RESISTing Reliability Degradation through Proactive Reconfiguration

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

[1] George Mason University
Department of Computer Science
{dcooray, smalek, ckilgor1}@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. They are 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 runtime 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 intended for mission-critical settings, called *REsilient SItuated SofTware system (RESIST)*, and evaluated it using a mobile emergency response system.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification – *Reliability*; D.2.11 [Software Engineering]: Software Architectures.

General Terms
Design, Reliability

Keywords
Reliability, Software Architecture, Self-Adaptation, Mobility

1. INTRODUCTION
Software systems are fast permeating a variety of domains, including emergency response, industrial automation, navigation, health care, power grid, and civil infrastructure. We call this emerging class of systems *situated software systems*, which are predominantly mobile, embedded, and pervasive. They are characterized by their highly dynamic configuration, unknown operational profile, and fluctuating conditions. At the same time, given the *mission critical* nature of the domains in which they are deployed (e.g., emergency response), majority of situated systems are expected to satisfy stringent reliability requirements.

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 resource utilization 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, the *optimal configuration* for a situated system cannot be determined prior to its deployment, and no particular configuration can be optimal for the system's entire operational lifetime. Thus, runtime reconfiguration of the system may be necessary to achieve the system's maximum potential. Given the mission critical nature of situated systems, we define the optimal configuration as one that satisfies the reliability requirement, while taking into consideration other quality attributes of concern (e.g., resource utilization efficiency, such as memory and CPU usage).

In this paper, we describe and evaluate *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 runtime by incorporating various sources of information. In addition to the architectural models and the monitoring data, RESIST incorporates contextual information 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 show that our reliability predictions are accurate with respect to the *observed* system reliability. We thus consider the predicted reliability as an indicator for decision making. An important contribution of our work is *proactive* adaptation based on our reliability analysis that reconfigures the system at runtime prior to actual reliability degradation. This trait clearly sets our

ASE'10, September 20–24, 2010, Antwerp, Belgium.

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 architectural modeling and analysis environment. Finally, RESIST is evaluated using a robotics emergency response system.

The remainder of this paper is organized as follows. Section 2 presents a motivating example. Section 3 provides a high-level overview of RESIST, while Section 4 presents our failure model. Sections 5 and 6 present the component-level and system-level reliability models, respectively. Section 7 details the configuration selection process. A prototype of RESIST and evaluation of the approach are presented in Sections 8 and 9. An overview of related work and avenues of future research conclude the paper.

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 we focus on the robotic subsystem. A 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 runtime aim at accommodating these constraints.

An example architectural strategy for improving the system's resource utilization efficiency is to 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.

Figures 1a and b show two alternative allocations of the robot's software components to OS processes. Based on the above discussion, from a system's perspective it is reasonable to expect the architecture depicted in Figure 1a to be more efficient in terms of utilization of system's resources, 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., memory and CPU utilization, available battery), and (2) the reliability of the system's constituent components, which as detailed later may vary due to changes in 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 runtime, 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.

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 runtime. RESIST consists of three conceptual software components.

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 runtime to assess a variety of configuration choices and to serve as predictors for the 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

