Proactive Self-Adaptation for Improving the Reliability of Mission-Critical, Embedded, and Mobile Software

Deshan Cooray, Ehsan Kouroshfar, Sam Malek, *Member*, *IEEE*, and Roshanak Roshandel, *Member*, *IEEE*

Abstract—Embedded and mobile software systems are marked with a high degree of unpredictability and dynamism in the execution context. At the same time, such systems are often mission-critical, meaning that they need to satisfy strict reliability requirements. Most current software reliability analysis approaches are not suitable for these types of software systems, as they do not take the changes in the execution context of the system into account. We propose an approach geared to such systems which continuously furnishes refined reliability predictions at runtime by incorporating various sources of information, including the execution context of the system. The reliability predictions are leveraged to proactively place the software in the (near-)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 number of component replicas. 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.

Index Terms—Context awareness, software architecture, self-adaptive systems, reliability, 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. These systems 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 these systems are expected to satisfy stringent reliability requirements.

Engineers of a mobile and embedded software system typically spend significant effort to determine a good configuration for the system that ensures its functional and nonfunctional requirements. For instance, they may perform a tradeoff 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* [3].

Due to variability in the execution context, the *optimal configuration* for a mobile and embedded 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 many mobile and embedded systems, we define a good configuration as one that satisfies the reliability requirement, while taking into consideration other quality attributes of concern (e.g., efficiency).

In this paper, we describe and evaluate *REsilient SItuated SofTware system (RESIST)*, a framework intended to improve the reliability of embedded and mobile software systems. RESIST continuously analyses and dynamically adapts the software to deal with changes in the execution context that could degrade its reliability.

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.

- D. Cooray is with VeriSign, Inc., Reston, VA 20190. E-mail: dcooray@gmu.edu.
- E. Kouroshfar and S. Malek are with the Computer Science Department, George Mason University, Fairfax, VA 22030. E-mail: {ekourosh, smalek}@gmu.edu.
- R. Roshandel is with the Department of Computer Science and Software Engineering, Seattle University, Seattle, WA 98122. E-mail: roshanak@seattleu.edu.

Manuscript received 6 May 2012; revised 8 Mar. 2013; accepted 28 July 2013; published online 2 Aug. 2013.
Recommended for acceptance by S. Clarke.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2012-05-0116.
Digital Object Identifier no. 10.1109/TSE.2013.36.

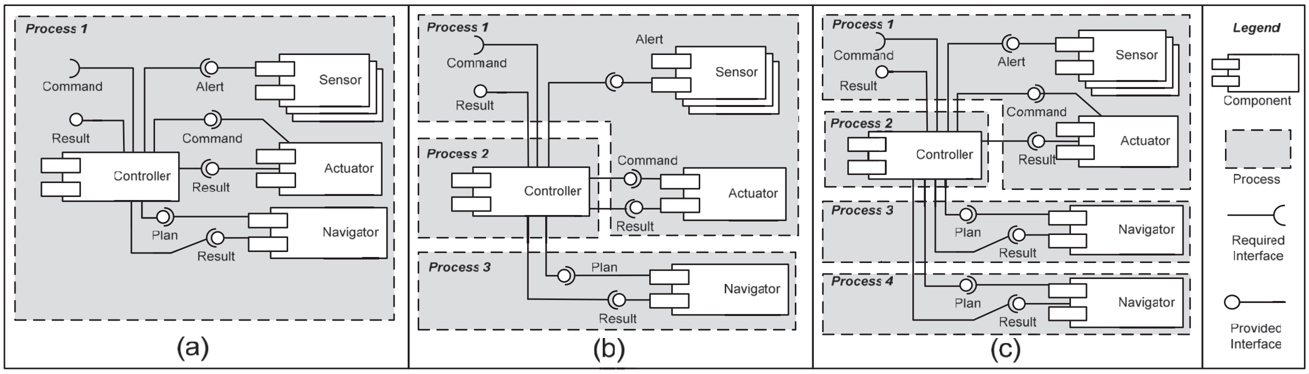

Fig. 1. Component-to-process allocation alternatives: (a) All components allocated to the same process, (b) Controller and Navigator allocated to separate processes, and (c) Controller allocated to separate process, and the Navigator is replicated and placed in separate processes.

RESIST uses the reliability predictions to 1) proactively determine when the system should be adapted, and 2) find the (near-)optimal configuration for the near future operation of the system. Our evaluation shows that our reliability predictions are accurate with respect to the *observed* system reliability. We, thus, use the predicted reliability as an indicator for making adaptation decisions. 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 work apart from the majority of existing self-adaptive frameworks that are *reactive* in their decision making [4], [21].

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.

RESIST was originally introduced in our prior work [8]. Beyond a comprehensive and significantly more detailed description of RESIST, this paper reports on several new contributions and experiments. First, we formally specify the notion of context and describe its impact on both component and system reliability prediction in RESIST. Second, we present *RESISTER*, a novel heuristic-based algorithm for finding a near-optimal configuration that satisfies the system's reliability requirement, while taking into consideration its efficiency. Third, we report on several new experiments and evaluation results, which demonstrate the accuracy and performance of RESIST.

The remainder of this paper is organized as follows: Section 2 presents a motivating software system used to describe and evaluate the research. Section 3 describes the impact of context on the system's architecture and subsequently its reliability predictions. Section 4 introduces RESIST by providing a high-level overview of its components. Section 5 defines our failure model and the assumptions underlying this research. Section 6 presents the component-level and configuration-level reliability estimation techniques. Section 7 describes the reconfiguration tactics for improving the system's reliability. Section 8 defines the configuration selection problem and an algorithm targeted at that. Sections 9 and 10 present a prototype implementation of RESIST and its evaluation, respectively. Section 11 provides an explicit discussion of threats to validity. An overview of the 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 a previous collaboration with a government agency [12], [25]. 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 smartphones capable of controlling the robots and sensors, and mobile *robots* that execute the commands received from the firefighters.

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 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.

Figs. 1a and 1b show two alternative allocations of the robot's software components to OS processes. Although not depicted in the figure, the connectors mediating the component interaction in these two architectures are different. In Fig. 1a, local interactions occur via shared data structures, while in Fig. 1b interprocess communication occurs via remote method invocations. Therefore, the two architectures differ in terms of both allocation and component-connector viewpoints [7].

Based on the above discussion, from a system's perspective it is reasonable to expect the architecture depicted in Fig. 1a to be more efficient, while the one depicted in Fig. 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 a 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 example system in the remainder of the paper to describe and evaluate our approach.

3 Impact of Context on Architecture

Any type of information that characterizes the runtime conditions of the system and alters the system's behavior can be considered as its context [1]. A system's context may consist of a representation of several different aspects of its changing execution environment that could potentially impact the behavior and properties of a system. Among them three main categories of context can be identified [1], [46]:

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

A context-aware system uses knowledge about its context to provide relevant information and/or services to its user [1]. While in some systems contextual information is directly used to provide services to the user, in others contextual information is used to optimize the manner in which services are provided to the user. For example, a GPS-enabled mobile phone, which displays a map based on the user's location, considers the location as an input to the service that is provided. In contrast, a mobile robot engaged in firefighting may need to reconfigure itself (e.g., its architecture) depending on its contextual characteristics so that its dependability is optimal with respect to other quality attributes such as resource usage. As described in the next section, RESIST is aimed at the second class of systems. Specifically, RESIST uses the system's context to perform architectural reconfiguration of the system in response to anticipated changes in its reliability.

Changes to the operational context of a system impact its runtime behavior, which in turn could potentially impact aspects of system's quality, such as reliability. In *architecture-based adaptation* [21], [34] the system's software

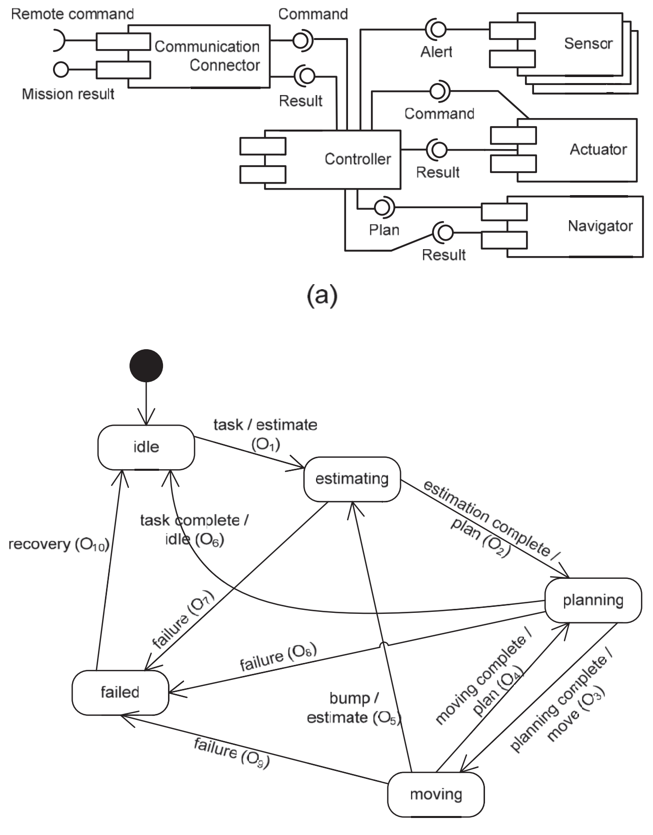

Fig. 2. The robot's architecture: (a) the robot's structural model, and (b) the behavioral model of the robot's Controller.

architecture forms the basis for adaptation reasoning. Consequently, we argue that it is important to be able to model the effect of changes in the context on a system's architecture as a first class entity. In our work, we adopt a broad interpretation of the system's *architecture*, which simply captures the knowledge about the system. This *knowledge* includes many different aspects of the system, including the principal design decisions about the system, its structure and behavioral models, as well as the execution properties of the system captured in the form of an operational profile model.

To exemplify the effect of context on a system's architecture, below we present how the mobile nature of a robotic system introduces contextual changes that can impact its operational profile, and in turn its reliability. Fig. 2a shows the architectural models of the mobile robot. It receives a command from an external system (i.e., a smartphone) and returns the result of executing the command. Upon receiving a command, it uses its Sensor component to gather data about its environment, such as nearby obstacles and proximity to heat, and determines a plan and subsequently executes it using its Navigator and Actuator components, respectively. Fig. 2b shows the robot's Controller component's behavioral model in the form of a UML state chart. It includes behavioral states *idle*, *estimating*, *planning*, and *moving*, during which the Controller invokes interactions with the other components in the system (i.e., Sensors, Actuator, Navigator, etc.). The *failed* state denotes a common failure state of the component.

Transitions O_1 to O_6 denote behavioral transitions resulting from input events such as interface calls on the component. Transitions O_7 to O_9 denote a failure that may arise under some circumstances. Such failures are caused by faults in the software that could lead to a failure. Transition O_{10} denotes eventual recovery of the component as a result of automatic or manual reinitialization of the component.

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

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

We model the changes in context and its effect on the system's architecture as follows:

- a set of contextual parameters $C = \{C_1, \ldots, C_l\}$ representing the information about a system's context that may impact the system,
- a set of architectural parameters $A = \{A_1, \ldots, A_m\}$ representing the architectural properties that are amenable to change as a result of variations in the system's context,
- a set of interactions $I = \{I_1, \ldots, I_n\}$ between contextual and architectural parameters, where in each interaction, one or more contextual parameters instigates a change in an architectural parameter,
- a set of functions representing the effect of interactions.

