RESISTING RELIABILITY DEGRADATION
THROUGH PROACTIVE RECONFIGURATION

by

Deshan Cooray
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Software Engineering

Committee:

_____	Dr. Sam Malek, Thesis Director
_____	Dr. Paul Ammann, Committee Member
_____	Dr. Joao Pedro Sousa, Committee Member
_____	Dr. Roshanak Roshandel, Committee Member
_____	Dr. Hassan Gomaa, Department Chair
_____	Dr. Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering
Date:_____	Summer Semester 2010 George Mason University, Fairfax, VA

Resisting Reliability Degradation through Proactive Reconfiguration

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

by

Deshan Cooray

Director: Sam Malek, Assistant Professor
Department of Computer Science

Summer Semester 2010
George Mason University
Fairfax, VA

DEDICATION

To my family

ACKNOWLEDGEMENTS

First and foremost I owe my deepest gratitude to my advisor, Dr. Sam Malek, for his support and guidance during the last year and a half. It has been a privilege to have worked with such an inspirational individual and I am grateful for all he has taught me. I would also like to thank the other members of my committee, Dr. Roshanak Roshandel, Dr. Paul Ammann, and Dr. Joao Sousa for their guidance, feedback and devoting time from their busy schedules.

My sincere thanks also go out to my colleagues and friends of the Software Engineering research group. I appreciate the help you extended at all times, and my special thanks go out to David Kilgore for his significant contributions to the development of RESIST.

I have been blessed with family and friends who have been supportive throughout my years at George Mason University. To my dear parents, Myrna and Bertram, and brothers Jude, Sajith and Geekz - I am forever grateful for all the sacrifices you have made for me. Last but not least, I would like to thank my fiancée Ashantha for all her love, support, and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ABBREVIATIONS

RESIST – Resilient Situated Software System
xADL – Extensible Architecture Description Language
FSP – Finite State Processes
UML – Unified Modeling Language
SOP – Software Operational Profile
DTMC – Discrete Time Markov Chain
HMM – Hidden Markov Model
XTEAM - eXtensible Tool-chain for Evaluation of Architectural Models
LTS - Labeled Transition System

ABSTRACT

RESISTING RELIABILITY DEGRADATION THROUGH PROACTIVE RECONFIGURATION

Deshan Cooray, M.S.

George Mason University, 2010

Thesis Director: Dr. Sam Malek

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

SItuated SofTware system (RESIST), and evaluated it using a mobile emergency response system.

1. INTRODUCTION

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

Engineers of a situated software system typically spend significant effort to determine a good configuration for the system to ensure its adherence to functional and non-functional requirements. For instance, they may perform a trade-off analysis between the system's efficiency and reliability when they decide the allocation of software components to operating system (OS) processes. Clearly the overall reliability of such systems depends on problems both internal (e.g., software bugs) and external (e.g., network disconnection, hardware failure) to the software. The key underlying insight in our research is that some internal software problems may manifest themselves only under certain dynamic characteristics external to the software (e.g., physical location), which is traditionally referred to as *context* [1].

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

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

RESIST uses the reliability predictions to (1) proactively determine when the system should be adapted, and (2) find the optimal configuration for the near future operation of the system. Our evaluations show that our reliability predictions are accurate with respect to the *observed* system reliability. We thus consider the predicted reliability as an indicator for decision making. An important contribution of our work is *proactive* adaptation based on our reliability analysis that reconfigures the system at runtime prior

to actual reliability degradation. This trait clearly sets our work apart from the majority of existing self-adaptive frameworks that are *reactive* in their decision making [2][3].

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.

1.1. Research Hypotheses

This research investigates the following hypotheses.

1.1.1. Architecture-based Software Reliability Prediction

Insight: Some internal software problems may manifest themselves only under certain dynamic characteristics external to the software (e.g., a system's physical environment) which is traditionally referred to as *context*. The execution context of many software systems can be determined a priori (e.g., an ecommerce system experiences higher workload certain times of the year).

Insight: Knowledge embodied in a system's architectural models (e.g., behavioral and structural models of the components and the system) could be used to reason about its runtime characteristics, including its reliability.

Hypothesis #1: *Given the future execution context of a software system and its architectural models, it is possible to estimate the future reliability of the system and its components.*

1.1.2. Proactive Architecture-based Adaptation

Insight: A software system's architectural configuration (e.g., architectural style, deployment architecture) has a significant impact on the system's quality attributes.