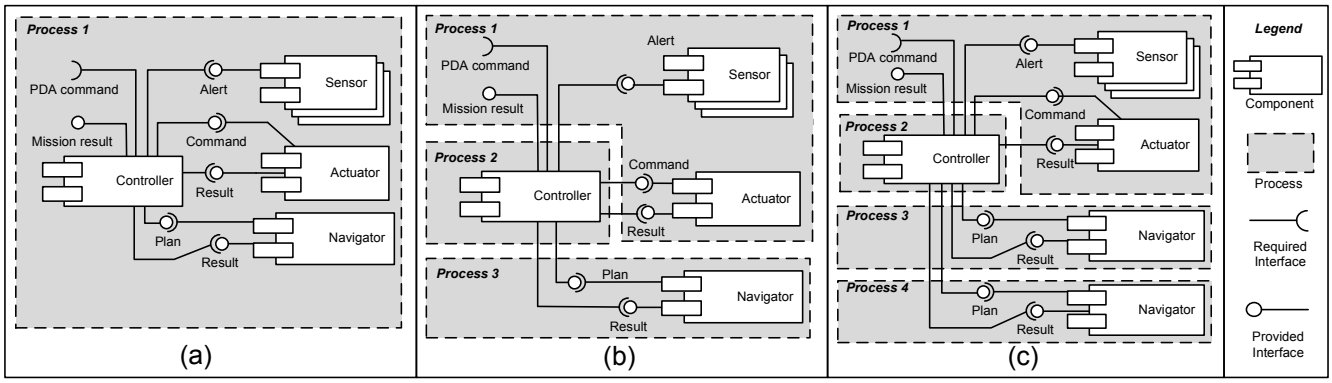

Figure 1. Component-to-process allocation alternatives.

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.

Once a new configuration is selected, the *Context-Aware Middleware* adapts the system at runtime 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 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 [13][23][31] 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.

4. RELIABILITY AND FAILURE MODEL

RESIST estimates *reliability* as the probability that a system performs its required functions under stated conditions for a specified period of time [20]. In situated software systems, given the ongoing changes in system's operational conditions, the reliability may change over time. We consider a *failure* to be an inconsistent behavior of a system with respect to its specification. *Faults are* caused by *defects* (e.g., software or hardware error), and are abnormal conditions that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function. Thus, faults are causes of failures [20].

Consistent with other architecture-based reliability approaches [8][9][10][11] we assume that the occurrence of a failure is stochastic and that components failure model is *fail-stop*. Failures are thus reliably detectable by middleware facilities. Furthermore, failed components are assumed to eventually (automatically or manually) *recover* and resume normal behavior.

We consider two types of failure in RESIST: component and process failures. Component failure is caused by a fault within the component's implementation. Its effects are contained within the boundary of the component except when it causes a process to fail. Process failure occurs when one of the components running as a thread within a process exits prematurely, causing the OS process, including all of the components deployed on it, to fail.

RESIST's reliability model is targeted at distinguishing among alternative architectural configurations, and thus does not consider failures (e.g., wrong results, mismatched data type) that cannot be resolved through architectural means. We assume either such defects are detected during the construction of the system or the failure is contained within the component in which the fault occurred (e.g., through the use of appropriate pre- and post-conditions). While RESIST could be extended to accommodate these additional types of failures, we do not believe such failures could be treated effectively through architectural reconfiguration.

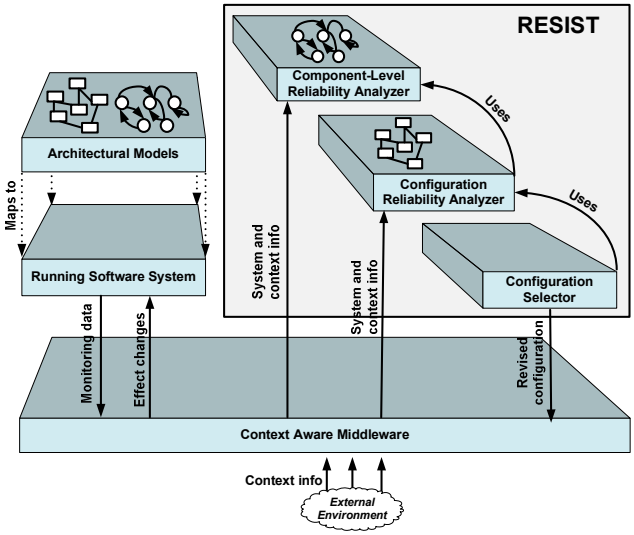

Figure 2. Overview of RESIST framework.