These functions can be defined as follows:

$$\text{Given } I_i \in I, \qquad \mu_i : \mathcal{P}(C) \to A, \tag{1}$$

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

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

4 Approach Overview

Fig. 3 provides an overview of our approach in light of Kramer and Magee's three-layer reference model of self-management [21]. RESIST corresponds to the top layer (i.e., *Goal Management* layer), which is the focus of this paper. However, RESIST also relies on the presence of an underlying implementation platform and runtime support (i.e., *Change Management* and *Component Control* layers) for runtime monitoring and adaptation of the software system. We first provide an overview of the RESIST's components. We then describe the relationship between RESIST and the required capabilities in the bottom two layers.

4.1 RESIST (Goal Management Layer)

RESIST's goal is to maintain a configuration for a software system that satisfies a user specified reliability requirement (e.g., system reliability should be greater than 99 percent), while minimizing its resource usage. To that end, and as shown in Fig. 3, RESIST assumes two inputs from the user: 1) a reliability constraint representing the system-level reliability requirement, and 2) one or more *utility functions* indicating the acceptable tradeoffs with respect to resource usage.

RESIST is comprised of three conceptual components: *Component Reliability Analyzer, Configuration Reliability Analyzer*, and *Configuration Selector*.

Architecture-based reliability models along with contextual and monitoring information obtained from the system are used by the *Component 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 the current architectural configuration. If the current configuration is deemed to be unsatisfactory (e.g., does not satisfy the required reliability), the *Configuration Selector* is then invoked to find a suitable configuration for the near future operation of the system.

To find an architecture that satisfies the reliability requirements, *Configuration Selector* generates one or more *Alternative Architecture*, and uses the *Configuration Reliability Analyzer* to predict the reliability of each. In tandem with reliability predictions, it may use estimates of resource usage in the selection process. Note, however, the process for obtaining and estimating resource usage is beyond the scope of this paper.

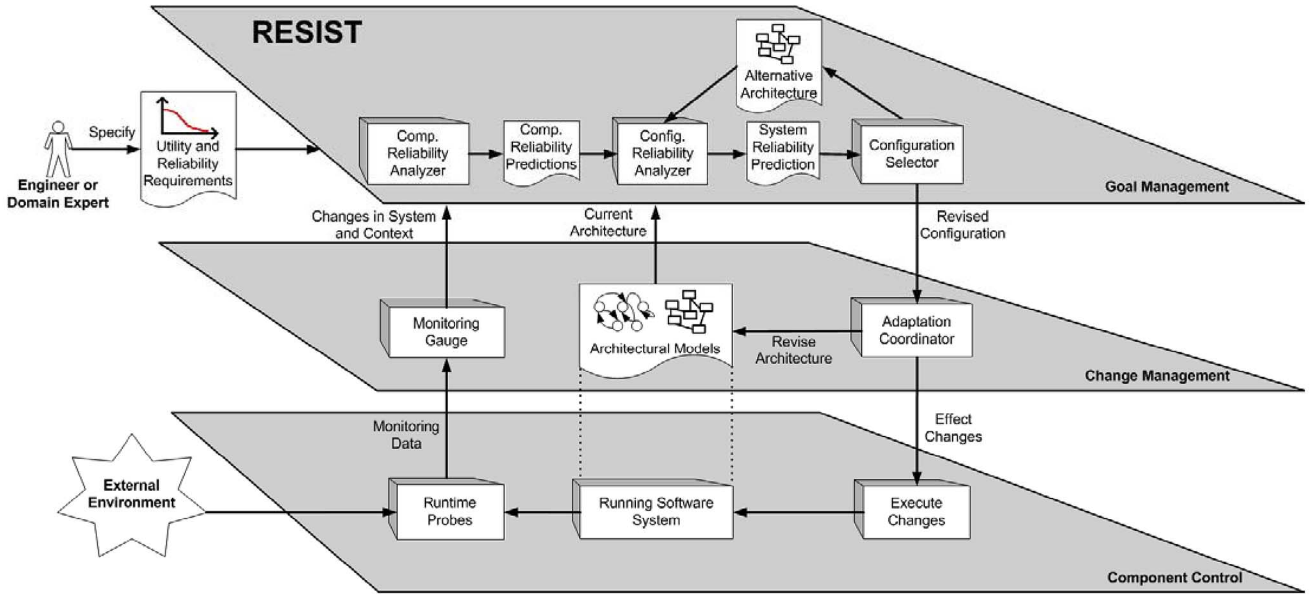

Fig. 3. Overview of RESIST framework, which is organized as a feedback control loop that continuously monitors, analyzes, and adapts the system at runtime.

If a new configuration is found to be superior to the current architecture, the revised configuration is passed to the *Change Management* layer to adapt the system.

4.2 Change Management Layer

The *Change Management* layer is responsible for ensuring orderly execution of changes in the system. To that end, this layer is responsible for maintaining an updated architectural representation of the running software system. As shown in the *Change Management* layer of Fig. 3, architectural models form the centerpiece of our approach, and 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 are annotated with contextual properties necessary for reliability analysis of mobile and embedded systems.

In addition, the *Adaptation Coordinator* of *Change Management* layer is responsible for determining the order in which changes should be effected in the software to avoid failures. For instance, the *Change Management* layer may devise a sequence of steps to ensure a software component is in the *quiescence* state [22] before it is replaced.

Finally, the *Monitoring Gauge* of the *Change Management* layer would process the raw monitored data collected from the *Runtime Probes* (discussed in the next section). For example, one such processing is to calculate the running average of collected data to make the approach resilient to anomalous spikes.

We have previously developed an architectural modeling and analysis environment, called XTEAM [10], that has been integrated with Prism-MW [24] to ensure the models and the running software are synchronized. We have used this setup in our implementation and experiments. Other researchers (e.g., Oreizy et al. [34] and Garlan et al. [13]) have also developed capabilities similar to the two bottom layers of Fig. 3. As a result, RESIST is not necessarily specific to our implementations of *Change Management* and *Component Control*, and in principle could be ported to work on top of these other environments.

4.3 Component Control Layer

RESIST is applicable to software systems where the internal and contextual properties of the software can be monitored and its architecture can be dynamically adapted. The *Component Control* layer, at the very bottom of Fig. 3, depicts these capabilities. We have previously developed a middleware platform, called Prism-MW [24], that provides such capabilities and used extensively in the implementation and evaluation of this research.

The middleware provides the ability to make changes to the software in terms of architectural operations (e.g., adding/removing components and connectors, changing the architecture's topology). The middleware uses various *Runtime Probes* to monitor the software system for information that is used to refine the reliability predictions. This information is obtained from multiple sources, both internal to the software (e.g., frequency of failures and exceptions, changes in workload and service requests) as well as external/contextual to the software (e.g., available network bandwidth and battery charge, changes in physical location).

Since the monitored data represent the most recent operational, structural, and contextual profile of the system's execution, it can be used to assess the system reliability more accurately than what would be possible at design-time. Note that unlike previous approaches [23], [41], [51], we do not rely solely on monitoring the system's failure. Instead, we incorporate architectural knowledge, monitoring data, and contextual changes at runtime in a complementary fashion to produce more accurate results that are indicative of the system's near future reliability.

In summary, the overall process depicted in Fig. 3 is organized as a feedback control loop that continuously monitors, analyzes, and adapts the system at runtime. In terms of the feedback control loop concepts, RESIST corresponds to the *controller* of the system, user specified reliability requirement corresponds to the *set point*, and RESIST assumes the availability of sensors that can measure the *input variables* (e.g., variations in the context) and the existence of actuators that can change the *manipulated variables* (i.e., the configuration of the architecture).

5 Reliability and Failure Definitions

RESIST estimates *reliability* as the probability that a system performs its required functions under stated conditions for a specified period of time [36]. In embedded and mobile software systems, given the ongoing changes in a 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 [36].

Consistent with other architecture-based reliability approaches [14], [15], [16], [17], we assume that the occurrence of failure is stochastic and that component 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 postconditions). 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.

6 Reliability Analysis and Prediction

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 [35]. Architectural models are typically *compositional*: structure and behavior of complex systems are described in terms of their constituent components and their interactions. Despite this, however, as identified by recent surveys [14], [16], [17], majority of existing architecture-based reliability modeling approaches largely focus on analysis at the system level alone. Moreover, those approaches that incorporate individual component reliabilities into analysis assume that component reliabilities are known a priori. Consequently, existing approaches are not suitable for mobile and embedded systems, where the reliabilities of components and system fluctuate with the *context* in which they operate. A purely coarse-grained system-wide analysis offers little help in optimizing the system's architecture in this setting. As described in Section 3, reliability analysis must be performed by considering behavioral changes within components, as well as changes in the interactions among components and their operational context.

Therefore, as shown in Fig. 3, RESIST performs the reliability analysis at two levels: at component level and subsequently at configuration level. At both levels, architecture-based reliability techniques are used in conjunction with monitoring information obtained from the system and its context. Architecture-aware reliability analysis enables architecture-based adaptation techniques to be utilized to improve or maintain the system's reliability. Moreover, since the context impacts both the internal behavior of components and the interactions among them, RESIST incorporates context information into the reliability analysis at both component and configuration level.

To perform reliability analysis and prediction, RESIST considers the software operational profile (SOP) [36] of both components and the system. This enables RESIST to quantify components' and system's behavioral properties that affect the overall reliability. SOP represents the set of runtime events occurring during system execution along with the probabilities with which they occur in a given environment. As described in Section 3, the probabilities in the SOP may be affected by changes in the system's context. Therefore in this case, we consider SOP's probabilistic values as relevant architectural parameters.

For the purpose of modeling the SOP of components and the system, we use Discrete Time Markov Chains (DTMC) [18]. A DTMC is defined as a stochastic process with a set of states $S = \{S_1, S_2, \dots, S_n\}$ and a transition matrix $A = \{a_{ij}\}$, where a_{ij} is the probability of transitioning from state S_i to state S_j Ultimately, our goal is to use the DTMCs representing software components' SOP and overall system's SOP, incorporate context information and predictions about anticipated operational profile, and perform reliability predictions first for individual components and subsequently for the system as a whole. However, obtaining the DTMC directly from observed runtime events poses challenges, particularly when working with complex models. These challenges are rooted in the fact that a sequence of observed events may not always have a one-to-one mapping to a sequence of states in the Markov model.

We thus rely on Hidden Markov Model (HMM) methodology where such an assumption is loosened and appropriate algorithms have been developed to obtain the DTMC [39]. Specifically, we use the Baum-Welch algorithm [39] to learn from runtime data and obtain matrix A values.

Formally, an HMM is defined by a set of states $S = \{S_1, S_2, \dots, S_n\}$, a transition matrix $A = \{a_{ij}\}$ representing the probabilities of transitions between states, a set of observations $O = \{O_1, O_2, \dots, O_m\}$, and an observation probability matrix $E = \{e_{ik}\}$ that represents the probability of observing event O_k in state S_i. The sets S and O of the

HMM come from architectural models of the system, while runtime data obtained through monitoring become training data for the HMM. Note that in this case, a sequence of observations obtained at runtime does not deterministically map to a sequence of states.