Insight: Due to variability in the execution context, the optimal configuration for a software system cannot be determined prior to its deployment, and no particular configuration can be optimal for the system's entire operational lifetime.

Hypothesis #2: *Context-driven reliability predictions could be employed to improve the resilience of a software system to failures through proactive architecture-based adaptation.*

The remainder of this thesis is organized as follows. Chapter 2 presents a motivating example, and Chapter 3 describes the impact of context on the architecture of a system. Chapter 4 provides a high-level overview of the RESIST framework, while Chapter 5 presents its failure model. Chapter 6 presents the component-level and configuration-level reliability models. Chapter 7 describes the alternative architectural configurations aimed at improving reliability. Chapter 8 details the configuration selection process. Chapter 9 describes the implementation and tool support for RESIST, while Chapter 10 presents a detailed evaluation of the work. Chapter 11 describes the related work, and finally Chapter 12 present concluding remarks for the thesis and avenues of future research.

2. MOTIVATING EXAMPLE

Emergency response is a domain that entails a high degree of mission criticality. Software systems designed for this domain thus have stringent reliability requirements. As a motivating example, consider a mobile distributed emergency response system intended to aid the emergency personnel in fire crises, a prototype of which was developed in our previous work [4]. This system consists of several entities, including a central *dispatcher* that serves as the "Headquarters" for coordinating the crew activities, smart *fire engines* that are designed to alert the dispatcher of the current location of the vehicle and provide its occupant with information concerning the crisis scene, *firefighters* equipped with PDAs capable of controlling the robots and sensors, and mobile *robots* that execute the high-level commands.

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

An example architectural strategy for improving the system's 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.