5. COMPONENT-LEVEL ANALYSIS

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 [19]. Architectural models are typically *compositional*: structure and behavior of complex systems are described in terms of their constituent components. Despite this however, as identified by recent surveys [8][10][11], majority of existing architecture-based reliability modeling approaches largely focus on analysis at the system level. Moreover, those approaches that incorporate individual component reliabilities into analysis, assume that component reliabilities are known apriori. Consequently, existing approaches are not suitable for situated systems, where the reliabilities of components and system fluctuate with the *context* in which they are deployed. A purely system-wide analysis offers little help in optimizing the system's architecture in this setting.

5.1 Component Reliability Calculation

Our component-level reliability model relies on dynamic learning techniques, specifically Hidden Markov Models (HMMs) [22], 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 [30]) of *not* being in any failure state. A number of approaches can be taken to ensure tractability if the state space size is determined to be too big [30].

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 runtime 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 runtime 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) [22] can be used to *learn* from runtime 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 come from the component's architectural model (e.g., statechart diagram), while runtime data obtained through monitoring becomes training data for the HMM.

We use the Baum-Welch algorithm [22] to train and solve the HMM. The input to the algorithm is the data obtained from runtime 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* state, it can receive commands from the firefighter's PDA, and when it is in *estimating*, *moving* or *planning* states the robot makes use of other components such as sensors and actuators. From this diagram we can derive the sets:

States $S = \{S_1 \ldots S_4, F\}$ and Observations $O = \{O_1 \ldots O_{11}\}$

where F denotes a common failure state, $S_1 \ldots S_4$ denote behavioral states (*idle, estimating, planning, moving*), and $O_1 \ldots O_{11}$ denote the observations (state transitions).

At runtime, 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*:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.9879 & 0 & 0.0121 \\ 0.5021 & 0 & 0 & 0.4973 & 0.0006 \\ 0 & 0.1421 & 0.8559 & 0 & 0.0019 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The steady state vector obtained from A represents the probability of being in any of the states as the system operates overtime: $[0.1966 \quad 0.2238 \quad 0.3849 \quad 0.1914 \quad 0.0033]$. Here the last column represents the probability of being in a failure state. The *Controller* reliability based on its present runtime context is computed as:

$$R_c = 1 - 0.0033 = 0.9967$$

5.2 Incorporating Context into Analysis

As mentioned earlier, given the dynamism present in situated system's domain, it is critical to incorporate the notion of *context*

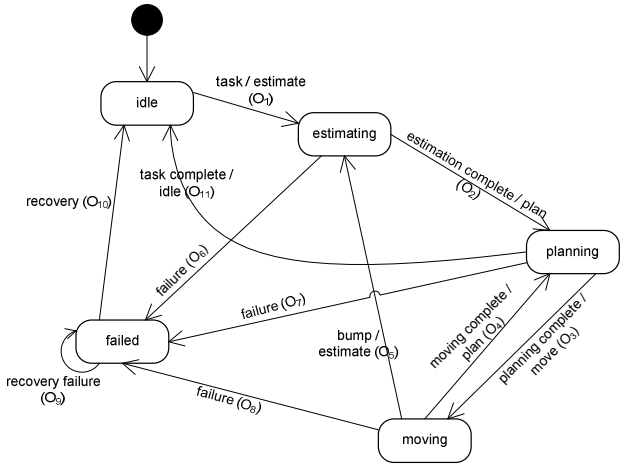

Figure 3. Behavioral model of the robot's Controller component.

into the analysis. Context corresponds to conditions external to the software system, which change the behavior of the system, and hence impact 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 [26] 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 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 a transition probability from state S_k to state S_j in matrix A which is affected by changes in a specific contextual parameter C_n, then $a'_{kj} = \mu(a_{kj}, \Delta C_n)$, where μ is a 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 [26] to update the probability of transitioning from *moving* to *estimating* states based on the complexity of the terrain.

When updating a_{kj} to a'_{kj} the other elements in row k of the matrix A must also be revised to ensure the cumulative probability of all transitions in that row remains at 1, thereby retaining the properties of a stochastic matrix. When revising the transition probabilities in row k, transition probability from state S_k to failed state S_f is unchanged, since the failure probability is independent

of changes in context. The remaining transition probabilities in the row are adjusted proportionately such that:

$$a'_{kj} + a_{kf} + \sum_{m \neq f,j} a'_{km} = 1$$

where a_{kf} is the transition probability from state S_k to failed state S_f, and a'_{km} is a transition probability in row k after proportional adjustment.