The Baum-Welch algorithm [39] is then used to train the model based on the runtime data, solve the HMM, and obtain the matrix A. The training data used as input to this algorithm consist of sequences of observations. Given an initial HMM constructed as described above, the Baum-Welch algorithm converges on the transition matrix A. We use this technique to derive the SOP for both components and the system. In the following sections, we elaborate on the techniques used to estimate the SOP and how they are used in predicting the reliability of components and configurations.

6.1 Component Reliability Analysis

In the case of component reliability, the states (i.e., set S) and observations (i.e., set O) are identified using the component's behavioral model, such as the state chart diagram depicted in Fig. 2b. For example, for the robot's Controller, we can obtain the following:

$$S = \{S_1, S_2, S_3, S_4, S_f\}, \text{and } O = \{O_1, \ldots, O_{10}\},$$

where states $S_1, \ldots, S_4$ represent the behavioral states: *idle*, *estimating*, *planning*, and *moving*, state S_f represents the common *failed* state, and observations $O_1, \ldots, O_{10}$ represent the transitions between states as shown in Fig. 2b. At runtime, the component 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 SOP of the component based on the training data, which represents the component's behavior based on its *current* context.

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

$$A_{Controller} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.932 & 0 & 0.068 \\ 0.049 & 0 & 0 & 0.947 & 0.004 \\ 0 & 0 & 0.99 & 0 & 0.01 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

This represents the Controller's operational profile based on its present context. To compute its reliability, we obtain the *steady-state* vector of the above transition matrix, from which we determine the probability of not being in failure state S_f. Intuitively, a *steady state* indicates the probability of a stochastic phenomenon modeled using a Markov model being in that state [18]. The steady-state vector for the matrix $A_{Controller}$ is

$$[0.0319 \quad 0.0319 \quad 0.4763 \quad 0.4511 \quad 0.0085].$$

Here, the last column represents the probability of being in the failed state. The Controller's reliability based on its present context can then be computed as follows:

$$R_{Controller} = 1 - 0.0085 = 0.9915.$$

6.2 Context-Aware Component Reliability Prediction

An important contribution of our research is the incorporation of contextual knowledge in arriving at reliability predictions, which enables proactive reconfiguration of the software. To arrive at a reliability prediction for a component, RESIST utilizes information from the emerging context to determine the behavioral changes that can occur in the near future operation of the component. This is performed by considering the changes that occur in the component's SOP as a result of the anticipated contextual changes. To determine the *future* SOP of the component, the transition probabilities in the SOP are updated by utilizing functions of the form of (1), which captures the impact of context on architectural parameters. In this case, the architectural parameters are the transition probabilities between states in the component's SOP.

Thus, let us consider a point of interaction $I_n \in I$, where architectural parameters represented by transition probabilities in transition probability matrix A are impacted by the system's contextual parameter C, and where the impact on each architectural parameter is given by function μ_n. The component's *future* SOP is derived based on the transition probabilities of the *present* SOP, by applying the following three rules $\forall I_n \in I$:

- The transition probability a_{ij} from state S_i to S_j that is impacted as a result of I_n is revised such that the updated value a'_{ij} is given by
$$a'_{ij} = \mu_n(\mathcal{P}(\mathcal{C})). \qquad (2)$$
For example in Fig. 2b, the transition probability from *moving* to *estimating* state is directly impacted by the navigational complexity of the robot's environment. Here, μ_n is a function that correlates the navigational complexity (i.e., a contextual parameter) to the transition probability from *moving* to *estimating* state.
- Given the changed a_{ij}, the transition probability a_{if} from state S_i to failure state S_f remains unchanged. This is because the probability of failure while the component is in state S_i is independent of the contextual changes that cause transitions to state S_j. For example in Fig. 2b, the transition probability from *estimating* to *failed* state does not change as the transition probability from *moving* to *estimating* state changes.
- Given the changed a_{ij}, the remaining transition probabilities in row i of the transition probability matrix A are updated so that the cumulative probability of all transition probabilities in each row remains at 1. Intuitively, a change in an operational profile in terms of increased probability of one activity will result in a decrease in probability of other activities, given that the total probability of all activities is capped at 1. The update is necessary to maintain A as a valid stochastic matrix. For example in Fig. 2b, as a result of an increase in the transition probability from *moving* to *estimating* state, the probability from *moving* to *planning* state will

decrease. Thus, all transition probabilities a_{ik} in row i excluding a_{ij} and a_{if} are adjusted so that

$$a'_{ik} = a_{ik} \times \left(1 - \frac{a'_{ij} - a_{ij}}{\sum_{k \neq j,f} a_{ik}}\right). \tag{3}$$

To illustrate, consider the following function μ_{bump} that quantifies the transition probability a_{42} from *moving* to *estimating* state in the robot's Controller with respect to the navigational complexity of the robot's physical context given by c:

$$\mu_{bump}(c) = \begin{cases} 0.8196c + 0.00063, & 0 \leq c \leq 0.3 \\ 0.7144c + 0.06916, & 0.3 < c \leq 0.65 \\ 0.6094c + 0.13296, & 0.65 < c \leq 1. \end{cases}$$

In this case, the robot periodically takes snapshots of the environment and using existing techniques [45] determines the complexity of the terrain, which correlates with the probability of encountering an obstacle in its path. This forms the contextual parameter c. 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 relevant parameters in the SOP are updated to reflect the contextual change. For example, if the navigational complexity of the terrain is anticipated to increase, the transition probability a_{42} in the matrix is updated by computing $\mu_{bump}(c)$ for the relevant value of c.

Assuming that the terrain complexity c is expected to increase to 0.45, we can update the new transition probability a_{42} based on $\mu_{bump}(c)$ above, and adjust the remaining elements in the row based on the rules presented above to obtain the following SOP for the Controller:

$$A'_{Controller} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.932 & 0 & 0.068 \\ 0.049 & 0 & 0 & 0.947 & 0.004 \\ 0 & 0.3906 & 0.5994 & 0 & 0.01 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

As before, by computing the steady-state vector, we can derive the reliability of the component from its expected *future* SOP, which in this example results in a decreased reliability of 0.9826.

6.3 Configuration Reliability Analysis

Once the reliability predictions for all components have been obtained, a compositional model is used to predict 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 the 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 describe the configuration-level reliability analysis approach.

Our Markov-based configuration-level reliability estimation approach is based on the model presented by Wang et al. [52], where a system's 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. A *state* s_i maps to one or more components in concurrent execution whose completion is required to transfer control over to the next state. A *state transition* with a probability P_{ij} represents the probability of undergoing a transition from state s_i to state s_j. Accordingly, system reliability R is computed as

$$R = (-1)^{k+1} R_k \frac{|E|}{|I - M|}, \tag{4}$$

where M is a $k \times k$ matrix in which s_1 is the entry state and s_k is the exit state and whose elements are computed as follows:

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

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 $(I - M)$.

This reliability model utilizes information from the system's SOP to derive the reliability for a configuration. Specifically, it requires the transition probabilities between the states (i.e., P_{ij}). At the same time, as described in Section 3, transition probabilities of the SOP are dependent on the context in which the system operates. Thus, RESIST monitors the system at runtime to obtain observations that correspond to interactions between components to derive transition probabilities between states required by the model presented in (4). To derive these transition probabilities, an HMM is trained using the Baum-Welch algorithm using the observations obtained at the system level.

To construct the HMM for the system's SOP, RESIST utilizes the system's structural model, such as the one depicted in Fig. 2a for the robot. In this scenario, a fireman interacts with the robot using a smartphone. The fireman issues a high-level command (e.g., go to a particular waypoint), which is received by the robot's *Controller* through the *Communication Connector (CC)*. The *Controller* executes the appropriate sequence of intermediate actions, resulting in the successful completion of (or inability to complete) the original command, which is sent back to the smartphone through the *Communication Connector*. To complete the task, the *Controller* makes use of a variety of *Sensors*, which detect obstacles and heat, a *Navigator*, which performs planning for the command being executed, and an *Actuator*, which carries out the mechanical activities.

The state model in Fig. 4a depicts the components in the system mapped to states, and control flow interactions among the components are depicted as transitions between states. As shown, each of the components *CC*, *Controller* (*C*), and *Navigator* (*N*) have been mapped directly to separate states S_1, S_2, and S_4, respectively, as they execute in a

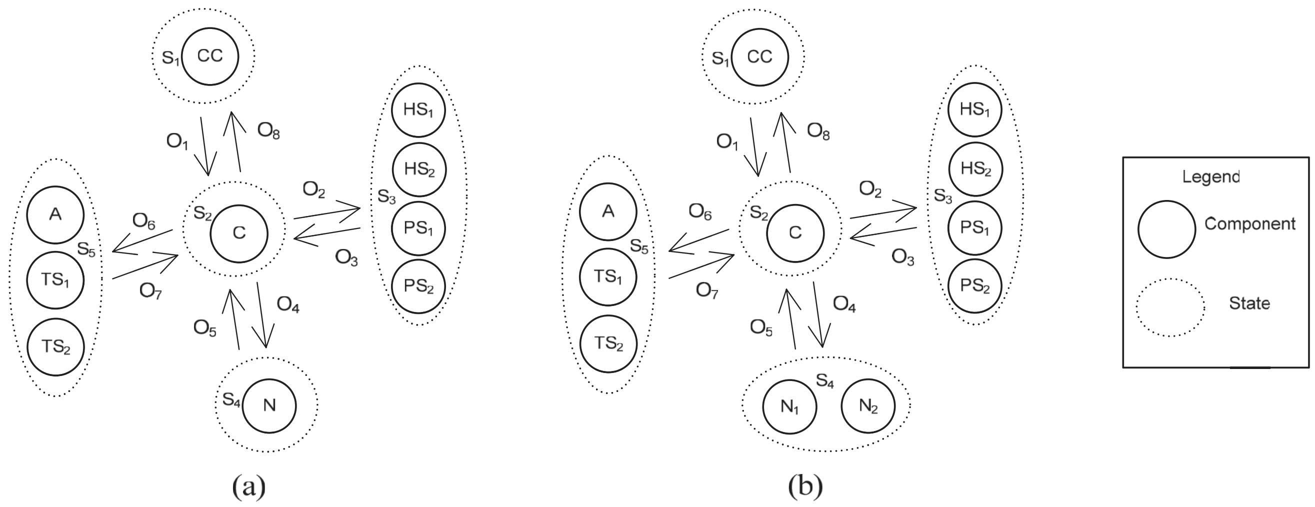

Fig. 4. (a) State model for the robot. (b) State model with the Navigator replicated.

sequential manner. *Heat Sensors* and *Proximity Sensors* (HS_1, HS_2, PS_1, and PS_2) have been mapped to a single state S_3, since they all execute in parallel upon receiving the execution control, and upon completion, the control transfers back to C. Similarly, the *Actuator* (A) and *Touch Sensors* (TS_1 and TS_2) are mapped to a single state S_5. To derive the SOP for the system, a HMM is constructed by using the information in the state model. Thus, from Fig. 4a, we can identify the states (i.e., set S) and observations (i.e., set O) for the HMM as follows:

$$S = \{S_1, \ldots, S_5\} \text{ and } O = \{O_1, \ldots, O_8\},$$

where observations $O_1, \ldots, O_8$ represent the state transitions between states that result from transfer of control between components (i.e., interactions) as shown in Fig. 4a. The runtime data used to train the HMM consist of these observation sequences, which correspond to state transitions.

