RESISTing Reliability Degradation through Proactive Reconfiguration

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

[1] George Mason University
Department of Computer Science
{dcooray, ckilgor1, smalek}@gmu.edu

[2] Seattle University
Department of Computer Science and Software Engineering
roshanak@seattleu.edu

ABSTRACT

Situated software systems are an emerging class of systems that are predominantly pervasive, embedded, and mobile and marked with a high degree of unpredictability and dynamism in the execution context. At the same time, such systems often need to satisfy strict reliability requirements. Most current software reliability analysis approaches are not suitable for situated software systems. We propose an approach geared to such systems, which continuously furnishes refined reliability predictions at run-time by incorporating various sources of information. The reliability predictions are leveraged to *proactively* place the software in the optimal configuration with respect to changing conditions. Our approach considers two representative architectural reconfiguration decisions that impact the system's reliability: reallocation of components to processes and changing the architectural style. We have realized the approach as part of a framework, called REsilient SItuated SofTware system (RESIST), and evaluated it using a mobile emergency response system.

Keywords

Reliability, Software Architecture, Self-Adaptation, Mobility

1. INTRODUCTION

Software systems are increasingly situated in dynamic mission critical settings, such as emergency response. *Situated software systems* are predominantly mobile, embedded, and pervasive. They are characterized by their highly dynamic configuration, unknown operational profile, and fluctuating operating conditions. In turn, reliability of such systems has become an especially important area of concern.

Engineers of a situated software system typically spend significant effort to determine a good configuration for the system to ensure its adherence to functional and non-functional requirements. For instance, they may perform a trade-off analysis between the system's efficiency and reliability when they decide the allocation of software components to operating system (OS) processes. Clearly 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 our 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].

Due to variability in the execution context, often the *optimal configuration* for a mobile system cannot be determined prior to its deployment. Moreover, 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. Given the mission critical nature of situated software, the optimal configuration is one that provides the required level of reliability, while taking into consideration other quality attributes of concern (e.g., efficiency).

Unfortunately, majority of the existing reliability estimation approaches are not suitable for situated software systems and are instead geared to static design-time analysis [8][9][10][11][13][21][22][25][29]. With the exception of KAMI [6], a framework for run-time model adaptation, other approaches make several unrealistic assumptions. Most approaches do not offer insights into obtaining component reliabilities and only focus on system level analysis (except [9][21]). Others assume upfront availability of operational profile [13][21], and predictive trend modeling to estimate future reliability of the system [29]. Finally, black-box approaches that treat the system as a monolithic entity [8] and ignore the internal structure of the system are not suitable for the situated software system domain.

In this paper, we present REsilient SItuated SofTware system (RESIST), a framework intended to address reliability concerns in mission critical dynamic and mobile setting. RESIST furnishes a compositional approach to reliability estimation starting with analysis at the component level, which in turn makes it possible to assess the impact of adaptation choices on the system's reliability. The analysis is performed continuously at run-time by incorporating various sources of information. On top of the architectural models and the monitoring data, RESIST incorporates contextual information (e.g., physical location) to predict the reliability of the system in its near future operation.

RESIST uses the reliability predictions to (1) proactively determine when the system should be adapted, and (2) find the optimal configuration for the near future operation of the system. Our evaluations under carefully controlled conditions show that our analysis is accurate with respect to the *observed* system reliability. In practice however, the unanticipated nature of this domain as well as external factors such as network operation could impact the *observed* system reliability. We thus refer to the predicted reliability values as an *indicator* (i.e., relative measure of reliability) and use it primarily for decision making. An important contribution of our work is proactive adaptation based on our reliability analysis that reconfigures the system at run-time prior to actual reliability degradation. This trait clearly sets our work apart from the majority of existing self-adaptive frameworks that are *reactive* in their decision making [2][12].

We have developed a prototype implementation of RESIST on top of a tool-suite, which consists of an existing context-aware architectural middleware integrated with a visual architecture-

based modeling and analysis environment. Finally, RESIST is evaluated using a mobile emergency response system.

The remainder of this paper is organized as follows. In Section 2 we describe a software system that is used for both motivating the research and evaluation of our work. Section 3 provides a high-level overview of RESIST framework. In Sections 4 and 5 we present the reliability models for software components and overall configurations, respectively. Section 6 offers insights into how an optimal configuration is selected. A prototype implementation of RESIST and evaluation of the approach are presented in Sections 7 and 8, respectively. Finally, the paper concludes with an overview of related work and avenues of future research.

2. MOTIVATING EXAMPLE

Emergency response is a domain that entails a high degree of mission criticality. Software systems designed for this domain thus have stringent reliability requirements. As a motivating example, consider a mobile distributed emergency response system intended to aid the emergency personnel in fire crises, a prototype of which was developed in our previous work [5]. This system consists of several entities, including a central *dispatcher* that serves as the "Headquarters" for coordinating the crew activities, smart *fire engines* that are designed to alert the dispatcher of the current location of the vehicle and provide its occupant with information concerning the crisis scene, *firefighters* equipped with PDAs capable of controlling the robots and sensors, and mobile *robots* that execute the high-level commands.

While the entire system is highly dynamic and could benefit from our approach, for the clarity of exposition in this paper we focus on the robotic subsystem. The robot consists of several electronic sensors and mechanical actuators that allow it to autonomously navigate, detect smoke, stream video, and extinguish fire. It is constrained by limited battery life, memory, processing speed, and connectivity. Architectural design choices affecting the system at run-time aim at accommodating these constraints.

An example architectural strategy for improving the system's efficiency is the use a thread-based architecture. Software components are deployed as separate threads within a single OS process, thus allowing for the resources (e.g., stack memory) to be shared among components while avoiding the overhead (e.g., context switching) associated with managing many separate processes. However, since a process may exit prematurely due to an errant thread, a disadvantage of the thread-based model is a potential decrease in system reliability due to the propagation of fault among components.

Figure 1 shows two alternative allocations of the robot's software components to OS processes. Based on the above discussion, from system's perspective it is reasonable to expect the architecture depicted in Figure 1a to be more efficient, while the one depicted in Figure 1b to be more reliable. Determining the best configuration depends on (1) the device's fluctuating resources (e.g., available memory, CPU utilization, battery charge), and (2) the reliability of the system's constituent components, which as will be detailed later varies based on the system's context.

The above scenario demonstrates the impact of architectural decisions on system's quality attributes. Such decisions while critical to system's dependability cannot be made effectively at design-time. It is only reasonable to assume that some of these decisions must be made at run-time, requiring specialized methodologies that continuously evaluate the impact of these decisions on system's dependability. We use this system in the remainder of the paper to describe and evaluate our approach.