6. CONFIGURATION-LEVEL ANALYSIS

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, runtime adaptation becomes necessary to ensure that the system's reliability requirements remain satisfied.

While majority of runtime 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.

6.1 System Reliability Calculation

Our Markov-based system-level reliability estimation approach is based on the model presented by Wang et al. [32], 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 [32]. 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:

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

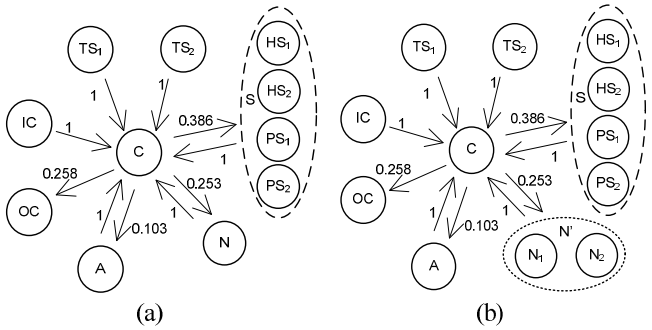

Figure 4. (a) State model for the robot (b) State model with the Navigator replicated.

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* = 0.9967 and *Navigator: N* = 0.9751 using the approach described in Section 5. 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 runtime 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*:

$$M = \begin{array}{c} \\ IC \\ TS_1 \\ TS_2 \\ C \\ S \\ N \\ A \\ OC \end{array} \begin{array}{c} \begin{array}{cccccccc} IC & TS_1 & TS_2 & C & S & N & A & OC \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.2238 & 0.3849 & 0.1913 & 0.1966 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.9751 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Solving the model according to equation (1) yields a system reliability of 0.9385.

6.2 Impact of Architectural Style

Architectural styles are a set of constraints on the structure and behavior of a system to elicit particular desirable qualities [19]. Use of specific architectural styles is a way to apply preconceived solutions to similar recurring software problems. Runtime 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 three processes with the *Navigator* and *Controller* components running on separate OS processes. 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. Recall from section 4 that we have adopted a probabilistic failure model, commonly used in the literature, Here, an underlying assumption is that replicas fail independently. Figure 1c 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 [32]. Hence the probability of state N' executing correctly is 0.9994. Assuming the reliability of all other components and each of the Navigator components to be the same as before, matrix M can be updated where state N is replaced by the new state N', and the matrix element representing the transition from N (now N') to C increases to 0.9994 from 0.9751. Solving the model above according to equation (1) yields a system reliability of 0.9824, which is an improvement of 4.7%.

6.3 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 failure in one component could cause all other components sharing the process to fail. 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, one where the *Controller* and the *Navigator* are deployed as two separate processes and another where the two components are deployed as 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 0.9967 and 0.9751 respectively, the effective reliability of the two components would be $N' = C' = 0.9719$. Intuitively, the drop in the two components' reliability results in a decrease in the overall system reliability. Therefore, the deployment architecture in which the two components are deployed as separate processes yields better configuration reliability.

7. CONFIGURATION SELECTION

The reliability estimation approach presented earlier can be used to determine the most reliable configuration for a situated software system. However, in practice, reliability estimates are used in conjunction with the estimates of other quality attributes (e.g., resource utilization efficiency, response time) to determine the *optimal configuration* for the system. As you may recall, the optimal configuration in RESIST is defined as one that satisfies the system's reliability requirement, while improving other quality attributes of concern. In other words, in RESIST, reliability takes precedence over 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:

$$C^* = argmax_{(C)} \sum_{\forall q \in QualityObjectives} U_q(C) \qquad (2)$$

Subject to $R(C) \geq \delta, \delta \in \mathbb{R}, 0 < \delta \leq 1$

where U_q is a utility function indicating the engineer's preferences for the quality attribute q, R is equation (1) that calculates the expected reliability of a given architecture C as further detailed below. A utility function is used to perform trade-off analysis between competing (conflicting) quality concerns. In the emergency response system, we would need two utility functions: one specifies the user's preference for improvements in *reliability*, while another one specifies the same for resource utilization efficiency. Elicitation of user's preferences is a topic that has been investigated extensively in the literature (e.g., [28]). RESIST does not place a constraint on the format of utility functions. Arguably any user can specify hard constraints, which can be trivially modeled as step-functions. Alternatively, a utility function may take on more advanced forms (e.g., sigmoid curve), and elicited using the techniques in [28].

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

Thus, for a system with t number of software components, where each component's reliability prediction r_i has computed according to the method described in Section 5, and h is the number of processes, an architectural configuration for the aforementioned optimization problem can be formally specified as follows:

- Decision variable $p_i \in \mathbb{Z}^+$ represent the number of replicas for component i
- Decision variable $x_{ij} \in [0,1]$ to indicate if component i is placed on the process j

The configuration is subject to the following constraints:

- Each component *must* be placed on a process: $\forall i \in \{1,..,t\}, \sum_{j=1}^{h} x_{ij} = 1$
- An architectural constraint may be applied to limit the number of replicas allowed for a component: $\forall i \in \{1,..,t\}, p_i \leq w_i,$ where $w \in \mathbb{Z}^+$
- Though a component is allowed to be both replicated and share a process with another component, an architectural constraint is imposed such that they may not both happen simultaneously. This is because replication is most effective (i.e., achieves maximum improvement in reliability) if both the component and its replicas are isolated into separate processes. Thus, we introduce binary variable q_i, which indicates if component i is sharing a process with another component:

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

$$q_i = \begin{cases} 1, & \text{if the } i^{\text{th}} \text{ component shares a process} \\ 0, & \text{if the } i^{\text{th}} \text{ component does not share a process} \end{cases}$$

where $\forall i, k \in \{1, .., t\}$, and;

$$q_i = 1 - \sum_{j=1}^{h} x_{ij} \prod_{k \neq i}^{t} (1 - x_{kj})$$

Thus, the effective reliability of component i is:

$$r_{i_{eff}} = q_i r_{i_{share}} + (1 - q_i)\, r_{i_{rep}}$$

where $r_{i_{share}}$ is the effective reliability of component i when the component shares a process with a different component, and;

$$r_{i_{share}} = \sum_{j=1}^{h} r_i x_{ij} \prod_{k \neq i}^{t} [r_k x_{kj} + (1 - x_{kj})],$$

and $r_{i_{rep}}$ is the effective reliability of component i when the component is replicated with $p_i \geq 0$ number of replicas, and;

$$r_{i_{rep}} = 1 - (1 - r_i)^{1+p_i}$$

The system reliability *R(C)* is computed by mapping the effective reliability $r_{i_{eff}}$ of the components to states as described in equation (1). There are $O(h^t)$ ways of allocating software components to OS processes. The total number of different architectures resulting from the application of fault tolerant style is $O(max\{w_i\}^t)$. Thus, the size of the solution space for this optimization problem is $O((max\{w_i\} \times h)^t)$. Clearly the solution space is large, even for small values of *w*, *h*, and *t*. However, the solution space may be significantly pruned by imposing architectural constraints, such as the limit on the number of replications allowed.

Many commonly available algorithms could be used to solve the above optimization problem. For small problems RESIST finds the optimal solution using Integer Programming Solvers, while for large problems it uses stochastic techniques such as greedy and genetic. The details of these algorithms are outside the scope of this paper.

