend on other SOA technologies, and are placed in the *SOA Technology Stack*.

As shown in Figure 2, *Monitoring Service* provides a set of operations that allow other services to discover and obtain monitoring data from any platform. Clearly before a service can access this data, the appropriate authentication is performed, the details of which are not discussed for brevity. Similarly, *Adaptation Service* provides a set of operations that expose the adaptation capabilities supported by the underlying *Adaptation Engine* to the rest of the system.

The monitoring data provided by *Monitoring Services* are used by *Gauge Services*, which look for new patterns of behavior in the monitored data, and potentially process the raw data for further analysis. Once a change in the monitored data of interest is identified, *Gauge Service* updates the appropriate *Architectural Models*. This in turn triggers RESIST's *Analyzer*, which uses the updated models to detect potential changes in the system's expected reliability in the manner described earlier and to possibly find a better architectural configuration. If a new configuration is selected for effecting, *Change Management Service* is invoked, which leverages the adaptation operations provided by the *Adaptation Service* to actually change the running system. *Change Management Service* coordinates the adaptation process to ensure that the system's functionality is not jeopardized, and the system downtime is minimized. For

this, we have built on earlier research on pattern-based dynamic software adaptation [21].

In order to enable engineers construct the required architectural models using well-known Architectural Description Languages, and to perform reliability analysis and architecture trade-off analysis, RESIST uses an extensible architectural modeling and analysis environment, called XTEAM [25]. XTEAM enables engineers to build structural and behavioral models such as xADL, and FSP. We extended XTEAM's structural and behavioral meta-models with the annotations needed for reliability analysis. To that end, the traditional FSP support in XTEAM was extended to include the notion of failure states, and associated a transition probability with each FSP actions. We also extended the traditional xADL model support in XTEAM to model reliability properties of the architectural constructs, such as component reliability.

VI. CONCLUSION

Mobile software systems are characterized by their highly dynamic and unpredictable execution context. When placed in mission-critical settings, such systems' ability to meet their SLA, in particular stringent reliability requirements, is of utmost concern. This is further complicated by mobile system's dynamic operational context, and the inability for a single architectural configuration to meet reliability requirements throughout the system's lifetime. The contributions of our work include (1) modeling the system's context as a first class entity, (2) incorporating context into reliability analysis to yield reliability predictions, (3) automatically find the optimal architectural configuration that meets the SLA, and (4) SOA infrastructure for the implementation of such systems using RESIST.

In our future work, we intend to evaluate the scalability of RESIST in large-scale software systems comprising of hundreds of components and hardware hosts. We also intend to increase the types of reconfiguration decisions and dependability trade-offs that RESIST supports. Finally, we plan to investigate the use of other stochastic approaches (e.g., Dynamic Bayesian Networks, and Hierarchical HMM).

VII. REFERENCES

[1] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. IEEE Transactions on Software Engineering, Vol 29, 2003.

[2] H. Pham, System Software Reliability, Springer, 2006.

[3] J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. International Conference on Software Engineering, Minneapolis, Minnesota, May 2007.

[4] B. Cheng, et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. Software Engineering for Self-Adaptive Systems, LNCS hot topics, 2009.

[5] D. Garlan, et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. IEEE Computer, 37(10), 2004.

[6] K. Goseva-Popstojanova, et al., Architecture-Based Approaches to Software Reliability Prediction. International Journal of Computer and Mathematics with Applications, 46(7), Oct 2003.

[7] S. Krishnamurthy, A. Mathur. On the Estimation of Reliability of a Software System Using Reliabilities of its Components. International Symposium. on Software Reliability Engineering, 1997.

[8] R. Reussner, et al. Reliability Prediction for Component-Based Software Architectures, Journal of Systems and Software, 66(3), 2003.

[9] G. Rodrigues, et al. Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems. International Conference on Fundamental Approaches to Software Engineering, Edinburgh, UK, April 2005.

[10] W. Wang, D. Pan, M. Chen. An Architecture Based Software Reliability Model. Journal of Systems and Software, 2005.

[11] G. Abowd, et al. Towards a Better Understanding of Context and Context-Awareness. Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing, p.304-307, September 1999, Karlsruhe, Germany.

[12] B. Schilit, et al. Context-Aware Computing Applications. 1st International Workshop on Mobile Computing Systems and Applications, December 1994.

[13] S. Weerawarana, et al. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall, 2005.

[14] S. Krishnamurthy, A. Mathur. On the Estimation of Reliability of a Software System Using Reliabilities of its Components. International Symposium on Software Reliability Engineering, 1997.

[15] W. Wang, et al. Moving Average Modeling Approach for Computing Component-Based Software Reliability Growth Trends. INFOCOMP Journal of Computer Science, 5(3), 2006.

[16] S. Malek, et al. Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, May 2007.

[17] S. Malek, et al. Self-Architecting Software Systems (SASSY) from QoS-Annotated Activity Models. International Workshop on Principles of Engineering Service Oriented Systems (PESOS 2009), Vancouver, Canada, May 2009.

[18] S. Malek, at al. A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. IEEE Transactions on Software Engineering, vol. 31, no. 3, March 2005.

[19] S. Malek, at al. Tailoring an Architectural Middleware Platform to a Heterogeneous Embedded Environment. International Workshop on Software Engineering and Middleware (SEM 2006), Portland, Oregon, November 2006.

[20] N. Esfahani, at al. A Modeling Language for Activity-Oriented Composition of Service-Oriented Software Systems. ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 09), Denver, Colorado, Oct 2009.

[21] H. Gomaa, et al. Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures. Fourth Working IEEE/IFIP Conf. on Software Architecture, Oslo, Norway, 2004.

[22] L. Cheung, R. Roshandel, et al. Early Prediction of Software Component Reliability. International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 2008.

[23] L. R. Rabiner. A Tutorial on Hidden Markov Models, in Proceedings of the IEEE, vol. 77, pp. 257-286, 1989.

[24] W.J. Stewart. Introduction to the numerical solution of Markov Chains. Princeton University Press, 1994.

[25] G. Edwards, S. Malek, and N. Medvidovic. Scenario-Driven Dynamic Analysis of Distributed Architecture. International Conference on Fundamental Approaches to Software Engineering (FASE 2007), Braga, Portugal, March 2007.