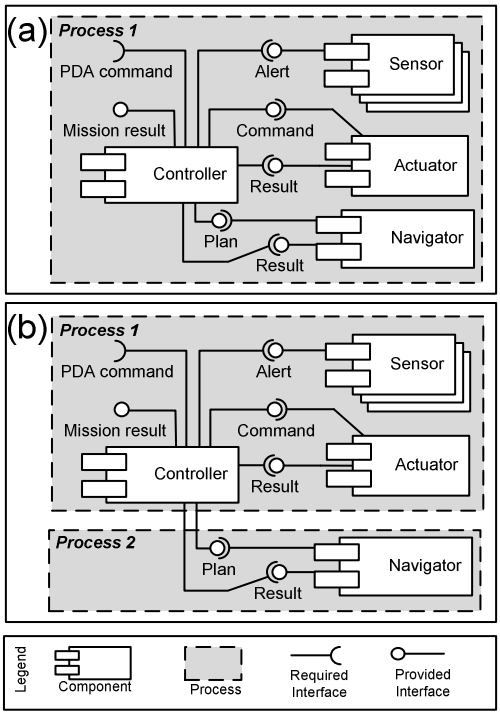

Figure 1. Component-to-process allocation alternatives.

3. FRAMEWORK OVERVIEW

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 on top of a context-aware middleware.

At design time and before the system's implementation is complete, an initial set of architecture-based reliability models are developed. These models are used at run-time to assess a variety of configuration choices and to serve as predictors for the future reliability of the system. Unlike traditional architectural models, they embody contextual properties necessary for reliability analysis of situated systems. As will be described below, these models are expected to be updated and refined at run-time.

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 utilized 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 making the configuration selection. Note that the process for obtaining and estimating such properties is neither shown in the figure, nor discussed in detail in this paper, since the focus of this paper is on satisfaction of reliability concerns.

Once a new configuration is selected, the *Context-Aware Middleware* adapts the system at run-time to reflect the changes in configuration. The *Context-Aware 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 run-time, 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 software properties (e.g., network fluctuations, battery charge), 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 we do not rely solely on the monitoring data. Instead, we incorporate architectural knowledge, monitoring data, and contextual changes at run-time in a complementary fashion to produce more accurate results.

In the next three sections, we describe the three conceptual components of RESIST in details using the mobile emergency system introduced earlier.

4. COMPONENT RELIABILITY

Structural and behavioral knowledge embedded in software architectural models provide an appropriate level of abstraction from which reasoning about system's quality attributes is feasible [18]. Software architectural models are typically *compositional* in nature: structure and behavior of complex systems are described in terms of the structure of constituent components and their interaction. Despite this however, majority of existing architecture-based reliability modeling approaches largely focus on analysis at the system level. As identified by recent survey [8][10][11], a shortcoming of those approaches that incorporate individual component reliabilities into analysis, is the assumption that component reliabilities or reliability of their services are known apriori. In the domain of dynamic, situated, and mobile software systems, it is even more critical to incorporate the reliability analysis of individual components. This is because the reliability of components and systems depend greatly on the *operational context* in which they are deployed. For this reason, an upfront static reliability prediction of the software components and systems offers little help in the face of dynamic and adaptable system configuration: reliability assessment must become an integral part of self-adaptive systems, where adaptation scenarios are assessed in terms of their impact on reliability.

Our component-level reliability model relies on dynamic learning techniques, specifically Hidden Markov Models (HMMs) [20], to provide continuous reliability refinement. Component reliability is estimated stochastically using a Discrete Time Markov Chain (DTMC) and in terms of the fraction of the time spent in failure states by the component. A DTMC is defined as a stochastic process with a set of states $S = \{S_1, S_2, ..., S_N\}$ and a transition matrix $A = \{a_{ij}\}$, where a_{ij} is the probability of transitioning from state S_i to state S_j. Reliability is computed by solving for the steady state probability (obtained from standard numerical methods [28]) of *not* being in any failure state. A number of

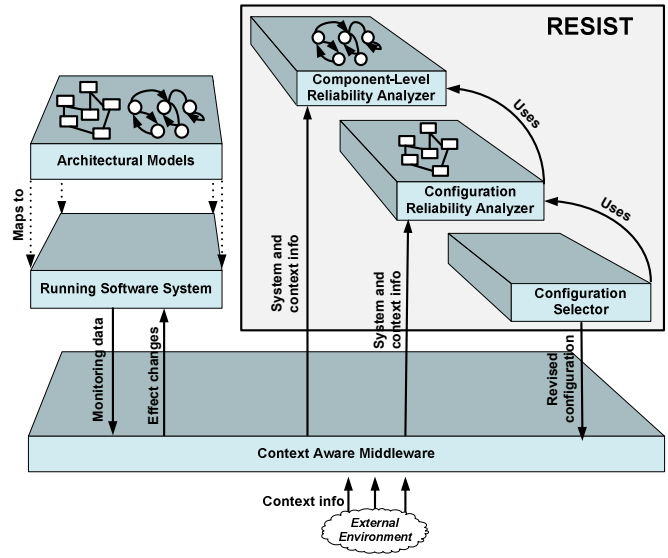

Figure 2. Overview of RESIST framework.

approaches can be taken to ensure tractability if the state space size is determined to be too big [28].

Obtaining transition probabilities (matrix A) can be challenging especially at design-time. Our past research [3] has explored a range of information sources that can be used to derive these probabilities at design-time. In the case of mobile, distributed, and situated software systems, obtaining these values are further complicated by the fact that the system's behavior changes at run-time in response to changes external to the system. We rely on the availability of monitoring data obtained from the running system to determine the transition probability matrix A. While a standard Markov-based approach would assume that there is a one-to-one correspondence between observed run-time events and sequence of states in the model, such correspondence may not exist in systems with realistic level of complexity.

As confirmed by our preliminary results [3], in such circumstances Hidden Markov Models (HMMs) [20] can be used to *learn* from run-time data and to obtain behavioral transition probabilities. An HMM is defined by a set of states $S = \{S_1, S_2, ..., S_N\}$, a transition matrix $A = \{a_{ij}\}$ representing the probabilities of transitions between states, a set of observations $O = \{O_1, O_2, \ldots O_M\}$, and an observation probability matrix $E = \{e_{ik}\}$, which represents the probability of observing event O_k in state S_i. The sets S and O of the HMM comes from component's architectural model (e.g., statechart diagram), while run-time data obtained from monitoring becomes training data for the HMM.

We use the Baum-Welch algorithm [20] to train and solve the HMM. The input to the algorithm is the data obtained from run-time monitoring of the software system, and consists of sequences of observations. Given an initial HMM constructed as described above, the Baum-Welch algorithm converges on the transition matrix A, which as described above is used to calculate probability of failure (or unreliability) in a DTMC.