8. 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 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 5 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

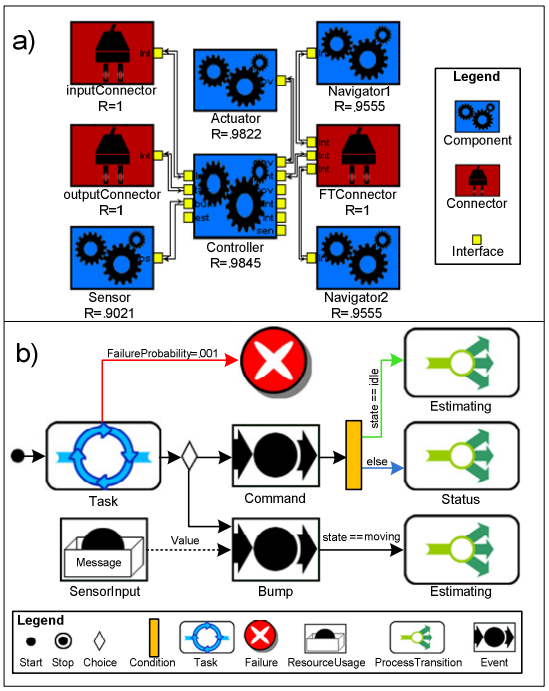

Figure 5. Reliability-annotated architectural models of a portion of robot's Controller component in XTEAM: (a) structural view in xADL, and (b) behavioral view in FSP.

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 middleware, which is integrated with RESIST to facilitate *monitoring* and *adaptation*. Prism-MW's monitoring services provide the runtime 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 described in [15].

9. 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 validity of reliability prediction based on expected changes in the context, (2) the effectiveness of proactive system reconfiguration, and (3) the performance overhead of runtime analysis. We used XTEAM to control the system's operational profile (i.e., usage) and Prism-MW for gathering runtime data. Neither the robotic software nor RESIST was controlled, which allowed them to behave as they would in practice.

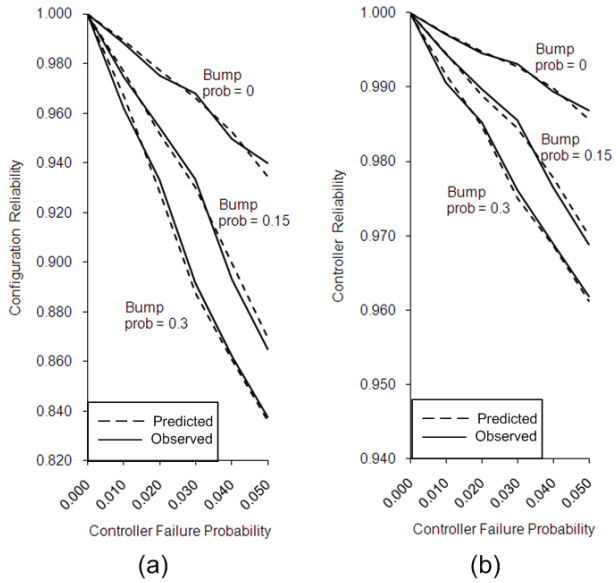

Figure 6. Accuracy of reliability predictions: (a) system reliability (b) Controller's reliability.

9.1 Validity of Reliability Prediction

As described in Section 5, RESIST uses the system's *context* to predict system's near-future reliability by estimating the impact of contextual changes on a components' internal behavior. We have examined the validity of our results by comparing RESIST's predicted reliability values with those estimations obtained from the system's actual behavior. While we have evaluated the validity of our predictions for the entire system, in this section, we present details of the *Controller's* reliability analysis.

For this experiment, we controlled the influence of context by varying the probability of the robot encountering an obstacle on its path, which we refer to as *bump probability*. The bump probability correlates to the *complexity* of the terrain through which the robot navigates in order to accomplish an assigned task. An increase in the bump probability causes the *Controller* to transition from the *moving* state to the *estimating* state with a higher probability (recall Figure 3), thereby altering its operational profile. The techniques presented in [26] together with multi-linear regression were used in our experiments to derive function μ (recall Section 5.2) that estimates the impact of change in terrain to change in bump probability with $\pm$ 2.1% error at 95% confidence level.