Figure 1 (a) and (b) show two alternative allocations of the robot's software

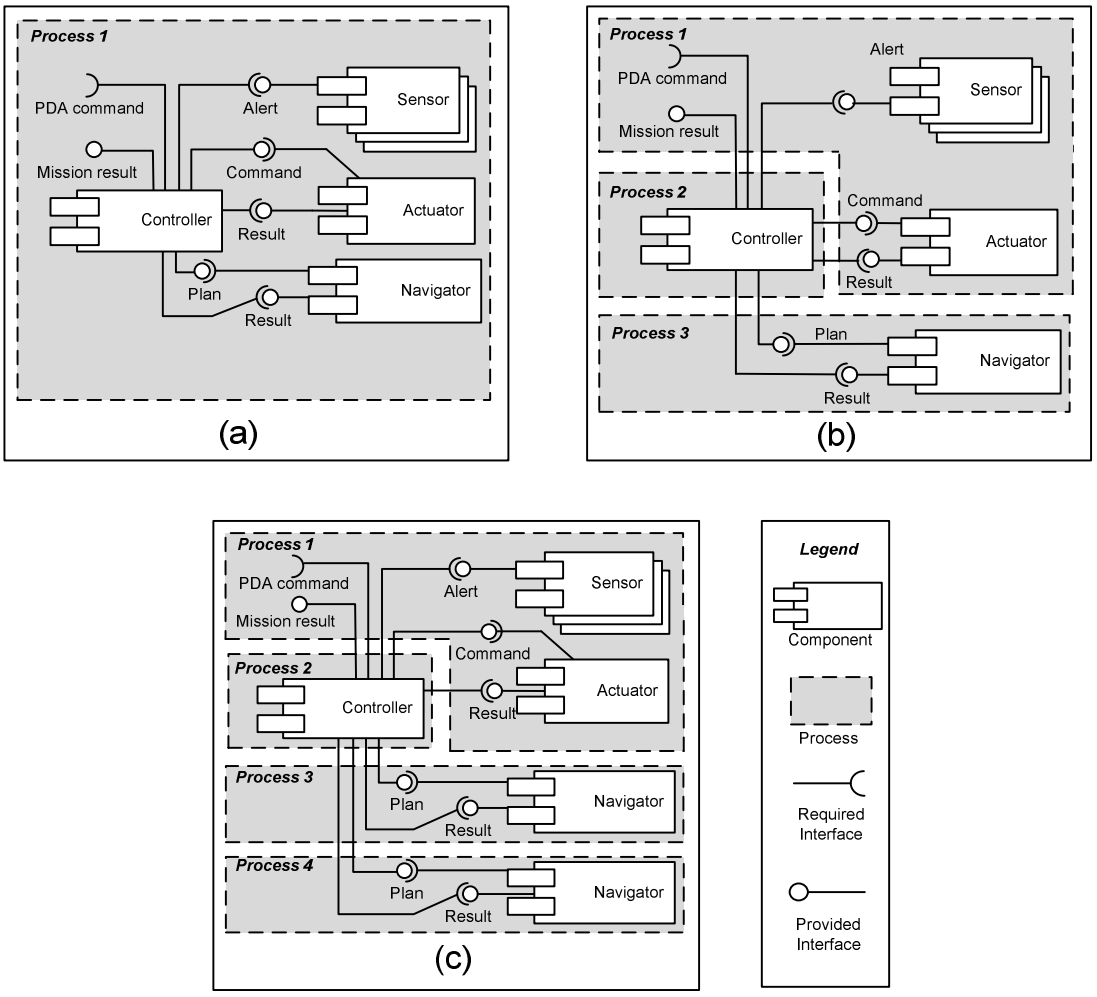

Figure 1. Component-to-process allocation alternatives: (a) All components allocated to the same process, (b) Controller and Navigator allocated to separate processes, and (c) Controller allocated to separate process, and the Navigator is replicated and placed in separate processes.

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

The above scenario demonstrates the impact of architectural decisions on system's quality attributes. Such decisions while critical to system's dependability cannot be made effectively at design-time. It is only reasonable to assume that some of these decisions must be made at runtime, requiring specialized methodologies that continuously evaluate the impact of these decisions on system's dependability. We use this system in the remainder of the thesis 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 its behavior can be considered its context [5]. A system's context may consist of several different aspects of its changing execution environment that could potentially impact the behavior and properties of a system. Among them three main categories of context can be identified [5][6];

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

A context-aware system uses knowledge about its context to provide relevant information and/or services to the user [5]. While in some systems contextual information is directly used to provide services to the user, in some others contextual information is used to optimize the manner in which services are provided to the user. For example, a GPS enabled mobile phone which displays a map based on the user's location considers the location as an input to the service that is provided. In contrast, a mobile robot engaged in firefighting may need to reconfigure itself depending on its contextual characteristics

so that its dependability is optimal with respect to other quality attributes such as resource usage. As described in the next section, RESIST is aimed at the second class of systems. Specifically, RESIST uses the system's context to perform architectural reconfiguration of the system so that it remains resilient in the face of degrading reliability.

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

To exemplify the effect the context has on a system's architecture, below we present how the mobile nature of a robotic system introduces contextual changes that can impact its operational profile, and in-turn its reliability. Figure 2(a) shows the architectural models of the mobile robot. It receives a command from an external system such as a PDA, and returns the result of executing the command. Upon receiving a command, it uses its Sensors to gather data about its environment, such as near-by obstacles and proximity to heat, and determines a plan and executes it using its Navigator and Actuator components, respectively. Figure 2(b) shows the robot's Controller