To clarify the approach, consider the state machine depicted in Figure 3 for the *Controller* component of the robot in the emergency response system introduced earlier. When the *Controller* is in *idle*, *estimating* and *planning* states, it can receive commands from the fireman PDA, and when the *Controller* is in

the *estimating* or *moving* states the robot is making use of its sensors or actuators. From this diagram we can derive the sets:

States $S = \{S_1 \ldots S_4, F\}$ and Observations $O = \{O_1 \ldots O_{13}\}$

where F denotes a common failure state, $S_1 \ldots S_4$ denote behavioral states (*idle, estimating, planning, moving*), and $O_1 \ldots O_{13}$ denote the observations (state transitions).

At run-time, the system is monitored to obtain execution traces in the form of observation sequences. These execution traces are then used to train the HMM, using the Baum-Welch algorithm. The Markov model obtained from this algorithm represents the operational profile of the system based on the training data, which represents the system's behavior based on its current context.

To better illustrate the concepts, consider the following transition probability matrix obtained by running the Baum-Welch algorithm on sample data obtained from the robot's *Controller*:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0.0677 & 0.9307 & 0 & 0.0016 \\ 0.2325 & 0.1080 & 0 & 0.6575 & 0.0020 \\ 0 & 0.0756 & 0.9227 & 0 & 0.0017 \\ 0.3889 & 0 & 0 & 0 & 0.6111 \end{bmatrix}$$

The steady state vector obtained from A represents the probability of being in any of the states as the system operates overtime: $\begin{bmatrix} 0.1016 & 0.1817 & 0.4299 & 0.2827 & 0.0042 \end{bmatrix}$. In this case, the last column represents the probability of failure. The *Controller* reliability based on its present run-time context is computed as:

$$R_c = 1 - 0.0042 = 0.9958$$

As mentioned earlier, given the dynamism present in the mobile and adaptive domain, it is critical to incorporate the notion of *context* into the analysis. Context corresponds to conditions external to the software system, which impact the behavior of the system, and hence change its reliability. As a result, to satisfy their reliability requirements, situated software systems may need to be reconfigured in response to contextual changes.

An important contribution of our research is the incorporation of this contextual knowledge into our reliability predictions, which enables proactive reconfiguration of the software prior to actual degradations in reliability. In the case of this example, the robot periodically takes snapshots of the environment and using existing techniques [24] determines the complexity of the terrain. The robot then compares the complexity of the current terrain with previous snapshots. In cases where the terrain seems less/more complex than the past operational context, the model is updated to reflect the contextual change. For example, if there are many obstacles in the field the robot anticipates more *bumps.* In the transition probability matrix the probabilities corresponding to the robot's behavior in presence of bumps (e.g., probability of transition from *moving* to *estimating* states) are updated to reflect this contextual change.

More generally, we define a set $C = \{C_1, \ldots, C_x\}$ to denote a set of contextual parameters monitored by our runtime infrastructure. Our goal is to arrive at a revised transition probability matrix A' that more accurately reflects the near future operation of the component given the expected contextual changes. If a_{kj} is an element in matrix A that is affected by changes in a specific contextual parameter C_l, then $a'_{kj} = \mu(a_{kj}, \Delta C_l)$, where μ is a

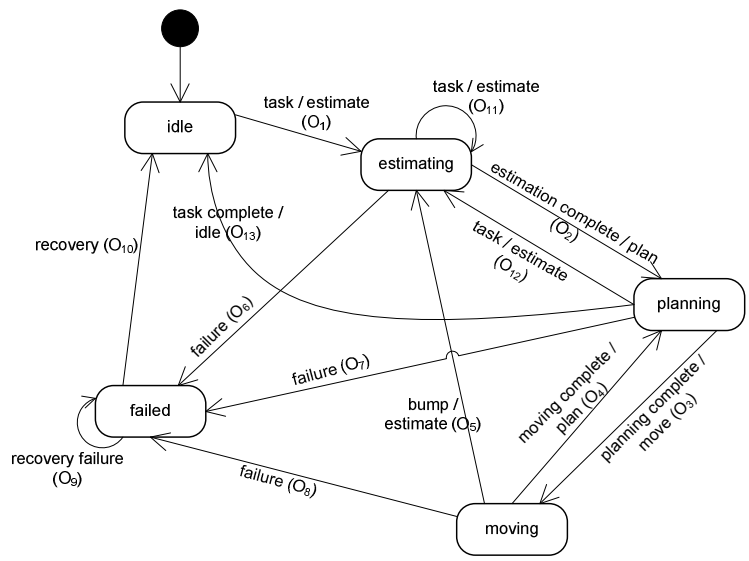

Figure 3. Behavioral model of the robot's Controller component.

context-specific function quantifying the impact of contextual change on the transition probability. In the case of the robotic system, we have used the technique described in [24] to update the probability of transitioning from *moving* to *estimating* states based on the complexity of the terrain. When adjusting a'_{kj} the other elements in row k of the matrix A must also be revised to ensure the cumulative probability of all possibilities in that row remains at 1: $\sum_i^m a_{ij} - a_{kj} + a'_{kj} = 1$.

The reliability analyzer then utilizes the executable model of the architecture, and the updated transition probability matrix A' to generate new *simulated* training data that more accurately represent the anticipated operational context. The new simulated training data is essentially a sequence of observations generated based on the probability distribution corresponding to the matrix A'. Finally, the HMM model is retrained using the new training data and updated components (and subsequently system) reliability values are computed.

Using the kind of analysis described here, we are able to *proactively* adapt the system's behavior in order to avoid a degradation of the system reliability in response to changes in the operating context.

5. CONFIGURATION RELIABILITY

Once the reliability of all components is obtained, a compositional model is used to determine the reliability of specific system configurations. Configuration reliability is in turn leveraged to assess the adherence of a given configuration to the system reliability goals. When a system does not meet the intended reliability threshold, run-time adaptation becomes necessary to ensure that the system's reliability requirements remain satisfied.

While majority of run-time adaptation approaches take a *reactive* stance in response to degradation of the system reliability, our approach can be used *proactively* in anticipation of reliability degradation. This is done by system monitoring and continuous reliability assessment that incorporates fluctuating operational context as described earlier. In the rest of this section, we briefly describe the system-level reliability analysis approach and the role of architectural style and deployment architecture.