In addition to analyzing the effect of context, we varied the failure probability of the *Controller*, specifically the probability of failure from the *estimating* state. We compared RESIST's reliability predictions with the actual observed reliability of the robot during operation. In this experiment, the *Navigator* and the *Controller* were placed in separate processes, and except for the *Controller*, all other components' failure probability was fixed at 0.

Figure 6 shows the comparison of predicted reliability and observed reliability in three execution scenarios where different bump probabilities were predicted, and varied the failure probability of the *Controller* component from 0 to 0.05. As shown, the *Controller*'s reliability decreases as the bump probability increases. This is because an increase in transitions to the *estimating* state leads to more failures. Further, the deviation between observed and predicted reliability both at the level of system and *Controller* are extremely small. Note that since the function μ used in the experiment had a 95% likely error bound of 2.1%, small deviation in results is to be expected. However, the deviation is small enough that very accurate adaptation decisions could be made.

9.2 Proactive Reconfiguration

We evaluate RESIST's ability to satisfy the system's reliability requirement through proactive reconfiguration. We compared an instance of the robot using RESIST against one without RESIST. The failure probabilities of all components in both instances were fixed. We varied the bump probability (effectively changing the context) and observed the proactive reconfiguration process. The robot was required to maintain a system reliability of *at least* 97% throughout its execution, which formed the constraint in our optimization problem.

Initially, the *Navigator* was placed in a separate process, and the other components were placed together in one process. This configuration was based on a design-time analysis of the system that satisfied the reliability requirement and minimized the resource utilization. In order to predict the resource utilization, we used an analytical model that given a configuration of the system predicts its resource demand in terms of memory and CPU utilization. The analytical model considers the number of required OS processes, the number of component replicas, together with the average memory utilization, and the average CPU clock cycles required by each component. The components' memory and CPU utilization estimates were obtained through their design-time benchmarking. Sigmoid curve functions were employed for expressing the user preferences for each of the quality attributes.

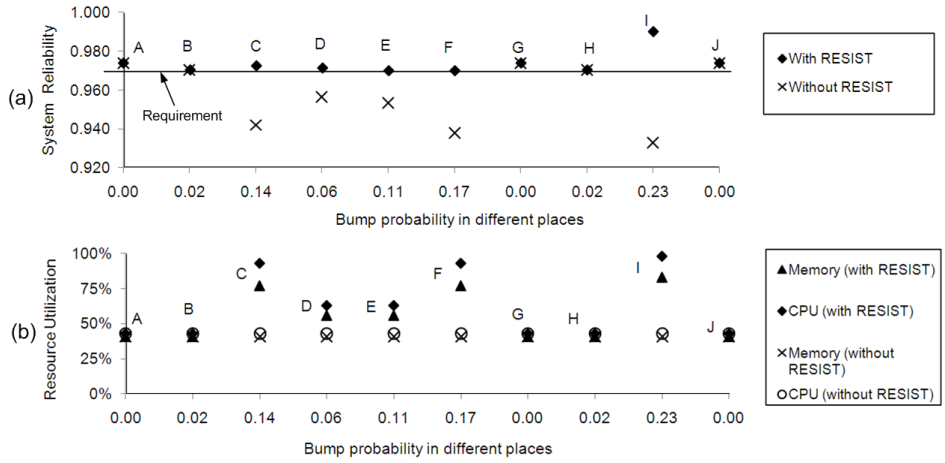

Figure 7. Context-aware proactive reconfiguration. (a) System reliability (b) Resource utilization efficiency.

Figure 7a illustrates the comparison between the two instances of the robot as they maneuver the same area within a building with varying levels of complexity (i.e., obstacles). RESIST predicts the near future reliability of the system as it approaches an area with a complexity that is different from its current location. For instance, as the robot passes point B and before it reaches point C, RESIST anticipates a drop in reliability (since the bump probability increases to 0.14) and proactively adapts the system to maintain its reliability above 97%. As a result, the *Navigator* is replicated and the *Controller* is redeployed to a separate process. This reconfiguration prevents the reliability from dropping below the requirement. In contrast, the reliability of the robot without RESIST deteriorates significantly, falling below the 97% requirement.