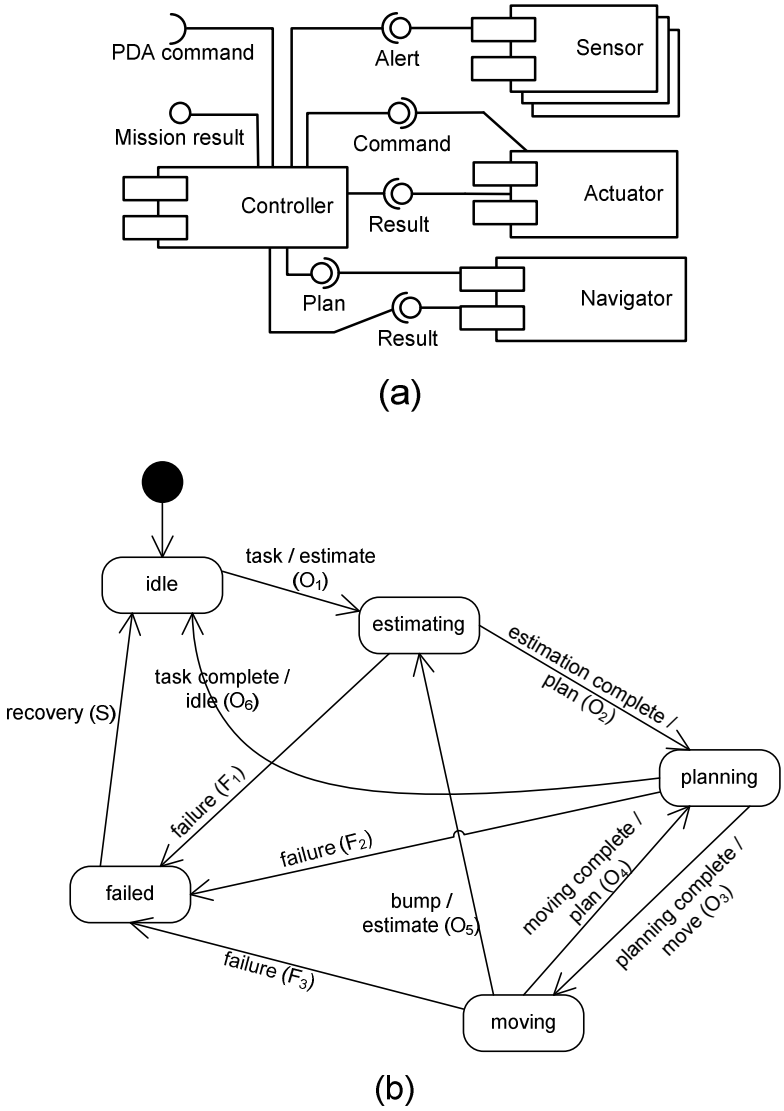

Figure 2. Robot's architecture: (a) robot's structural model, and (b) the behavioral model of the robot's Controller

component's behavioral model in the form of a UML state chart. It includes behavioral states *idle*, *estimating*, *planning* and *moving*, during which the Controller invokes interactions with the other components in the system (i.e., Sensors, Actuator, Navigator, etc.). The *failed* state denotes a common failure state of the component. Transitions O_1 to

O_6 denote behavioral transitions resulting from input events such as interface calls on the component. Transitions F_1 to F_3 denote a failure that may arise under some circumstances. Such failures are caused by faults in the software that could lead to a failure. Transition S denotes eventual recovery of the component as a result of automatic or manual re-initialization of the component.

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

The impact of the system's context is not limited to internal changes in the component behavior, as they may also change the manner in which components interact, and thus influence the system's reliability. For example, the Controller interacts with the Sensors in order to perform estimations prior to planning its navigation route. However,

if the number of bump events increases, the Controller interacts with the Sensors with a higher frequency in order to perform re-estimations. Thus, the impact of the Sensor components' reliability on system's reliability depends on how frequently the Controller needs to interact with the Sensors, which is in turn determined by location dependent contextual information such as the complexity of the terrain (i.e. the probability of bumps).

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

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

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

where $\mathcal{P}(C)$ denotes the power set of contextual parameters.

Considering the robotic system described above as an illustration, the probability of the robot encountering an obstacle on its path is an example of a contextual parameter which changes as a result of its mobility. This contextual parameter has an effect on two architectural parameters: the transition probability from moving to estimating state in the Controller, and the probability that the Controller interacts with the Sensor components. In this illustration we have described two points of interaction between the contextual parameter and architectural parameters. However in any sizable system one could expect multiple points of interaction, which further highlight the importance of properly modeling and incorporating context in engineering mobile systems.

In the next section, we present an overview of the RESIST framework and in the subsequent sections we show how a system's contextual parameters together with their interactions with architectural parameters can be used in predicting a system's reliability and optimizing its architecture.

4. FRAMEWORK OVERVIEW

An overview of RESIST framework is depicted in Figure 3. The process is organized as a feedback control loop that continuously monitors, analyzes, and adapts the system at runtime. RESIST consists of three conceptual software components.

At design-time and before the system's implementation is complete, an initial set