5.1 System Reliability Calculation

Our Markov-based system-level reliability estimation approach is based on the model presented by Wang et al. [30], where the system reliability is estimated compositionally based on the reliability of individual components, the architectural style governing their interactions, and the system's operational profile. A DTMC is built by mapping the components and their interactions to a state diagram [30]. A state s_i maps to one or more components in concurrent execution whose completion is required in order to transfer control over to the next state. A *state transition* with a probability P_{ij} represents the probability of undergoing a transition from s_i to state s_j. Accordingly, system reliability R is computed as:

$$R = (-1)^{k+1} R_k \frac{|E|}{|I-M|} \qquad (1)$$

where M is a $k \times k$ matrix in which s_i is the entry state and s_k is the exit state and whose elements are computed as follows:

$$\mathrm{M(i,j)} = \begin{cases} \mathrm{R_i P_{ij}} & \text{state } \mathrm{s_i} \text{ reaches state } \mathrm{s_j} \text{ and } \mathrm{i} \neq \mathrm{k} \\ 0 & \text{otherwise} \end{cases}$$

where R_i is the reliability of state s_i, and R_k is the reliability of the exit state. $|I-M|$ is the determinant of matrix $(I-M)$, while $|E|$ is the determinant of the remaining matrix excluding the last row and the first column of $(I-M)$.

As an example, consider the following deployment scenario for the emergency response robot. A fireman interacts with the robot using a PDA. The firemen issues a high-level command (e.g., go into the restaurant and extinguish a grease fire) which is received by the *Controller*. The *Controller* decides upon the appropriate sequence of intermediate actions, which will result in the successful completion of (or inability to complete) the original command. To complete the task, the *Controller* makes use of a variety of sensors, which detect obstacles, proximity, and heat, a navigator which plots waypoints, and a mechanical actuator which is used to perform the physical activities.

Let us assume that the initial component reliabilities for the *Controller* and *Navigator* components are respectively computed to be *Controller: C* = 0.9958 and *Navigator: N* = 0.9921 using the approach described in Section 4. For the purpose of this illustration, we assume the remaining components and connectors in the system (*Input Communication Connector: IC, Touch Sensors: TS_1, TS_2, Heat Sensors: HS_1, HS_2, Proximity Sensors: PS_1, PS_2, Actuator: A,* and *Output Communication Connector: OC*) are 100% reliable.

The state model in Figure 4a depicts the control flow interactions among the various components in this configuration, and the transition probabilities between the components obtained through run-time monitoring. As shown, each of the components *IC*, TS_1, TS_2, *C, N, A* and *OC* have been mapped directly to a state since they execute in a sequential manner. Components HS_1, HS_2, PS_1, and PS_2 have been mapped to a single state *S* since they all execute in parallel upon receiving control, and upon completion the control transfers back to *C*. From this state model a corresponding transition matrix *M* is created with the matrix elements representing probability of successfully transitioning from state S_i to S_j computed as $R_i \times P_{ij}$. In cases where a state transition occurs in a sequential manner, R_i is the reliability of the component executing in state S_i, whereas when a transition occurs out of the parallel set, R_i is the multiplication of the reliabilities of all components in state S_i. Using the transition probabilities in the state model (P_{ij}) and the component-level reliabilities, we obtain the following for transition matrix *M*:

$$\begin{array}{c} \\ IC \\ TS_1 \\ TS_2 \\ C \\ S \\ N \\ A \\ OC \end{array} \begin{array}{c} \begin{array}{cccccccc} IC & TS_1 & TS_2 & C & S & N & A & OC \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.3844 & 0.2519 & 0.1026 & 0.2569 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.9921 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Solving the model according to equation (1) yields a system reliability of 0.9765.

5.2 Impact of Architectural Style

Architectural styles are a set of constraints on the structure and behavior of a system to elicit particular desirable qualities [18]. Use of specific architectural styles is a way to apply preconceived solutions to similar recurring software problems. Run-time adaptation and reconfiguration of the system aimed at improving system's quality may often require changes to the system's architectural style. The fault tolerant style, for example, improves reliability by replicating critical components. A fault tolerant connector in the form of middleware can be used to handle component failures and to manage the hot standby copies. In the case of the robot, the original architecture (Figure 1b) demonstrates the system when the components are allocated to two processes with the *Navigator* component running on its own OS process. Applying the fault tolerant architectural style in this case can improve the reliability by replicating the *Navigator* component, which represents a critical point of failure. Figure 5

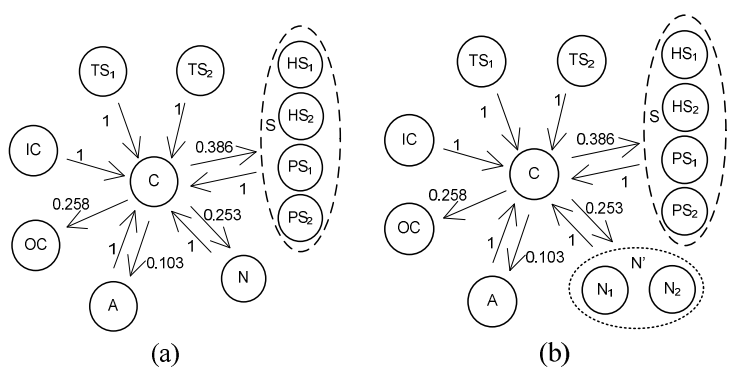

Figure 4. (a) State model for the robot (b) State model with the Navigator replicated.

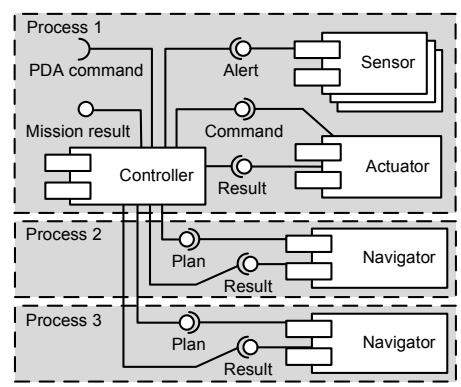

Figure 5. Architecture with replicated Navigator.

shows a replicated *Navigator* component added to the original architecture while running on a new process. The corresponding state model (Figure 4b) shows the two replicated instances of the *Navigator* N_1 and N_2 both mapped to state N'. The reliability of the new state N' can be computed as the probability that at least one of them does not fail [30]. Hence the probability of state N' executing correctly increases to 0.9999. Assuming the reliability of all other components and each of the Navigators to be the same as before, the following matrix *M* can be constructed.

$$\begin{array}{c} \\ IC \\ TS_1 \\ TS_2 \\ C \\ S \\ N' \\ A \\ OC \end{array} \begin{array}{c} \begin{array}{cccccccc} IC & TS_1 & TS_2 & C & S & N' & A & OC \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.3844 & 0.2519 & 0.1026 & 0.2569 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.9999 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Solving the model above according to equation (1) yields an improved system reliability of 0.9839.