Figure 7b shows the effect of reconfiguration on the system's resource utilization efficiency. For instance, at point C both CPU and memory utilization increase significantly due to the addition of the *Navigator* replica and separate processes.

Similarly, RESIST continues to proactively manage the system's configuration. In points F and I, in anticipation of a drop in reliability, RESIST proactively places the system in a more reliable configuration, albeit less efficient. On the other hand, in points D, G, and J, in anticipation of an improvement in reliability, RESIST proactively places the system in a more efficient configuration, while meeting the 97% reliability requirement.

9.3 Overhead of Reliability Analysis

Since RESIST is intended to manage situated software systems at runtime, 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 platform, which is representative of the average hardware capability present in modern mobile robots (e.g., [17]). The results show the time it took for performing the reliability analysis for varying number of *commands* (i.e., tasks sent to the robot). Each command on average resulted in 20 different monitoring observations (e.g., component interface invocations) to be collected and used for training the HMM. The benchmark in the largest scenario, consisting of 2,000 commands and 41,879 observations took 10.45 seconds. However, in practice, our experience with the emergency response robot shows the analysis is often performed on much smaller number of observations, requiring only a fraction of a second for completion.

10. 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][23][24][27][32]. The underlying assumptions in these approaches 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 profile of the system is known and does not change at runtime. Finally, none considers the impact of contextual change on the software system's reliability. Three recent surveys [8][10][11] corroborate these observations.

Table 1. Execution time of reliability analysis in seconds.

Num. of Commands	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

Our past research has addressed some of the uncertainties associated with design-time reliability analysis by incorporating various sources of information [3][25]. 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. In contrast to our previous work, here we are concerned with runtime reliability of the system and rely on the availability of its implementation. Moreover, we incorporate latest operational and contextual information to predict the system's reliability and proactively place it in the optimal configuration.

Few approaches combine software architecture and reliability analysis using runtime data [6][20][31]. While [20] and [31] target traditional and highly predictable software, KAMI framework [6] provides continuous dependability analysis using a model-driven approach. Specifically, KAMI uses runtime data to update the *parameters* of reliability and performance models. The focus of RESIST has been different from KAMI. KAMI reactively adjusts the system's models, while RESIST proactively predicts near future reliability of the system. Moreover, unlike KAMI, RESIST furnishes reliability predictions at the component level. 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.

Related to our 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 [29] 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 [18] 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 situated software systems through proactive adaptation.

11. 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 runtime; (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 runtime.

12. ACKNOWLEDGMENTS

This work is supported in part by grants CCF-0937472 and CCF-0820060 from the National Science Foundation.

13. REFERENCES

[1] A. Chan, et al. MobiPADS: Reflective Middleware for Context-Aware Mobile Computing. IEEE *TSE*, 29(12), Dec. 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. *ICSE*, 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, et al. Model Evolution by Run-Time Parameter Adaptation. *ICSE*, 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 TSE*, Vol.29, No.10, Oct 2003.

[10] K. Goseva-Popstojanova, et al., Architecture-Based Approaches to Software Reliability Prediction. *Int'l. 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. *ICSE*, 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, et al. Improving the Reliability of Mobile Software Systems through Continuous Analysis and Proactive Reconfiguration. *ICSE*, 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] Mobile Robots Inc. http://www.mobilerobots.com/

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

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

[20] H. Pham, Software Reliability, *Springer,* 2002.

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

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

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

[24] G. Rodrigues, et al. Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems. *In'l Conf. on Fundamental Approaches to Software Engineering*, Edinburgh, UK, April 2005.

[25] 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.

[26] 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.

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

[28] J. P. Sousa, et al. User Guidance of Resource-Adaptive Systems. *Int'l Conf. on Software and Data Technologies*, Porto, Portugal, July 2008.

[29] 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.

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

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

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