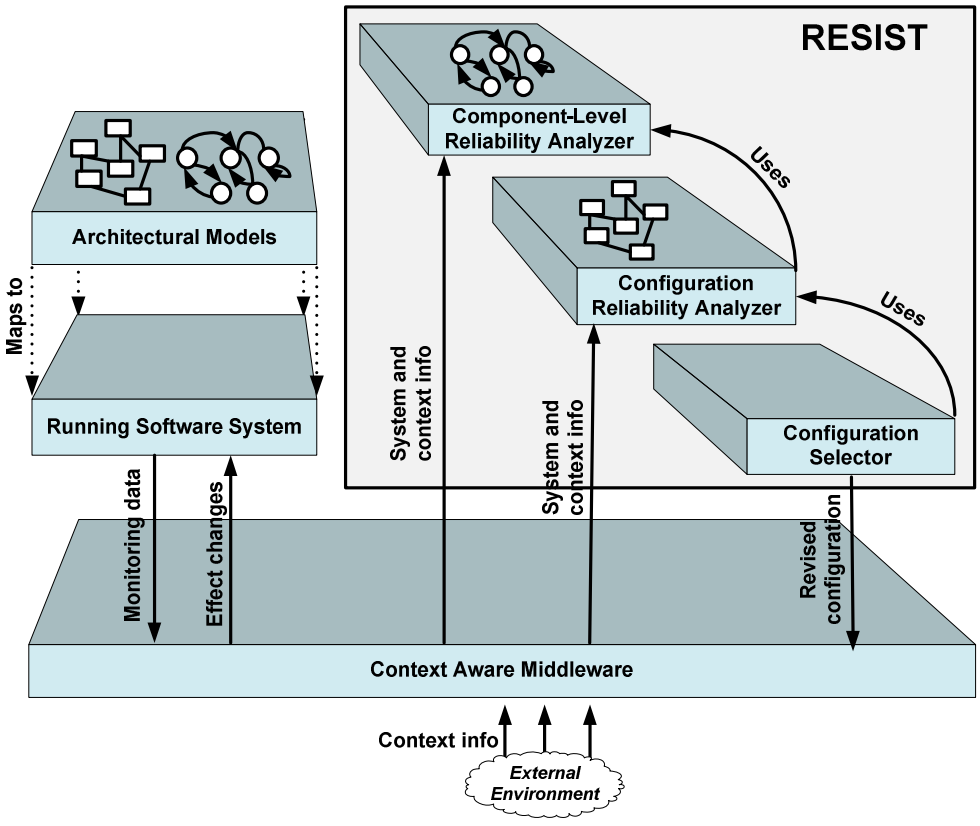

Figure 3. Overview of RESIST framework, which is organized as a feedback control loop that continuously monitors, analyzes, and adapts the system at runtime.

of architecture-based reliability models are developed. These models are used at runtime to assess a variety of configuration choices and to serve as predictors for the future reliability of the system. Unlike the traditional architectural models, they embody contextual properties necessary for reliability analysis of situated systems. As described below, these models are expected to be updated and refined at runtime.

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

Once a new configuration is selected, the *Context-Aware Middleware* adapts the system at runtime to reflect the changes in configuration. The *Context-Aware Middleware* provides support for execution, monitoring, and adaptation of a software system in terms of its architectural constructs (e.g., components, connectors, and configuration). At runtime, the middleware monitors the software system for information that is used to refine the reliability predictions. This information is obtained from multiple sources, such as monitoring internal (e.g., frequency of failures, exceptions, and service requests) and external (e.g., network fluctuations, battery charge) software properties, changes in the

structure of the software (e.g., disconnection of components due to network drop outs, off-loading of components due to drained battery), and contextual properties (e.g., physical location). Since the monitored data represents the most recent operational, structural, and contextual profile of the system's execution, it can be used to assess the system reliability more accurately. Note that unlike previous approaches [7][8][9] we do not rely solely on the monitoring data. Instead, we incorporate architectural knowledge, monitoring data, and contextual changes at runtime in a complementary fashion to produce more accurate results.

5. RELIABILITY AND FAILURE MODEL

RESIST estimates *reliability* as the probability that a system performs its required functions under stated conditions for a specified period of time [10]. In situated software systems, given the ongoing changes in system's operational conditions, the reliability may change over time. We consider a *failure* to be an inconsistent behavior of a system with respect to its specification. *Faults* are caused by *defects* (e.g., software or hardware error), and are abnormal conditions that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function. Thus, faults are causes of failures [10].

Consistent with other architecture-based reliability approaches [12][13][14][15] we assume that the occurrence of a failure is stochastic and that components failure model is *fail-stop*. Failures are thus reliably detectable by middleware facilities. Furthermore, failed components are assumed to eventually (automatically or manually) *recover* and resume normal behavior.

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

within a process exits prematurely, causing the OS process, including all of the components deployed on it, to fail.

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

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 [11][50]. Architectural models are typically *compositional*: structure and behavior of complex systems are described in terms of their constituent components. Despite this however, as identified by recent surveys [12][14][15], 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 *apriori*. Consequently, existing approaches are not suitable for situated systems, where the reliabilities of components and system fluctuate with the *context* in which they operate. A purely 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 while considering behavioral changes both within components, as well as interactions between them.