For illustration, consider the following to be the transition probability matrix derived from HMM using the Baum-Welch algorithm:

$$A_{System} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0.1996 & 0 & 0.2001 & 0.4002 & 0.2001 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

The above transition probability matrix corresponds to the robot's system-level SOP based on its *present* context. To compute system reliability, a transition matrix M is derived for the model in (4) with the matrix elements representing probability of successfully transitioning from state S_i to S_j computed as $R_i \times P_{ij}$. Here, R_i is the reliability of each state computed using the reliabilities of the components mapped to the state, and P_{ij} is the transition probability from state S_i to S_j obtained from the system's SOP.

For example, let us assume that based on the robot's *present* context, the component reliabilities have been computed to be *Controller*: $R_C = 0.9915$ and *Navigator:* $R_N = 0.9751$ using the approach described in the previous section. For the purpose of simplifying this illustration, we assume the remaining components and connectors in the system, i.e., CC, HS_1, HS_2, PS_1, PS_2, TS_1, TS_2, and A are 100 percent reliable. 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 A_{System} and the component-level reliabilities, we obtain the following for transition matrix M:

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1983 & 0.3966 & 0.1983 & 0.1983 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0.9751 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Solving the model according to (4) yields the system's reliability in its *present* context as 0.9152.

6.4 Context-Aware Configuration Reliability Prediction

Similar to the component-level reliability prediction, predictions of the system reliability must incorporate contextual changes into the analysis. By considering the changes that can occur in a system's SOP as a result of the anticipated contextual changes, transition probabilities in matrix A_{System} are updated. This entails utilizing functions of the form (1) to obtain the SOP for the system's *future* context.

For the purpose of predicting the system's SOP, we follow an approach similar to the prediction of component's SOP described in Section 6.2. The only exception here is that the system-level reliability model does not directly incorporate the notion of failure states. The system's *future* SOP is derived based on the transition probabilities of the *present* SOP by applying the following two rules $\forall I_n \in I$:

- The transition probability a_{ij} from state S_i to S_j that is impacted as a result of I_n is revised such that the updated value a'_{ij} is given by

$$a'_{ij} = \mu_n(\mathcal{P}(C)). \qquad (5)$$

For example, in Fig. 4a the transition probability from state S_2 to S_3 is impacted by the navigational complexity of the robot's environment. Here, μ_n is a function that correlates the navigational complexity (i.e., a contextual parameter) to the transition probability from state S_2 to S_3.

- Given the changed a_{ij}, the remaining transition probabilities in row i of the transition probability matrix A are updated to preserve the stochastic property of matrix A. For example in Fig. 4a, as a result of an increase in the transition probability from state S_2 to S_3, the transition probabilities from state S_2 to S_1, S_4 and S_5 need to decrease. Thus, all transition probabilities a_{ik} in row i excluding a_{ij} are adjusted such that

$$a'_{ik} = a_{ik} \times \left(1 - \frac{a'_{ij} - a_{ij}}{\sum_{k \neq j} a_{ik}}\right). \tag{6}$$

As an illustration, μ_{bump} given below quantifies the transition probability a_{23} from state S_2 to state S_3 with respect to the navigational complexity of the robot's physical context given by c:

$$\mu_{bump}(c) = \begin{cases} 0.125c + 0.2003, & 0 \leq c \leq 0.3 \\ 0.09714c + 0.2142, & 0.3 < c \leq 0.65 \\ 0.6094c + 0.13296, & 0.65 < c \leq 1. \end{cases}$$

Continuing on the scenario from Section 6.2, let us assume that the terrain complexity c is expected to increase to 0.45. We update the transition probability a_{23} based on $\mu_{bump}(c)$ above, and adjust all other transition probabilities in that row using (6) to obtain the following SOP for the system:

$$A'_{System} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0.1855 & 0 & 0.2579 & 0.1855 & 0.3711 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

Recall from Section 6.2 that under the *future* context the reliability of the *Controller* is predicted to decrease to 0.9826, and that the reliabilities of the rest of the components remain the same. Using A'_{System} as the predicted system-level SOP, the matrix M' can be recomputed as follows to derive the system-level transition matrix required for (4):

$$M' = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.2534 & 0.1823 & 0.3646 & 0.1823 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0.9751 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Solving the model based on the revised matrix M' using (4) yields the system's reliability in its *future* context as 0.8736.

7 System Reconfiguration

If the predicted system reliability for a given architectural configuration does not meet the acceptable level of reliability, system reconfiguration may be required. In this section, we describe the architectural reconfiguration decisions utilized by RESIST that drive the process of reliability improvement.

7.1 Impact of Component Replica

Architectural patterns formulate preconceived solutions for recurring software problems, including those having to do with the system's quality attributes (e.g., reliability, performance) [35], [47]. Runtime adaptation and reconfiguration of the system aimed at improving a certain quality may entail application of a pattern known to promote that quality. The fault-tolerant pattern [30], for example, improves the system's reliability by replicating critical components. A *multiversioning* connector [30] in the form of a middleware service can be used to handle the component failures by relaying the requests to the hot standby component replicas.

In the case of the robot, the original architecture in Fig. 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 pattern in this case can improve the reliability by replicating the *Navigator* component, which represents a critical point of failure. Recall from Section 5 that we have adopted a probabilistic failure model, commonly used in the literature; hence, an underlying assumption is that replicas fail independently.[1] Fig. 1c shows a replicated *Navigator* component added to the original architecture while running on a new process. The corresponding state model (Fig. 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 [52]. Hence, the probability of state N' executing without failure 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 such that state N is replaced by the new state N', and the matrix element representing the transition from N (which is now N') to C increases to 0.9994 from 0.9751. Solving the model above according to (4) yields a system reliability of 0.9124. Given that in its present configuration, the reliability is predicted to be 0.8736, replication of the *Navigator* results in an improvement of approximately 4.4 percent.

7.2 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

1. This assumption may not hold in some settings, as demonstrated in the following seminal work [20]. In such settings, RESIST's failure model would need to be changed to account for failure dependence, which has been outside the focus of this paper.

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 us assume that R_N and R_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 $R_N \times R_C$, where failure in either the *Navigator* or the *Controller* will cause both components to fail. For instance, assuming that the reliabilities of the *Navigator* and the *Controller* have been predicted to be 0.9826 and 0.9751, respectively, the effective reliability of the two components would be $R'_N = R'_C = 0.9581$. Intuitively, the drop in the two components' effective reliability results in a decrease in the overall system reliability. Therefore, the deployment architecture in which components are deployed as separate processes yields higher reliability.

8 Configuration Selection

The reliability estimation approach presented earlier can be used to determine the most reliable configuration for a mobile and embedded software system. However, in practice, 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. In this section, we first formally specify the problem of finding an appropriate configuration for the systems managed using RESIST. Afterwards, we describe *RESISTER*, a heuristic-driven algorithm for solving such problems.

8.1 Problem Formulation

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.[2] Specifically, RESIST's objective is to find an architectural configuration C^* such that

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

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

where U_q is a utility function indicating the engineer's preferences for the quality attribute q, R corresponds to (4) that calculates the expected reliability of a given architecture C as further detailed below.

A utility function is used to perform tradeoff analysis between competing (conflicting) quality concerns. In the robotic software system, we used three utility functions: One specifies the user's preference for improvements in *reliability*, while two others specify the same for *efficiency* dimensions, namely *CPU* and *memory utilization*. Alternatively, we could have used a single utility function for efficiency, but as we will see in Section 10.1, since in the case of the robotic system efficiency was quantified in terms of CPU and memory utilization, expressing the utility in terms of those dimensions was more natural.

Elicitation of user's preferences is a topic that has been investigated extensively in the literature (e.g., [48]). 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 described in [48].

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.

We define T as a set of components and H as a set of processes. Thus, for a system with $|\mathrm{T}| = \mathrm{t}$ number of software components and $|\mathrm{H}| = \mathrm{h}$ processes, an architectural configuration for the aforementioned decision making problem can be formally specified as follows:

Let decision variable $p_i \in \mathbb{Z}^+$ represent the number of replicas for component i.

Let decision variable $x_{ij} \in [0, 1]$ to indicate if component i is placed on the process j.

The architectural configuration C is a tuple defined in terms of two sets of decision, the deployment of components on processes (x_{ij}) and number of replicas for each component (p_i):

$$C = \langle \{x_{ij} : \forall i \in T, \forall j \in H\} \mid \{p_i : \forall i \in T\} \rangle.$$

The configuration is subject to the following constraints:

- Each component *must* be placed on a process:

$$\forall i \in \{1, \dots, t\}, \ \sum_{j=1}^{h} x_{ij} = 1.$$

- An architectural constraint *may* be applied to limit the number of replicas allowed for a component:

$$\forall i \in \{1, \dots, t\}, p_i \leq w_i, \text{where } w \in \mathbb{Z}^+.$$

- 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:

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

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

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

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 another 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 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 (4).

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 pattern 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. The solution space may be pruned by imposing architectural constraints, such as the limit on the number of replications allowed.

An optimal solution to this problem can be found using an *exact* algorithm. An exact algorithm would be either an off-the-shelf solver (e.g., CPLEX [9]) that would first transform the problem from a nonlinear form into an integer linear programming form by adding auxiliary variables [27], and thereby exploding the complexity of the problem even further, or by exhaustively checking the entire solution space. The NP-hard nature of our architecture optimization problem means that optimal solutions can only be found in very small problems [2], [27]. This motivated the development of heuristic-based approximation algorithms as discussed in the next section.

8.2 RESISTER Algorithm

We have developed an algorithm that utilizes heuristics specific to our problem domain to find near-optimal solutions. Given the NP-hard nature of our problem, developing an algorithm capable of finding the optimal solution in polynomial time is infeasible. Moreover, in RESIST, since our objective is to apply runtime adaptation before the system's reliability degrades, optimality is not as much of a concern as the speed with which an acceptable solution is found.

The algorithm primarily consists of two steps: *satisfy* and *optimize*. The purpose of the *satisfy* step is to find a configuration that meets the reliability requirement, which in the case of mobile and embedded software takes precedence over every other criteria. This initial configuration is then used as a "seed" for the second step, *optimize,* which is aimed at evaluating several configurations derived from the seed using a neighborhood search strategy to find a configuration that yields the highest utility. In the following sections, we describe each step in greater detail.

```
satisfy
01 while $|VT| < |T|$
02     let $i \in T \mid (r_{i_{eff}} \le r_{k_{eff}} \ \forall k \in T \wedge i \notin VT)$
03     if $(q_i == 1)$
04         Let $j \in H \mid \forall i \in T,\ \sum_{i=1}^{t} x_{ij} = 0$
05         $x_{ij} = 1$
06         if $(R(C) \ge \delta)$
07             return $C$
08     while $(p_i < w_i)$
09         $p_i{+}{+}$
10         if $(R(C) \ge \delta)$
11             return $C$
12     $VT = VT \cup i$
13 return $C$
```

Fig. 5. Pseudocode for the *initialize* operation of the algorithm.

8.2.1 Satisfy Step

In this step, we leverage a greedy approach to find an initial configuration that meets the reliability constraint. The approach is based on a *largest-variance-first* tactic [37], where we first apply changes to the configuration that yield the maximum reliability gain. Given an initial configuration, the algorithm begins by selecting the most unreliable component in the system. It applies architectural reconfiguration operations on the component to improve the reliability of the system. It then visits the next most unreliable component and repeats the process until either all components have been visited, or an architecture that meets the reliability requirement has been found.