5.3 Impact of Deployment Architecture

A system's deployment architecture is essentially an allocation of its software components to hardware hosts and OS processes. A system may be realized using more than one deployment architecture. At the same time, the deployment architecture has a significant impact on system's reliability. In this paper, we focus on the component-to-process allocation, as another representative method employed by RESIST to prevent reliability degradations.

When multiple components are allocated to the same process, a component failure could propagate to other components within the process, and impact their reliability. In this case, redeploying components to separate processes could improve a system's reliability. In the case of the robot, consider two deployment configurations of the architecture where the *Controller* and the *Navigator* are deployed as two separate processes, and as two threads sharing the same process.

Let's assume that *N* and *C* represent reliability of the *Navigator* and the *Controller* components respectively when they execute on separate processes. When the two components are redeployed to share the same process, the effective reliability of each component is simply $N \times C$, where failure in either *N* or *C* will cause both components to fail. For instance, assuming that *N* and *C* to be 0.9921 and 0.9958 respectively, the effective reliability of the two components would be $N' = C' = 0.9879$. Intuitively, the drop in the two components' reliability results in a decrease in the overall system reliability. Therefore, the initial deployment architecture in which the two components were deployed as separate processes yields better configuration reliability.

6. CONFIGURATION SELECTION

The reliability estimation approach presented earlier can be used to determine the most reliable configuration for a situated software system. However, in reality, reliability estimates are used in conjunction with the estimates of other quality attributes (e.g., efficiency, response time) to determine the *optimal configuration* for the system. This is due to the intricate relationship resulting in a tradeoff among different quality attributes. As you may recall, the optimal configuration in RESIST is defined as one that provides the required level of reliability, while taking into consideration other quality attributes. In other words, the goal of RESIST is to ensure the satisfaction of reliability requirement, while reducing the negative impact on other quality attributes. This is a reasonable objective for the domains targeted by RESIST (i.e., mission critical), but it may not be appropriate for others. Consequently, the configuration selection problem becomes one of an optimization problem[1]. Specifically, RESIST's objective is to find an architectural configuration C^* such that:

$$C^* = argmax_{(C)} \sum_{\forall q \in QualityObjectives} U_q(C)$$

Subject to $R(C) > \delta$

where U_q is a utility function indicating the engineer's preferences for the quality attribute *q*, *R* is a function that calculates the expected reliability of a given architecture *C* based on the method described in Section 5, and δ is the system's reliability requirement $\delta \in \mathbb{R},\ 0 < \delta < 1$.

A utility function is used to perform trade-off analysis between competing (conflicting) quality concerns. In the emergency response system, we have two utility functions: one specifies the user's preference for improvements in *reliability*, while another one specifies the same for *efficiency*. RESIST does not place a constraint on the format of utility function. For expressing the preferences in the emergency response system we have adopted sigmoid curve functions, which have been shown appropriate for eliciting and expressing user's QoS preferences [26].

The optimization is subject to ensuring the specified reliability requirement is not violated. RESIST also uses this constraint to determine when a reconfiguration of the system is necessary.

There are $O(P^C)$ ways of allocating software components to OS process, where *P* is the number of processes and *C* is the number of component, and $P<C$. The total number of different architectures resulting from the application of fault tolerant style is $O(N^C)$, where *N* is the maximum number of replicas allowed per component. Thus, the size of the solution space for this optimization problem is $O((N\times P)^C)$. Clearly the solution space is large, even for small value of *N*, *P*, and *C*.

Many commonly available algorithms could be used to solve this problem. For small problems RESIST finds the optimal solution using Integer Programming Solvers, while for large problems it uses stochastic algorithms, such as greedy and genetic. For brevity the details of these algorithms are not provided in this paper.

7. IMPLEMENTATION

We have developed a prototype implementation of RESIST that integrates (1) an extended version of XTEAM [4] as the environment for maintaining the structural, behavioral, and reliability models (2) Prism-MW [15] as the context-aware middleware for obtaining monitoring data from the system and effecting reconfiguration changes, and (3) an off-the-shelf HMM toolbox for MATLAB.

XTEAM is an extensible architectural modeling and analysis environment that supports modeling of a system's software architecture using several well-known Architectural Description Languages (e.g., FSP and xADL for modeling the behavioral and structural properties of a system respectively). 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

[1] The analytical models used for estimating quality attributes other than reliability are outside the scope of this paper.

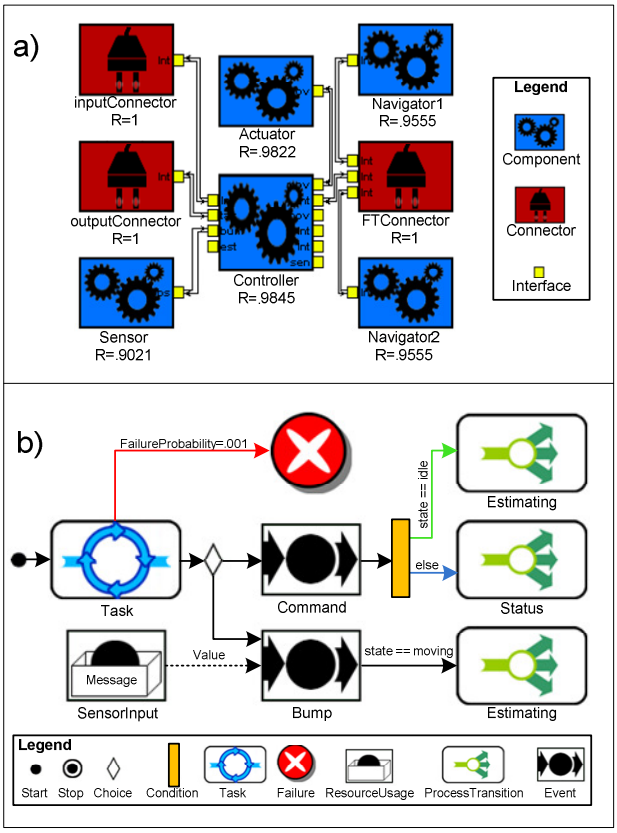

Figure 6. Reliability-annotated architectural models of a portion of robot's Controller component in XTEAM: (a) structural view in xADL, and (b) behavioral view in FSP.

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. Figure 6 depicts a snapshot of the reliability-annotated xADL and FSP models for a subset of the robot's software system.

We have used XTEAM's API for accessing and modifying the reliability-annotated models, which are then used to develop RESIST's reliability analysis and proactive reconfiguration modules. RESIST's analysis module reads the reliability-annotated architectural models to generate the appropriate HMM, which is then solved using MATLAB's HMM toolbox. The estimated reliability values are then used to find an optimal configuration for the system.