The two reconfiguration operations described in Section 7 are applied here: 1) changing the deployment architecture by redeploying the unreliable component to a vacant process, and 2) changing the architectural configuration by replicating the component and placing the replica on a new process. In this manner, the effect of each component's unreliability is alleviated, resulting in an increase in system reliability. The largest-variance-first tactic helps us reach the reliability goal with the minimum number of iterations.

Fig. 5 shows the pseudocode for the *satisfy* step. We define VT as an empty set, and iteratively each of the components in the architecture are added to it. In line 2, the algorithm selects an unvisited component that is least reliable. If it shares a process with another component (line 3), it finds a vacant process and deploys the component on it (lines 4 and 5). Then, it evaluates to see if the new configuration satisfies the reliability constraint, and if so, the *satisfy* step ends with C as the output (lines 6 and 7). Otherwise, the algorithm increases the number of replicas for the component under the constraint that it would not exceed the total number of replicas allowed for that component (lines 8 and 9). The resulting architecture is evaluated again to see if it meets the reliability requirement, and if so, the *satisfy* step ends with configuration C as the output (lines 10 and 11). If it does not meet the reliability constraint, the visited component is added to the set VT, and the process is repeated for the next most unreliable component.

If all components have been visited and no configuration is found that meets the reliability constraint, the *satisfy* step ends with the best available configuration C as the output.

```
optimize
01 SatisfiedC = C
02 $BestC = C$
03 for each iteration k
04     i = RAND(T)
05     j = RAND(H)
06     if (($\forall m \in T,\ \sum_{m=1}^{t} x_{mj} = 0$) $\vee$
       ($p_i = 0 \wedge \forall m \in T,\ x_{mj} = 1 \longrightarrow p_m = 0$ ))
07         $x_{ij} = 1$
08         if ($R(C) \geq \delta \wedge U(C) \geq U(BestC)$)
09             $BestC = C$
10     if ($p_i < w_i \wedge q_i == 0$)
11         $p_i{+}{+}$
12         if ($R(C) \geq \delta \wedge U(C) \geq U(BestC)$)
13             $BestC = C$
14     if ($p_i > 0$)
15         $p_i{-}{-}$
16         if ($R(C) \geq \delta \wedge U(C) \geq U(BestC)$)
17             $BestC = C$
18     C = SatisfiedC
19 return $BestC$
```

Fig. 6. The pseudocode for the *optimize* operation of the algorithm.

This corresponds to a situation where the reliability requirements cannot be met through reconfiguration of the software. In such cases, other measures would need to be employed to improve the system's reliability.

8.2.2 Optimize Step

Upon termination of the *satisfy* step, and assuming a solution has been found that meets the reliability requirement, the algorithm continues to the *optimize* step. This step is necessary because the solution that meets the reliability requirement may not be a good solution with respect to the overall utility. Recall from Section 8.1 that RESIST utilizes a multidimensional utility function to denote the user's preferences for other relevant properties (i.e., efficiency). The algorithm subsequently performs the *optimize* step, where it explores the neighborhood of the configuration found by the *satisfy* step in the hope of finding a solution with comparable reliability, but higher utility.

Fig. 6 shows the pseudocode of this step. *SatisfiedC* defined in line 1 is used to maintain the solution resulting from *satisfy* step, while *BestC* defined in line 2 is used to keep the best valid solution throughout the search. The following process then repeats for k iterations, where the value of k is selected based on either the time limit or system's performance requirement. First, a component and a process are picked randomly (lines 4 and 5). The algorithm then uses the randomly picked elements to explore the neighboring solutions by making three types of modifications:

- *Change the deployment of components (lines 6-9).* At line 6, the possibility of deploying component i on process j is evaluated. For this, either j should be a vacant process or i and all other components deployed on j should not have replicas. This heuristic allows the algorithm to avoid changes that are likely to break the reliability requirement. The reasoning behind it is that if a component has been replicated as a part of the *satisfy* step, that component is likely to be one of the least reliable components, and thus is not suitable for modification at this stage.
- *Increase the number of replicas (lines 10-13).* In line 10, if component i does not share the process with any other components and has not exceeded the maximum number of replicas, it is replicated (line 11). This change improves the reliability at the expense of efficiency, and could improve the utility if the user has placed more emphasis on reliability improvements over the system's efficiency, as captured in the corresponding utility functions.
- *Decrease the number of replicas (lines 14-17).* The algorithm attempts to find a configuration with better utility by decreasing the number of replicas. Contrary to the above case, this change improves the efficiency at the expense of reliability, and could improve the utility if the user has placed more emphasis on efficiency improvements over the system's reliability.

As part of each aforementioned change, the algorithm checks to see if 1) the changed configuration is valid (i.e., its reliability satisfies the required threshold) and 2) the overall utility is better than the best utility found so far. If so, the configuration is assigned to *BestC*. Upon termination, the algorithm returns *BestC*, which embodies a configuration for the system that not only satisfies the reliability requirement, but also achieves a good utility.

9 Implementation

We have developed a prototype implementation of RESIST that integrates 1) an extended version of XTEAM [10] as the environment for maintaining the structural, behavioral, and reliability models, 2) Prism-MW [24] 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 metamodels 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 associate a transition probability with each FSP action. We also extended the traditional xADL model support in XTEAM to model reliability properties of the architectural constructs, such as component reliability. Fig. 7 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

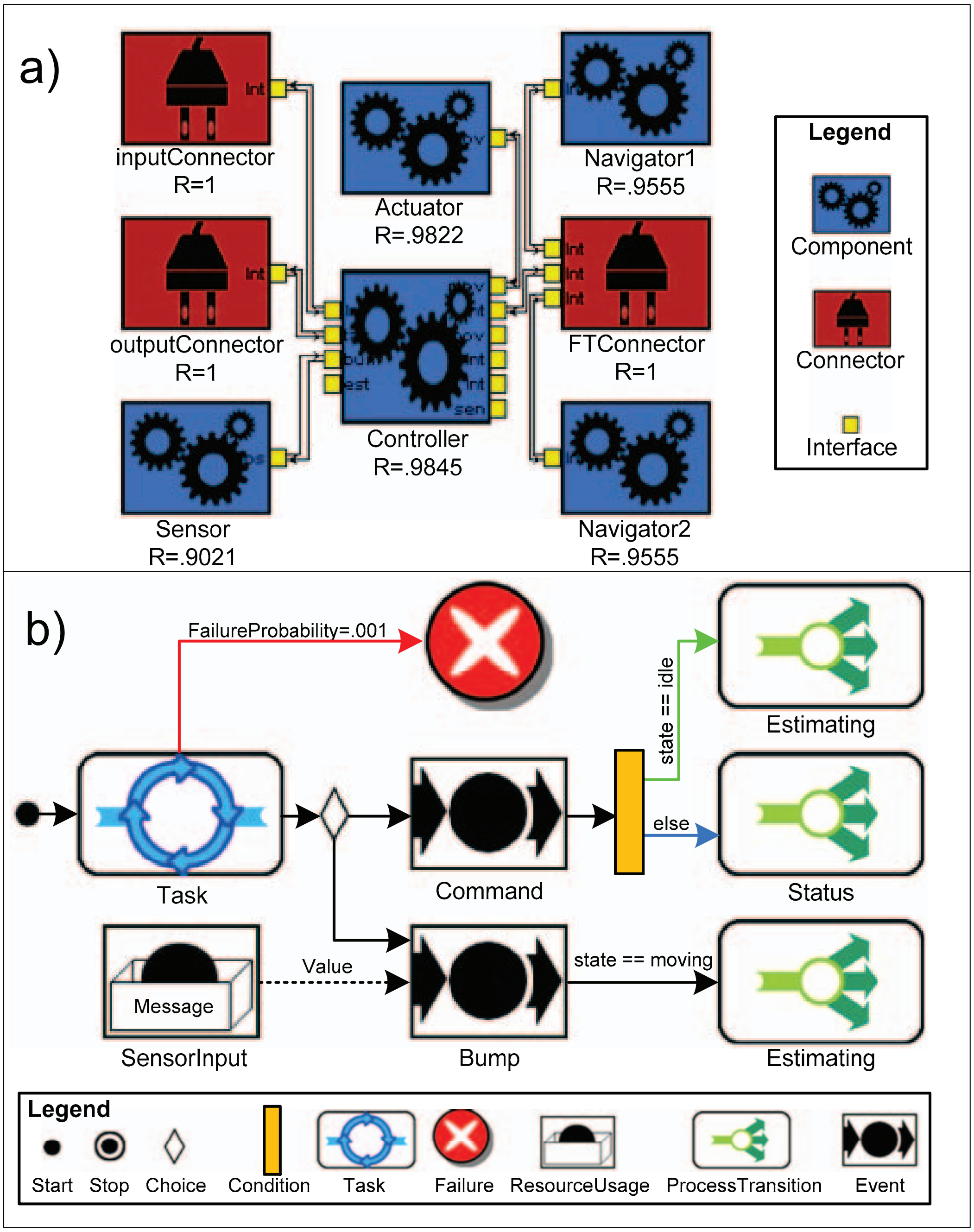

Fig. 7. Reliability-annotated architectural models of a portion of the robot's Controller component in XTEAM: (a) structural view in xADL, and (b) behavioral view in FSP.

used to find optimal or near-optimal configuration for the system using an implementation of exact algorithm or RESISTER algorithm, respectively.

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 [24].

The interested reader may download the artifacts comprising the RESIST prototype from [40].

10 Evaluation

We have evaluated RESIST using its prototype implementation and the mobile emergency response system described earlier. The evaluation consists of six criteria:

1. the impact of architectural reconfiguration decisions on the reliability of components and the system,

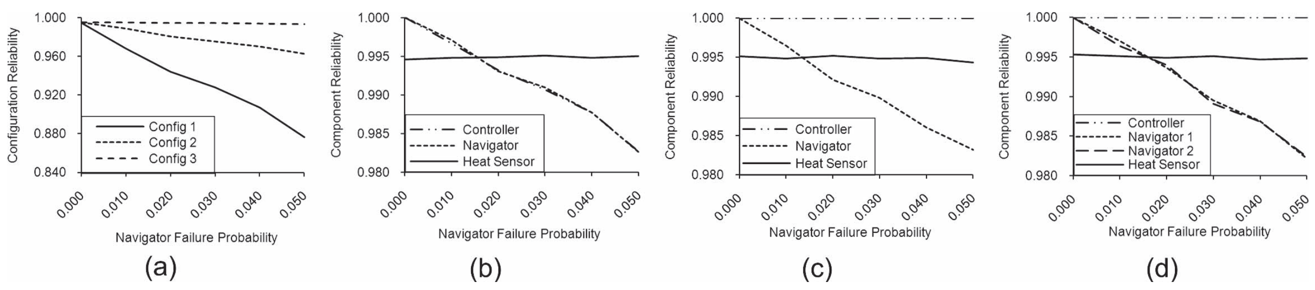

Fig. 8. (a) System reliability for three architectures; Parts (b), (c), and (d) show component reliabilities for configurations 1, 2, and 3, respectively.