The running system is implemented on top of Prism-MW which is integrated with RESIST to facilitate *monitoring* and *adaptation.* Prism-MW's monitoring services provide the run-time data and contextual information needed for RESIST's analysis. The reliability analysis may determine the need to change the system's configuration to prevent reliability degradation. In turn, a new configuration is effected by making the appropriate changes to XTEAM's architectural models. Whenever XTEAM's models change (i.e., RESIST selects a new configuration), an *architectural diff* is performed, and the differences are effected through the dynamic adaptation services of Prism-MW. The details of Prism-MW's support for mobility, context-awareness, and adaptation are beyond the scope of this paper [15].

8. EVALUATION

We have evaluated RESIST using its prototype implementation and the mobile emergency response system described earlier. The evaluation consists of three criteria: (1) the accuracy of RESIST's reliability analysis with respect to the observed reliability at both component and system level, (2) the impact of architectural reconfiguration decisions on system's reliability, and (3) the effectiveness of proactively reconfiguring the system with respect to changes in the system's execution context. We have used XTEAM to control the system's operational profile (i.e., usage) and Prism-MW for gathering run-time data.

8.1 Accuracy of Reliability Analysis

We have validated the accuracy of results produced by RESIST's reliability analysis against the observed reliability of the system. The observed reliability corresponds to the system's *reliability* from the usage perspective (i.e., black-box) and is estimated as the ratio of the total number of successfully received responses to the total number of sent requests. As will be detailed below, while observed reliability provides a good run-time measure of reliability against which we can assess RESIST's predictions, it does not necessarily represent the system's "actual" reliability, which no approach can claim to be able to furnish.

Our goal in the evaluation is to show that RESIST's reliability predictions are good indicators of the reliability trends. In other words, since the reconfiguration logic relies on RESIST's reliability predictions for making decisions, it is important to first show the quality of predictions. To that end, we have used RESIST to predict the reliability of both the robot system and its software components, and compared it against the observed reliability. To clearly demonstrate the accuracy of reliability predictions, we controlled the *Navigator* and *Controller* components by injecting defects with a specified probability of failure. The usage of the robot was also controlled, allowing us to evaluate the accuracy of predictions under identical execution scenarios. Finally, since in this set of experiments we are only concerned with the accuracy of our analysis, the robot's execution context was fixed.

The top two diagrams in Figure 7 present the comparison of estimated and observed reliability for both the *Navigator* component and the overall system in a configuration where the *Navigator* runs in its own dedicated process. The bottom two diagrams in Figure 7 depict the same for a different configuration, in which the *Navigator* and *Controller* are sharing a process. The failure probability is increased in steps of 0.0005. In both experiments, as expected, component and system reliability decreases as failure probability increases. The difference between the estimated and observed reliability values is negligible when the failure probability is low and intensifies as the failure probability increases. This behavior can be attributed to the fact that RESIST's Markov-based approach calculates the reliability at a more fine-grained level than the black-box approach taken for calculating the observed reliability. Hence, RESIST's HMM-based analysis is more sensitive to both behavioral and failure transitions for the reasons explained below.

For clarification, let us focus on the *Navigator* component and consider the following two execution scenarios when it receives a request: (1) several hundreds of behavioral transitions occur before a failure, or (2) a single behavioral transition occurs before a failure. In the HMM analysis, the duration of failure-free execution is taken into consideration; the component may have

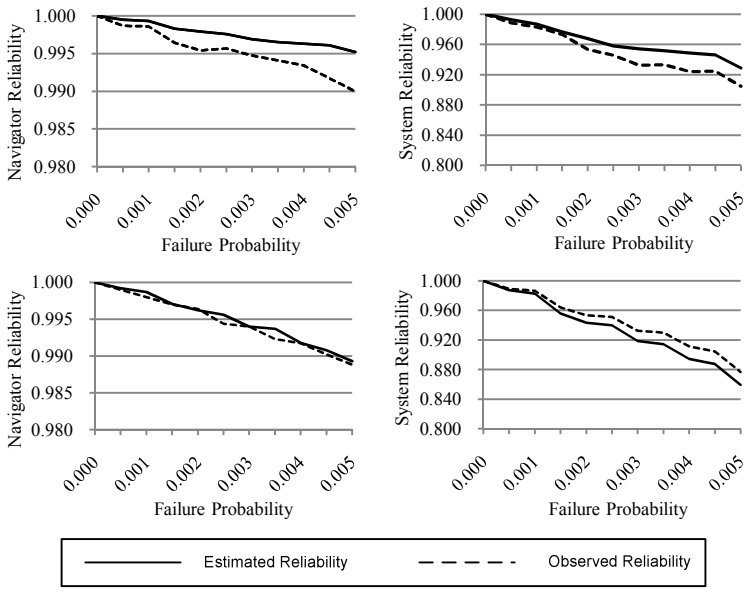

Figure 7. Navigator allocated to a separate process (top) and Navigator sharing a process with the Controller (bottom).

operated correctly for hours before a failure occurs. On the other hand, the black-box approach taken for computing the observed reliability only considers the fact that a response to a request was not received, irrespective of the duration of the successful executions prior to the component's eventual failure.

While it is hard to argue which model is the most accurate indicator of actual reliability, one can conclude that both RESIST's reliability prediction and observed reliability produce very similar trend, which gives us confidence that adaptation decisions based on RESIST's reliability predictions can produce desirable results. Moreover, the sensitive nature of Markov based reliability analysis presents an ideal model for discriminating between the reliability of alternative configurations.

8.2 Impact of Reconfiguration

To evaluate the impact of reconfiguration decisions on system's reliability we used RESIST to estimate both the component and system level reliability for different architectural configurations. As in the previous experiment, we have controlled the usage profile, and fixed the execution context. We have also manually injected defects with a specified probability of failure in the robot's *Navigator* and *Controller* components.

Figure 8 shows the reliability values for three different configurations as the failure probability is increased. In part (a), *Navigator* and *Controller* are allocated to the same process and configuration exhibits the lowest system reliability. Part (b) corresponds to a configuration where the *Navigator* is isolated to its own process. An improvement is observed for each of the component's reliabilities as failures in one component no longer affects the other. As expected, the isolation of components to separate processes also results in an overall improvement in system reliability. Finally, part (c) shows another scenario in which to further improve reliability, the *Navigator* component is replicated, which in turn results in further improvements in system reliability. Note that in contrast to reallocation to separate processes, replication does not impact the components' reliability, but results in a system wide improvement.

8.3 Context-Driven Reconfiguration

In this section we evaluate RESIST by observing its behavior in response to changes in the context. Unlike the previous experiments, here the probability of failure has been fixed, while the contextual effect is varied over time. Specifically, in the *Controller* component, probability of the failure transition from *estimating* state (the state impacted by location context) is fixed at 0.0015. The probability of failure from all other states in both *Navigator* and *Controller* is fixed at 0.001. As mentioned in Section 4, a contextual property with potential impact on the *Controller*'s reliability is physical location. For the purpose of this experiment, we controlled the effects of context by varying the probability of encountering an obstacle on the robot's path, which we refer to as *bump* frequency. In effect, the bump frequency indicates the complexity of the terrain through which the robot will navigate in order to accomplish an assigned task. In the robot's initial configuration, the *Controller* and the *Navigator* share the same process. As a non-functional requirement, the robot is required to maintain a system reliability of 97% throughout its execution.

Figure 9 illustrates a comparison of two instances of robot as it maneuvers a terrain with varying levels of complexity. One instance of the robot uses RESIST for run-time adaptation, while

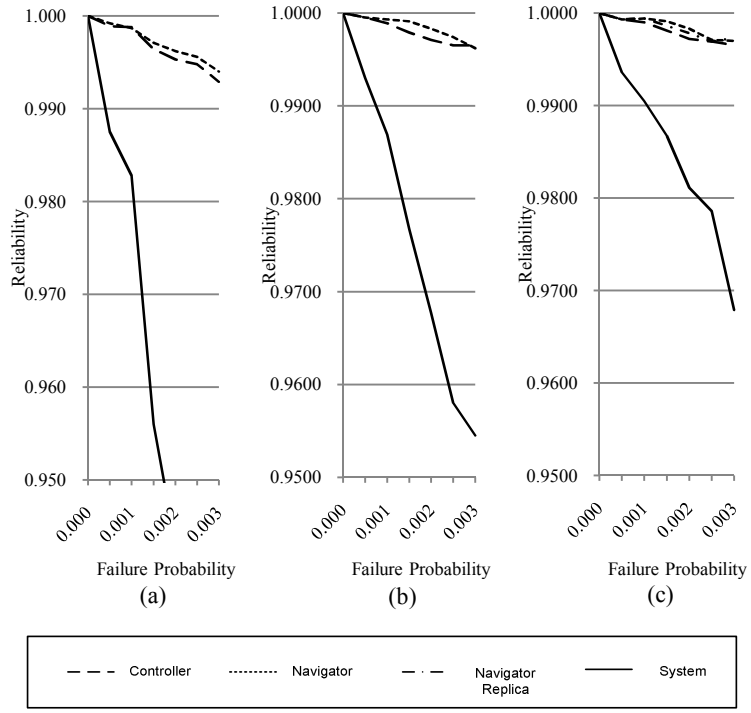

Figure 8. System reliability of different architectures: (a) Navigator and Controller on the same process, (b) Navigator isolated to a dedicated process, and (c) Navigator replicated.

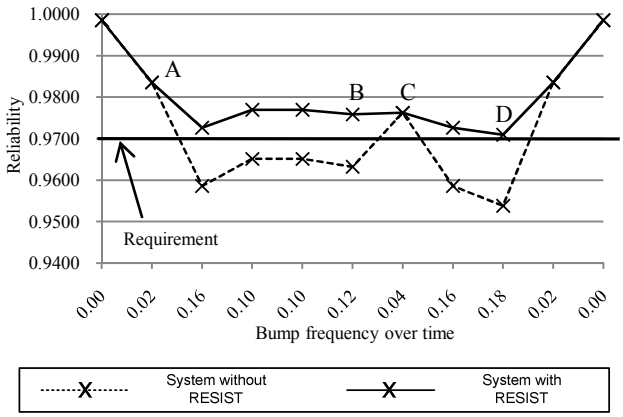

Figure 9. Context-aware proactive reconfiguration.

the other does not. The reliability analysis is performed periodically. Under both settings the robot's reliability degrades as the bump frequency increases. Conversely, a reduction in the bump frequency increases reliability. Recall from the robot *Controller*'s behavioral model in Figure 3 that a *bump* causes the *Controller* to transition into the *estimating* state. The reduction in system reliability can be attributed to the fact that not only there is a high probability of failure in the *estimating* state, but also an increase in bump frequency results in more frequent transitions to *estimating* state.

Figure 9 also shows the proactive reconfiguration process in action. As the robot navigates through the terrain, RESIST predicts the near future reliability of the system by incorporating the complexity of terrain ahead. For instance, at point *A*, RESIST anticipates a drop in reliability by estimating (using the techniques described in [24]) an increase in bump probability due to an increasingly complex terrain for maneuver. Based on anticipation of a rise in bump frequency of up to 16%, RESIST determined a future reliability (indicated by the robot when not using RESIST) of less than 97%. Thus, the robot reconfigures itself at point *A* by reallocating the *Navigator* and the *Controller* into separate processes. This results in an increase in reliability which is maintained until point *B*, at which point in anticipation of easier terrain and for the sake of efficiency RESIST reverts back to the previous configuration. A similar process is repeated at points *C* and *D* until the robot reaches its destination. In light of fluctuations in the context, the instance of robot using RESIST successfully satisfies the reliability requirement.

8.4 Overhead of Reliability Analysis

Since RESIST is intended to manage situated software systems, many of which are mobile and embedded, it is important to assess the performance overhead of RESIST's analysis. Table 1 shows the benchmarking results of RESIST's reliability analysis on an Intel Core 2, 2.4 GHz, 2 GB RAM hardware platform. The results show the time it took for varying number of tasks (i.e., orders sent to the robot). Each task on average resulted in 20 different monitoring observations (e.g., component interface invocations) to be collected. The execution time in the largest scenario, consisting of 2,000 tasks and 41,879 observations took only 10.45 seconds.

Table 1. Execution time of reliability analysis in seconds.

Num. of Tasks	10	50	100	250	500	1000	2000
Num. of Observation	174	1062	1741	5874	9553	20028	41879
Execution Time in Sec.	0.13	0.35	0.69	1.73	2.48	5.10	10.45

9. RELATED WORK

Over the past three decades many software reliability approaches have been proposed. The approaches most relevant to our work are those that consider the system's software architecture [9][10][13][21][22][25][30]. The underlying assumptions in these works make them unsuitable for use in the domain of situated, dynamic, and mobile systems. Majority of these approaches focus on system-level analysis and assume the reliabilities of the software components are fixed and known. Moreover, many of these approaches assume (sometimes implicitly) that the operational profiles of the system are known and do not change at run-time. Finally, none considers the impact of contextual changes on the software system's reliability. Three recent surveys [8][10][11] corroborate our observation.