2. the validity of reliability prediction based on expected changes in the context,
3. the ability to maintain the system's reliability through proactive reconfiguration,
4. the overhead of runtime reliability analysis, and finally,
5. the effectiveness, and
6. performance of RESISTER algorithm.

We first provide an overview of our experiment setup, followed by a detailed presentation of our evaluation results.

10.1 Experiment Setup

In all of the experiments discussed herein, RESIST was executed 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., [32]).

We estimated the efficiency of software in terms of its memory and CPU utilization. We used analytical models where the total memory and CPU utilization are computed as a function of the number of components, processes, and the average memory and CPU cycles required by the configuration. Given a configuration C, the following analytical models were used for computing memory utilization $M(C)$ and processing utilization $P(C)$:

$$M(C) = \frac{h \times mem_0 + \sum_{i=1}^{t} mem_i}{mem_{avail}} \times 100,$$

$$P(C) = \frac{h \times proc_0 + \sum_{i=1}^{t} proc_i}{proc_{avail}} \times 100,$$

where

$$\begin{aligned} mem_0 &= \text{average memory required by a process}, \\ mem_i &= \text{average memory required by component } i, \\ mem_{avail} &= \text{total memory available}, \\ proc_0 &= \text{average CPU cycles required by a process}, \\ proc_i &= \text{average CPU cycles required by component } i, \\ proc_{avail} &= \text{total CPU cycles available}. \end{aligned}$$

Recall from Section 8.1 that t and h denote the number of components and processes, respectively.

Sigmoid curves have been shown to be practical for specifying the user's quality preferences in terms of utility [48], in particular in the constructions of autonomic systems (e.g., [19], [31]). We used the following sigmoid curve functions to express the utility functions for the three quality attributes of concern:

$$\begin{aligned} U_{Rel}(R(C)) &= \frac{1}{1 + e^{-0.1(100R(C)-50)}}, \\ U_{MemUtiliz}(M(C)) &= \frac{1}{1 + e^{0.1(M(C)-50)}}, \\ U_{ProcUtiliz}(P(C)) &= \frac{1}{1 + e^{0.1(P(C)-50)}}. \end{aligned}$$

The global utility function $U_g(C)$ was then defined as follows:

$$U_g(C) = U_{Rel}(R(C)) + U_{MemUtiliz}(M(C)) + U_{ProcUtiliz}(P(C)).$$

The global utility range for all of the results presented here is $[0, 3]$, since each of the sigmoid curve functions representing the individual utility functions can return a value in the $[0, 1]$ range. However, this does not necessarily mean that for a given problem, it is possible to find a solution that achieves the maximum utility of 3. In fact, due to the NP-hard nature of our problem, determining the maximum utility possible in a given problem is not possible without exploring the entire solution space first.

In our experiments, we used XTEAM to control the system's operational profile (i.e., usage) and Prism-MW for gathering runtime data. However, neither the robotic software nor RESIST was controlled, which allowed them to behave as they would in practice.

10.2 Impact of Reconfiguration

We first evaluate our assertion regarding the impact of architectural reconfiguration on the system's reliability by comparing the components' and subsequently the system's reliability under different configurations. In this set of experiments, we have manually injected defects in the *Navigator* with varied probability of failure. The failure probability for the *Controller* and one of the *Heat Sensors* components is fixed at 0 and 0.15, respectively. We have controlled the experiment by fixing both the usage profile and context.

Fig. 8 shows the reliability estimates obtained for three different architectures as the *Navigator*'s failure probability increases. Part (a) shows the system reliability for the following three configurations: 1) *Navigator* and *Controller* are placed in the same process, 2) they are placed in separate processes, and 3) *Controller* remains in a separate process, the *Navigator* is replicated, and each replica placed in a separate process. In all configurations, the rest of the components in the system are placed in separate processes, and their failure probability is fixed at 0. Parts (b), (c), and (d) show the components' reliability for configurations 1, 2, and 3, respectively.

As shown in part (a), the different architectural configurations exhibit starkly different reliabilities, corroborating the impact of architectural decisions on system's reliability. Configuration 1 results in the lowest system reliability as the *Navigator*'s failure probability increases because the two components are placed on the same process. As shown in part (b), along with the increase in *Navigator*'s failure probability, the reliabilities of the *Navigator* and the *Controller* remain equal as they fail together, despite the fact that the *Controller*'s failure probability is 0. As expected, in Configuration 2, isolating the components to separate processes resulted in an overall improvement in system reliability. This is due to the fact that given the allocation of *Controller* and *Navigator* on separate processes, the effective *Controller*'s reliability is now increased to 1, shown in part (c). In Configuration 3, the *Navigator* component is replicated. This configuration is the most reliable of the three. As shown in part (d), in contrast with reallocation to separate processes, replication does not impact the components' reliability, but results in a system wide improvement. Finally, the *Heat Sensor* is unaffected throughout the experiments, as it is placed in a separate process.

10.3 Validity of Reliability Prediction

As described in Section 6, 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 with the *complexity* of the terrain through which the robot navigates 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 Section 3), thereby altering its operational profile. The techniques presented in [45] together with multilinear regression were used in our experiments to derive function μ (recall Section 6) that estimates the impact of change in terrain to change in bump probability with ± 2.1 percent error at 95 percent 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.

Fig. 9 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

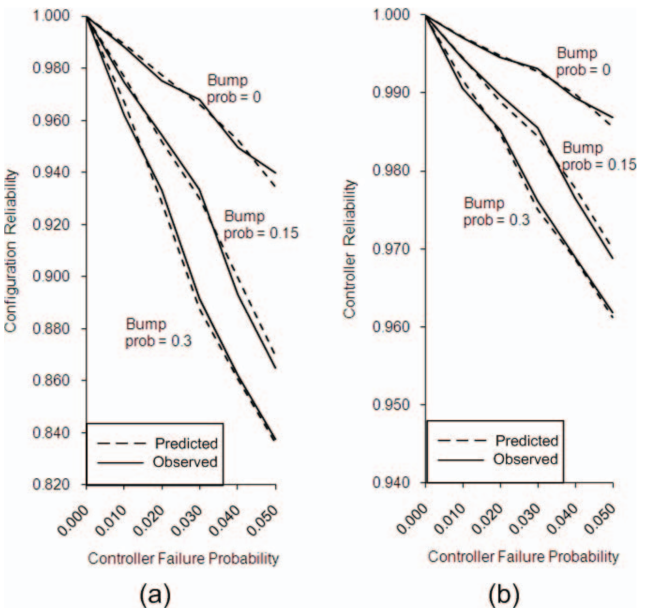

Fig. 9. Accuracy of reliability predictions: (a) system reliability, (b) Controller's reliability.

transitions to the *estimating* state leads to more failures. Further, the deviations 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 percent likely error bound of 2.1 percent, a small deviation in results is to be expected. However, the deviation is small enough that very accurate adaptation decisions could be made.

10.4 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. Results show that the system using RESIST was able to successfully satisfy the reliability requirement throughout its operation. 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 percent throughout its execution, which formed the constraint in our optimization problem. Initially, *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.

Fig. 10a 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 percent. As a result, the

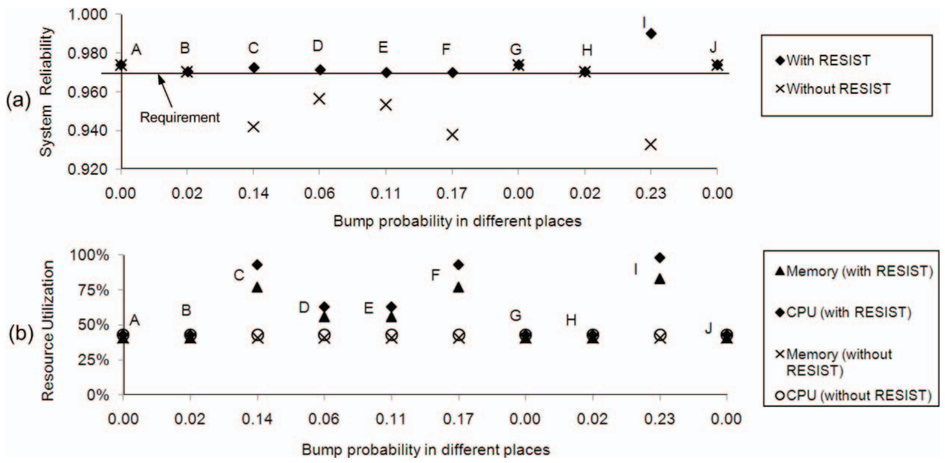

Fig. 10. Context-aware proactive reconfiguration. (a) System reliability. (b) Resource utilization efficiency.

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 percent requirement.

Fig. 10b shows the effect of reconfiguration on the system's resource utilization. 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 percent reliability requirement.

10.5 Overhead of Reliability Analysis

Since RESIST is intended to manage embedded and mobile software at runtime, it is important to assess the performance overhead of RESIST's analysis. Tables 1 and 2 show the benchmarking results of RESIST's reliability analysis. The results in Table 1 show the time it took to perform the reliability analysis for varying number of commands (i.e., tasks sent to the robot). While in each experiment the system is comprised of 10 components, 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 robot software system shows the analysis is often performed on a much smaller number of observations, requiring only a fraction of a second for completion.

TABLE 1
Execution Time of Reliability Analysis with Increasing Number of Observations

Number of Commands	10	50	100	250	500	1000	2000
Number of Observations	174	1062	1741	5874	9553	20028	41879
Execution Time (Seconds)	0.13	0.35	0.69	1.73	2.48	5.10	10.45

The results in Table 2 show the time it took to perform the reliability analysis for varying number of components. In each of the experiments, 100 commands were sent to the robotic system, while the number of components was increased in steps of 10, and the time taken to train the HMM was measured. In the largest scenario where the system comprised of 100 components, the elapsed time was 5.43 seconds.

10.6 Effectiveness of RESISTER

Here, we evaluate the effectiveness of RESISTER algorithm in finding high-quality solutions. This is achieved in terms of two criteria: 1) Is RESISTER capable of finding a solution that satisfies the reliability requirement in a highly constrained solution space (i.e., when only a very small number of configurations out of many satisfy the requirement)? 2) Is RESISTER capable of finding a solution that significantly improves the overall utility?

Evaluating the quality by comparing against the optimal solution is possible only for small problems, since due to the NP-hard nature of our problem the optimal solution for any sizable system is not known. Therefore, for large problems we compare the quality of the solutions found by RESISTER against an unbiased sample selected randomly from the solution space. The sampling of the solution space allows us to show the difficulty of a given problem and, thus, evaluate REISTER's ability to find a solution in cases where the majority of configurations do not satisfy the reliability requirement. Table 3 shows the result of our experiments in nine problems. Each problem dealt with finding a configuration that satisfied the 99 percent reliability requirement in a system with 15 components and processes, where up to five replicas per component were allowed. In generating these problems, we randomly assigned each component a reliability value within a range, which is depicted on the left most column of Table 3. As we traverse from the top to the bottom of the table this range is increased, resulting in problems that are easier to solve. This is because the more reliable the components are, the easier it is to find a configuration that satisfies the reliability requirement.

TABLE 2
Execution Time of Reliability Analysis with Increasing Number of Components

Number of Components	10	20	30	40	50	60	70	80	90	100
Execution Time (Seconds)	0.768	1.239	1.757	2.291	2.824	3.311	3.844	4.378	4.912	5.430

The table shows the result of applying the RESISTER algorithm on each problem, including both the reliability and the utility of the configuration that is found.

The table also shows the result of selecting 100,000 unbiased samples from the solution space. Each sample corresponds to a functionally correct configuration, albeit one that may not satisfy the reliability requirements. Given that we are sampling the solution space randomly, the easier a problem, the more likely it is that some of the sampled solutions satisfy the requirement. For each experiment, the table shows the highest reliability achieved among the 100,000 samples, ratio of valid solutions (i.e., those that satisfied the reliability requirement of 99 percent), and average utility of valid solutions.

We see that RESISTER is capable of finding a solution that satisfies the minimum reliability requirement of 99 percent in all cases. To fully appreciate RESISTER's ability to solve these problems, consider that none of the 100,000 sampled solutions satisfied the reliability requirement in any of the experiments, except in the easiest case (i.e., the last row). In the last row, where the components are highly reliable, 0.04 percent of 100,000 configurations satisfied the reliability requirement. As shown in the rightmost column, the average global utility of those valid solutions was 1.59, which is less than the global utility of 1.90 found by RESISTER in the same problem.

The comparison with the unbiased sample demonstrates RESISTER's ability to find a configuration that satisfies the reliability requirement and optimizes the utility, even when the solution space is highly constrained.

The optimal solution for a given problem can be found using an exact algorithm that would check the entire solution space (recall Section 8.1). However, the NP-hard nature of this problem makes it impossible to find the optimal solution, except in very small problems. Table 4 shows a comparison of the solutions found by RESISTER with the optimal solutions, for systems with seven components, but varying levels of component reliability. As expected the solutions found by RESISTER have lower utility than the corresponding optimal solutions, but the difference in execution time is considerable. On average, the exhaustive search requires 3 hours to find the optimal solution for this relatively small system of seven components, whereas RESISTER finds a solution in a fraction of second. In summary, the exact algorithm finds better solutions, but it is clearly infeasible to execute for most systems.

10.7 Performance of RESISTER

The time it takes to find a solution together with the time it takes for reliability analysis (evaluated in Section 10.5) constitute RESIST's total delay in reconfiguring the software in response to an expected change in the context. As a result, just like the reliability analysis, the timeliness of RESISTER is an important criterion for being able to reconfigure the software prior to its reliability degradation. Table 5 shows the result of benchmarking the RESISTER algorithm on problems ranging from 10 to 640 components in two scenarios with varied degrees of component reliability. The reliability constraint for all of these problems was set at 99 percent. In the first scenario, the reliability of components was randomly selected in the range of (0.955, 0.96), whereas in the second scenario it was selected in the range of (0.99, 1). We should note that the former represents a rather extreme scenario, as a system in which all of the

TABLE 3
Quality of Solutions Found by the RESISTER Algorithm Compared to Statistical Average

Component reliability range	Solution by RESISTER		100,000 unbiased samples from the solution space		
	Reliability $R(C)$	Utility $U_g(C)$	Highest reliability	Ratio of valid solutions	Average utility
(0.955, 0.96)	99.06%	1.09	84.93%	-	-
(0.96, 0.965)	99.01%	1.04	80.81%	-	-
(0.965, 0.97)	99.01%	1.48	86.98%	-	-
(0.97, 0.975)	99.04%	1.05	91.75%	-	-
(0.975, 0.98)	99.14%	1.17	92.69%	-	-
(0.98, 0.985)	99.23%	1.01	94.16%	-	-
(0.985, 0.99)	99.07%	1.29	96.97%	-	-
(0.99, 0.995)	99.03%	1.16	97.07%	-	-
(0.995, 1)	99.01%	1.90	99.99%	0.04%	1.59

TABLE 4
Quality of Solutions Found by RESISTER Compared to Optimal Solutions Found by Exact Algorithm

Component reliability range	Solution by RESISTER			Solution by exhaustive search		
	Reliability $R(C)$	Utility $U_g(C)$	Execution time (Sec)	Reliabi lity $R(C)$	Utility $U_g(C)$	Execution time (Sec)
(0.955, 0.96)	99.10%	1.24	0.01	99.04%	1.34	10049.99
(0.96, 0.965)	99.62%	1.05	0.02	99.13%	1.49	9788.24
(0.965, 0.97)	99.64%	1.05	0.02	99.36%	1.54	26924.78
(0.97, 0.975)	99.70%	1.01	0.02	99.05%	1.43	15507.49
(0.975, 0.98)	99.26%	1.17	0.03	99.07%	1.19	13582.05
(0.98, 0.985)	99.17%	1.25	0.02	99.15%	1.69	3112.08
(0.985, 0.99)	99.18%	1.70	0.02	99.12%	2.21	7488.40
(0.99, 0.995)	99.53%	1.21	0.02	99.02%	2.40	13357.35
(0.995, 1)	99.53%	2.62	0.01	99.15%	2.69	9243.78

components are that unreliable should go through additional verification and validation to improve their reliability before runtime management techniques, such as RESIST, become practical.

As one would expect, by increasing the size of the problem, the time taken to find a solution increases. The results show that RESISTER is able to find solutions to relatively large problems of up to 80 components in a fraction of a second. We found this result to be satisfactory in our experiments, and we believe it would be in most embedded and mobile software systems, which are not likely to have more components than that.

When the component reliability is chosen in the range of (0.955, 0.96), due to the unreliable nature of the components, the number of replicas needed to satisfy the reliability requirement of 99 percent increases. This in turn expands the solution space that RESISTER needs to search, which explains the increase in the execution time.

It is important to note that when we double the size of components and processes in Table 5, we are not simply doubling the size of the problem, as our problem has a very steep exponential growth (recall Section 8.1). As expected, when we have a very large system of 640 components and each of the components has a very low reliability in the range of (0.955, 0.96), it takes a long time for RESISTER to find a solution that satisfies the stringent reliability requirement. However, as mentioned before, we believe this is an extreme case, as we believe other techniques to improve the reliability of individual software components should be employed, before runtime management techniques, such as RESIST, become practical.

Finally, we believe it may be possible to improve RESISTER's performance in such extreme cases by employing parallelization techniques, but leave this to our future work.

11 Threats to Validity

An important threat to the validity of our research is the assumed model of reliability and the adopted principles for estimating it. In Section 5, we provided a detailed definition of reliability and failure in the context of our research. Majority of the state-based reliability analysis models assume Markov transfer of control among states, which means that the probability of transition to a state at time $t+1$ depends on the state at time t and is independent from its past history. In the context of architecture-based reliability modeling, this would imply that the transfer of control among components maintain the Markov property.

Cheung [5] argues that this assumption at the macroscopic (architecture) level is valid for most systems, although at instruction level it is debatable. A large body of prior research [14], [16], [17] has adopted this model of estimating reliability at the architecture level. As a result, we believe our technique is grounded in principles that are widely accepted. That said, the crux of RESIST is on how reliability predictions could be used to optimize the architecture of a software system at runtime. We believe other types of reliability analysis models

TABLE 5
Execution Time of the RESISTER Algorithm in Seconds

Number of components	10	20	40	80	160	320	640
Execution time for component reliability range (0.955, 0.96) in Seconds	0.04	0.08	0.17	0.73	6.68	190.16	11164.32
Execution time for component reliability range (0.99, 1.00) in Seconds	0.03	0.06	0.14	0.32	2.15	28.05	447.53

could be substituted to furnish RESIST with predictions that it could use for self-management.

Another issue with state-based approaches to reliability estimation is the state-space explosion problem during the analysis. This, however, is significantly less of a problem in an architecture-centric approach, particularly because of the hierarchical and compositional nature of architectural models. In order for architectural models to be useful, they need to remain at a "reasonable" granularity level. Software architects frequently zoom in and out as they change the abstraction level of their viewpoint. This directly helps address the state explosion problem [28], as the architectural models are abstractions of the system's implementation.

Our approach relies on the availability of functions relating contextual variations with changes in the architectural models. As you may recall from Section 10.3, we have used multilinear regression to derive an example of such a function relating the complexity of the terrain navigated by a robot to the frequency with which it may need to recalculate routes. Although our approach illustrates that at least in some cases it is feasible to derive such functions, this is not necessarily the case for all situations. In fact, if the changes in context are occurring completely independent of the historical trends, deriving such functions could be challenging. In such situations, an approach such as RESIST would need to be used in a reactive fashion, as meaningful prediction about the system's future reliability would not be possible.

There are two general limitations in RESIST's adaptation model. First, currently RESIST supports only two types of adaptation (recall Section 7), while there may be other types of adaptation that could improve the system's reliability and in certain situations they may be more appropriate than those supported so far. This assumption is not due to a conceptual limitation, but rather the lack of implementation support in our tool suite for affecting other types of runtime change in the software. Extending RESIST to support other types of adaptation based on the conceptual framework described in this paper should be possible. Second, RESIST assumes changes in the system (e.g., context) are not so frequent that they would trigger constant adaptation of the system. One approach to address this limitation would be to make the adaptation decisions based on predictions over a period of time in future, such that a tradeoff analysis as to the benefits of adaptation versus its overhead could be made.

12 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 [15], [16], [23], [41], [42], [44], [52]. The underlying assumptions in these approaches make them unsuitable for use in the domain of 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 [14], [16], [17] corroborate these observations.

Our past research has addressed some of the uncertainties associated with design-time reliability analysis by incorporating various sources of information [5], [43]. We also identified the challenges of reliability analysis in the mobile domain [26]. 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 the 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 [51], [11], [38]. While [51] and [38] target traditional and highly predictable software, the KAMI framework [11] 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 [3], [13], [21]. In contrast to them, RESIST is narrowly aimed at improving the reliability of dynamic mobile and embedded 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 [21].

Finally, related is previous research on middleware intended for mobile and embedded software. Aura [49] is an architectural style and supporting middleware for ubiquitous computing applications with a special focus on user mobility, context awareness, and context switching. XMIDDLE [29] is a data-sharing middleware for mobile computing. MobiPADS [3] is a reflective middleware that supports active deployment of augmented services (called mobilets) for mobile computing. Lime [33] 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 embedded and mobile software through proactive self-adaptation.

13 Conclusion

Software systems are increasingly deployed 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 lifetime. 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 near-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 comprised of hundreds of components and hardware hosts. We also intend to increase the types of reconfiguration decisions and dependability tradeoffs that RESIST supports. We plan to investigate the use of other stochastic approaches, namely Dynamic Bayesian Networks and Hierarchical HMM. Dynamic Bayesian Networks extend a traditional Bayesian Network by incorporating the notion of time, thus providing a more accurate representation of the failure probability. Hierarchical HMMs directly build upon the hierarchical nature of architectural models, thus allowing for a more natural and intuitive modeling foundation.

Finally, another interesting avenue of future research would be an integration with KAMI [11]. KAMI *incrementally* furnishes reliability estimates through fine-grained refinement of DTMC parameters, while RESIST *periodically* estimates the reliability of the entire system based on the changed DTMC parameters. We believe an integration of the two techniques would be interesting, as it allows us to improve the speed with which RESIST reacts to reliability violations. That is, instead of waiting for the periodic assessment of the system reliability to detect the violations and react to them, we would be able to detect the violations at an earlier time and, thus, react faster than what is currently possible with RESIST.