Our past research has addressed some of the uncertainties associated with design-time reliability analysis by incorporating various sources of information [3][23]. We also identified the challenges of reliability analysis in the mobile domain [14]. Our objective was to provide rough reliability predictions early in the software life-cycle when an implementation of the system is not available. Similar to our previous work, in RESIST we incorporate several sources of information in order to refine initially reliability estimates. In contrast to our previous work, we do this at run-time and rely on a running system to provide RESIST with the latest operational and contextual information.

Few approaches combine software architecture and reliability analysis using run-time data [6][19][29]. While [19] and [29] target traditional and highly predictable software, KAMI framework [6] provides continuous dependability analysis using a model-driven approach. Specifically, KAMI uses run-time data to update the parameters of reliability and performance models (e.g., DTMC). The focus of RESIST has been different from KAMI, as RESIST aims to furnish refined reliability predictions at the component level and to proactively adapt situated software. We believe KAMI and RESIST to be complementary, as the continuous refinement of parameters in KAMI could be utilized in updating RESIST's reliability models.

Also related to this work are the general purpose architecture-based adaptation frameworks [2][7][12]. In contrast to them, RESIST is narrowly aimed at improving the reliability of dynamic situated systems. While none of the existing frameworks directly achieves our objectives, they form the foundation of our research. In fact, our framework is compatible with the widely accepted three layer reference model of self-adaptation [12].

Finally, related is previous research on middleware intended for situated software systems. Aura [27] is an architectural style and supporting middleware for ubiquitous computing applications with a special focus on user mobility, context awareness, and context switching. XMIDDLE [16] is a data-sharing middleware for mobile computing. MobiPADS [1] is a reflective middleware that supports active deployment of augmented services (called mobilets) for mobile computing. Lime [17] is a Java-based middleware that provides a coordination layer that can be exploited for designing applications which exhibit either logical or physical mobility, or both. Unlike RESIST, none of the above technologies provides reliability-driven support for optimization of a mobile software system through adaptation.

10. CONCLUSION

Software systems are increasingly situated in mission critical settings, which present stringent reliability requirements. These systems are predominantly mobile, embedded, and pervasive, which are innately dynamic and unpredictable. In turn, no particular configuration of the system is optimal for the system's entire operational life-time. We presented RESIST, a framework intended to satisfy the reliability requirements, while taking into consideration other quality attributes (e.g., efficiency) through proactive reconfiguration of the software. The three key contributions of RESIST are: (1) incorporation of multiple sources of information, in particular contextual information, to provide refined reliability predictions at run-time; (2) automatically find the optimal architectural configuration that achieves the appropriate-level of tradeoff between reliability and other quality attributes; and (3) proactively adapt the system by positioning it in the optimal configuration before the system's reliability degrades.

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 tradeoffs that RESIST supports. Finally, we plan to investigate the use of other stochastic approaches (e.g., Dynamic Bayesian Networks, and Hierarchical HMM) and potentially an integration with KAMI [6] to support incremental refinement of DTMC parameters, as opposed to periodic assessment of the reliability at run-time.

11. REFERENCES

[1] A. Chan, S. Chuang. MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing. *IEEE Transactions on Software Engineering*, 29(12), December 2003.

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

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

[4] G. Edwards, S. Malek, et al. Scenario-Driven Dynamic Analysis of Distributed Architectures. *Int'l Conf. on Fundamental Approaches to Software Engineering*, Portugal, March 2007.

[5] N. Esfahani, S. Malek, et al. A Modeling language for Activity-Oriented Composition of Service-Oriented Software Systems. *Int. Conf. on Model Driven Engineering Languages and Systems*, Denver, Colorado, Oct 2009.

[6] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model Evolution by Run-Time Parameter Adaptation. *International Conference on Software Engineering*, Vancouver, Canada, May 2008.

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

[8] S. Gokhale, Architecture-Based Software Reliability Analysis: Overview and Limitations. *IEEE Transactions on Dependable and Secure Computing*, 4(1), Jan 2007.

[9] K. Goseva-Popstojanova, et al. Architectural Level Risk Analysis using UML. *IEEE Transactions on Software Engineering*, Vol.29, No.10, Oct 2003.

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

[11] A. Immonen, E. Niemela. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Journal of Software and Systems Modeling*, Jan 2007.

[12] J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. *Int'l Conf. on Software Engineering*, Minneapolis, MN, May 2007.

[13] S. Krishnamurthy, A. Mathur. On the Estimation of Reliability of a Software System Using Reliabilities of its Components. *Int'l Symp. on Software Reliability Engineering*, 1997.

[14] S. Malek, R. Roshandel, et al. Improving the Reliability of Mobile Software Systems through Continuous Analysis and Proactive Reconfiguration. *Int'l Conf. on Software Engineering*, Vancouver, Canada, May 2009.

[15] S. Malek, et al. A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. *IEEE Transactions on Software Engineering*, 31(3), March 2005.

[16] C. Mascolo, et al. XMIDDLE: A Data-Sharing Mid-dleware for Mobile Computing. *International Journal of Personal and Wireless Communications*, Kluwer, vol 21, 2002.

[17] A. L. Murphy, et al. Lime: A Middleware for Physical and Logical Mobility. *Int'l Conf. on Distributed Computing Systems*, Phoenix, Arizona, May 2001.

[18] D. Perry, A. Wolf. Foundations for the Study of Software Architecture. *Software Eng. Notes*, 17(4), October 1992.

[19] F. Popentiu, and P.Sens. A Software Architecture for Monitoring the Reliability in Distributed Systems. *European Safety and Reliability Conf.*, Munich, Germany, Sept 1999.

[20] L. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2), Feb. 1989.

[21] R. Reussner, et al. Reliability prediction for component-based software architectures, *Journal of Systems and Software*, 66(3), 2003.

[22] 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.

[23] R. Roshandel, et al. A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level. *Int. Conf. on Qual. of Soft. Arch.*, Boston, MA, July 2007.

[24] H. Seraji, A. Howard. Behavior-Based Robot Navigation on Challenging Terrain: A Fuzzy Logic Approach. *IEEE Trans. on Robotics and Automation*, vol. 18, no 3, June 2002.

[25] H. Singh, et al. A Bayesian Approach to Reliability Prediction and Assessment of Component Based Systems. *Int. Symposium on Software Reliability Engineering*, 2001.

[26] J. P. Sousa, et al. User Guidance of Resource-Adaptive Systems. *International Conference on Software and Data Technologies*, Porto, Portugal, July 2008.

[27] J. Sousa, D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. *Int'l. Conf. on Software Architecture*, Montreal, Canada, August 2002.

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

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

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