Acknowledgments

The authors wish to thank David Kilgore for his work on RESIST's tool support. This work was partially supported by grants CCF-1252644, CCF-1217503, and CCF-0937472 from the US National Science Foundation, grant D11AP00282 from the US Defense Advanced Research Projects Agency, and award W911NF-09-1-0273 from the US Army Research Office.

References

[1] G. Abowd et al., "Towards a Better Understanding of Context and Context-Awareness," *Proc. Int'l Symp. Handheld and Ubiquitous Computing,* Sept. 1999.

[2] M.C. Bastarrica, A.A. Shvartsman, and S.A. Demurjian, "A Binary Integer Programming Model for Optimal Object Distribution," *Proc. Int'l Conf. Principles of Distributed Systems,* Dec. 1998.

[3] A. Chan et al., "MobiPADS: Reflective Middleware for Context-Aware Mobile Computing," *IEEE Trans. Software Eng.,* vol. 29, no. 12, pp. 1072-7085, Dec. 2003.

[4] B. Cheng et al., *Software Engineering for Self-Adaptive Systems: A Research Roadmap. Software Engineering for Self-Adaptive Systems.* Springer-Verlag, 2009.

[5] R.C. Cheung, "A User-Oriented Software Reliability Model," *IEEE Trans. Software Eng.,* vol. 6, no. 2, pp. 118-125, Mar. 1980.

[6] L. Cheung et al., "Early Prediction of Software Component Reliability," *Proc. Int'l Conf. Software Eng. (ICSE '08),* May 2008.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond.* Pearson Education, 2010.

[8] D. Cooray, S. Malek, R. Roshandel, and D. Kilgore, "RESISTing Reliability Degradation through Proactive Reconfiguration," *Proc. IEEE/ACM 25th Int'l Conf. Automated Software Eng. (ASE '10),* Sept. 2010.

[9] IBM ILOG CPLEX Optimizer. www.ibm.com/software/integration/optimization/cplex-optimizer/, 2013.

[10] G. Edwards et al., "Scenario-Driven Dynamic Analysis of Distributed Architectures," *Proc. Int'l Conf. Fundamental Approaches to Software Eng. (FASE '07),* Mar. 2007.

[11] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model Evolution by Run-Time Parameter Adaptation," *Proc. Int'l Conf. Software Eng. (ICSE '09),* May 2009.

[12] N. Esfahani et al., "A Modeling language for Activity-Oriented Composition of Service-Oriented Software Systems," *Proc. Int'l Conf. Model Driven Eng. Languages and Systems (MODELS '09),* Oct. 2009.

[13] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *Computer,* vol. 37, no. 10, pp. 46-54, Oct. 2004.

[14] S. Gokhale, "Architecture-Based Software Reliability Analysis: Overview and Limitations," *IEEE Trans. Dependable and Secure Computing,* vol. 4, no. 1, pp. 32-40, Jan.-Mar. 2007.

[15] K. Goseva-Popstojanova, A. Hassan, A. Guedem, W. Abdelmoez, D.E.M. Nassar, H. Ammar, and A. Mili, "Architectural Level Risk Analysis Using UML," *IEEE Trans. Software Eng.,* vol. 29, no. 10, pp. 946-960, Oct. 2003.

[16] K. Goseva-Popstojanova and K.S. Trivedi, "Architecture-Based Approaches to Software Reliability Prediction," *Int'l J. Computer and Math. with Applications,* vol. 46, no. 7, pp. 1023-1036, Oct. 2003.

[17] A. Immonen and E. Niemela, "Survey of Reliability and Availability Prediction Methods from the Viewpoint of Software Architecture," *J. Software and Systems Modeling,* vol. 7, no. 1, Feb. 2008.

[18] S. Karlin and H.M. Taylor, *An Introduction to Stochastic Modeling,* third ed. Academic Press, 1998.

[19] J.O. Kephart and R. Das, "Achieving Self-Management via Utility Functions," *IEEE Internet Computing,* vol. 11, no. 1, pp. 40-48, Jan./Feb. 2007.

[20] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Trans. Software Eng.,* vol. 12, no. 1, pp. 96-109, Jan. 1986.

[21] J. Kramer and J. Magee, "Self-Managed Systems: An Architectural Challenge," *Proc. Int'l Conf. Software Eng. (ICSE '07),* May 2007.

[22] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Trans. Software Eng.,* vol. 16, no. 11, pp. 1293-1306, Nov. 1990.

[23] S. Krishnamurthy and A. Mathur, "On the Estimation of Reliability of a Software System Using Reliabilities of Its Components," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '97),* Nov. 1997.

[24] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems," *IEEE Trans. Software Eng.,* vol. 31, no. 3, pp. 256-272, Mar. 2005.

[25] S. Malek, G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic, and G. Sukhatme, "An Architecture-Driven Software Mobility Framework," *J. Systems and Software,* vol. 83, no. 6, pp. 972-989, June 2010.

[26] S. Malek, R. Roshandel, D. Kilgore, and I. Elhag, "Improving the Reliability of Mobile Software Systems through Continuous Analysis and Proactive Reconfiguration," *Proc. Int'l Conf. Software Eng. (ICSE '09),* May 2009.

[27] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An Extensible Framework for Improving a Distributed Software System's Deployment Architecture," *IEEE Trans. Software Eng.,* vol. 38, no. 1, pp. 73-100, Jan./Feb. 2012.

[28] L. Markides, L. Stetson, K. Zielinski, C. Mattmann, and R. Roshandel, "On the Granularity of Markov-Based Reliability Models," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '10),* Nov. 2010.

[29] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich, "XMIDDLE: A Data-Sharing Middleware for Mobile Computing," *Int'l J. Personal and Wireless Comm.*, vol. 21, no. 1, pp. 77-103, Apr. 2002.

[30] N. Medvidovic, M. Mikic-Rakic, and N. Mehta, "Improving Dependability of Component-Based Systems via Multi-Versioning Connectors," *Architecting Dependable Systems,* R. de Lemos, C. Gacek, and A. Romanovsky eds., Springer-Verlag, 2003.

[31] D.A. Menasce, J.M. Ewing, H. Gomaa, S. Malek, and J.P. Sousa, "A Framework for Utility-Based Service Oriented Design in SASSY," *Proc. Joint WOSP/SIPEW Int'l Conf. Performance Eng.,* Jan. 2010.

[32] Mobile Robots Inc. http://www.mobilerobots.com/, 2013.

[33] A.L. Murphy, G.P. Picco, and G.C. Roman, "Lime: A Middleware for Physical and Logical Mobility," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS '01),* May 2001.

[34] P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-Based Runtime Software Evolution," *Proc. Int'l Conf. Software Eng. (ICSE '98),* Apr. 1998.

[35] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *Software Eng. Notes,* vol. 17, no. 4, pp. 40-52, Oct. 1992.

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

[37] M. Pinedo and G. Weiss, "The Largest Variance First Policy in Some Stochastic Scheduling Problems," *Operations Research,* vol. 35, no. 6, pp. 884-891, Nov. 1987.

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

[39] L. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proc. IEEE,* vol. 77, no. 2, pp. 257-286, Feb. 1989.

[40] RESIST homepage, http://www.sdalab.com/projects/resist, 2013.

[41] R.H. Reussner, H.W. Schmidt, and I.H. Poernomo, "Reliability Prediction for Component-Based Software Architectures," *J. Systems and Software,* vol. 66, no. 3, pp. 241-252, 2003.

[42] G. Rodrigues, D. Rosenblum, and S. Uchitel, "Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems," *Proc. Int'l Conf. Fundamental Approaches to Software Eng. (FASE '05),* Apr. 2005.

[43] R. Roshandel, N. Medvidovic, and L. Golubchik, "A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level," *Proc. Int'l Conf. Quality of Software Architectures (QOSA '07),* July 2007.

[44] H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj, "A Bayesian Approach to Reliability Prediction and Assessment of Component Based Systems," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '01),* Nov. 2001.

[45] H. Seraji and A. Howard, "Behavior-Based Robot Navigation on Challenging Terrain: A Fuzzy Logic Approach," *IEEE Trans. Robotics and Automation,* vol. 18, no. 3, pp. 308-321, June 2002.

[46] B. Schilit, N. Adams, and R. Want, "Context-Aware Computing Applications," *Proc. Int'l Workshop Mobile Computing Systems and Applications,* Dec. 1994.

[47] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[48] J.P. Sousa, R.K. Balan, V. Poladian, D. Garlan, and M. Satyanarayanan, "User Guidance of Resource-Adaptive Systems," *Proc. Int'l Conf. Software and Data Technologies (ICSOFT '08),* July 2008.

[49] J. Sousa and D. Garlan, "Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments," *Proc. Int'l Working Conf. Software Architecture (WICSA '02),* Aug. 2002.

[50] W.J. Stewart, *Introduction to the Numerical Solution of Markov Chains.* Princeton Univ. Press, 1994.

[51] W. Wang, T.L. Hemminger, and M. Tang, "Moving Average Modeling Approach for Computing Component-Based Software Reliability Growth Trends," *INFOCOMP J. Computer Science,* vol. 5, no. 3, pp. 9-18, Sept. 2006.

[52] W. Wang, D. Pan, and M. Chen, "Architecture-Based Software Reliability Modeling," *J. Systems and Software,* vol. 79, no. 1, pp. 132-146, Jan. 2006.

Deshan Cooray received the BS degree in computer science and engineering from the Faculty of Engineering at the University of Moratuwa in 2004, and the MS degree in software engineering from the Department of Computer Science at George Mason University in 2010. His research interests include the field of software engineering with special interest in software reliability, software architecture, quality-of-service analysis and self-adaptive software. He is currently a software engineer at VeriSign, Inc., Reston, Virginia, where he has been employed in the Naming Services division since 2010 (views expressed by him in this paper are not necessarily the views of VeriSign).

Ehsan Kouroshfar received the BS degree from the Amirkabir University of Technology in 2006, and the MS degree in computer engineering from the Sharif University of Technology in 2009. Currently, he is working toward the PhD degree in the Computer Science Department at George Mason University. His research interests include software design, software repository mining, and self-adaptive systems.

Sam Malek received the BS degree in information and computer science from the University of California, Irvine, in 2000, and the MS and PhD degrees in computer science from the University of Southern California in 2004 and 2007, respectively. He is an associate professor in the Department of Computer Science at George Mason University (GMU). He is also the director of GMU's Software Design and Analysis Laboratory, a faculty associate of the GMU's C4I Center, and a member of the US Defense Advanced Research Projects Agency's Computer Science Study Panel. His general research interests include the field of software engineering, and to date his focus has spanned the areas of software architecture, autonomic software, and software dependability. He has received numerous awards for his research contributions, including the US National Science Foundation CAREER award (2013) and the GMU Computer Science Department Outstanding Faculty Research Award (2011). He is a member of the ACM, ACM SIGSOFT, and IEEE.

Roshanak Roshandel received the BS degree in computer science from Eastern Michigan University in 1998, and the MS and PhD degrees in computer science from the University of Southern California in 2002 and 2006, respectively. She is an associate professor in the Department of Computer Science and Software Engineering at Seattle University, where she is also the director of the Master of Software Engineering program. Her research interests include the field of software engineering, with specific interests in software architecture, software dependability, reliability, and security, and software product families. She is a member of the ACM, ACM SIGSOFT, and IEEE.