Therefore, as shown in Figure 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. While RESIST uses architectural

models of the system to facilitate this process, performing architecture-aware reliability analysis enables architecture-based adaptation techniques to be utilized in order to improve or maintain the system's reliability. 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.

In order to perform reliability analysis and prediction, RESIST considers the Software Operational Profile (SOP) of components and the system, which enables it to quantify behavioral properties of the system that affects its reliability at each level. SOP represents the set of executions that take place in a software program along with the probabilities with which they will occur in a given environment [10][48][49]. 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 model these probabilities in the SOP as relevant architectural parameters.

For the purpose of modeling the SOP of components and the system, we use existing techniques based on Discrete Time Markov Chains (DTMC). 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. Once the operational profile of the system in the form of a DTMC has been estimated, the context information is used to update the transition probabilities in the DTMC, so that it reflects the future operational profile. This enables us to capture behavioral changes expected in the system's new environment which in turns enables reliability prediction.

Our reliability models used for component and configuration reliability prediction rely on Hidden Markov Models (HMMs) [16] to estimate the transition probabilities of the DTMC (i.e., the transition probabilities in matrix A mentioned above). As confirmed by our previous results [18], HMMs can be used to learn from runtime data and to obtain transition probabilities. 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 $E = \{e_{ik}\}$, which 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 becomes training data for the HMM.

We use the Baum-Welch algorithm [16] to train and solve the HMM. The training data used as input to this algorithm consists 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 next sections, we elaborate the techniques used to estimate the SOP and how they are used in predicting 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 Figure 2(b). For example, for the robot's Controller, we can obtain the following:

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

where states $S_1, \dots, S_4$ represent the behavioral states: *idle*, *estimating*, *planning* and *moving*, state S_f represents the common *failed* state, and observations $O_1 \dots O_{10}$ represent the transitions between states as shown in Figure 2(b). At runtime, the component is monitored to obtain execution traces in the form of observation sequences (i.e. sequences of state transitions). 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. In order 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. 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 a failure state. Thus the Controller's reliability based on its present context can be computed as:

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

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. In order to arrive at a reliability prediction for a component, RESIST utilizes information from its 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 can occur in the component's SOP as a result of the anticipated contextual change. 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 (recall Section 3). In this case, the architectural parameters are the transition probabilities between states in the in the component's SOP.

Thus, let's 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 parameters 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 $\forall I_n \in I$:

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

$$a'_{ij} = \mu_n(\mathcal{P}(C)) \qquad (2)$$

   where $\mathcal{P}(C)$ denotes the power set of contextual parameters. For example in Figure 2(b), the transition probability from moving to estimating state is directly impacted by the navigational complexity of the robot's environment. Therefore in this case, μ_n correlates the navigational complexity (i.e. a contextual parameter) to the transition probability from moving to estimating state.

2. Given that the transition probability a_{ij} from state S_i to S_j changes as a result of contextual change, 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 Figure 2(b), the transition probability from estimating to failed state does not change as the transition probability from moving to estimating state changes.

3. Given that a_{ij} is a transition probability from state S_i to S_j changes as a result of contextual change, 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 the row remains at 1 since matrix A is a stochastic matrix. For example in Figure 2(b), 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:

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

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 c. In Section 10.5, we demonstrate how the following function was obtained using regression in our experiments on the robotic software.

$$\mu_{bump}(c) = \begin{cases} 0.8196c + 0.00063, & 0 \leq c \leq 0.3 \\ 0.7144c + 0.06916, & 0.3 < c \leq 0.65 \\ 0.5435c + 0.1372, & 0.65 < c \leq 1 \end{cases}$$

In this case, the robot periodically takes snapshots of the environment and using existing techniques [19] determines the complexity of the terrain (the contextual

parameter c), which is correlated to the probability of encountering an obstacle in its path. 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. As will be described in the evaluation, such a function $\mu_{bump}(c)$ which accurately correlates navigational complexity and bump probability can be derived through regression techniques.

Given 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 which represents its *future* behavior:

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

As before, by computing the steady state vector, we can derive the reliability of the component which corresponds to this *future* SOP, which results in a decreased reliability of 0.9826.

6.3. Configuration Reliability Analysis

Once the reliability prediction of all components has 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 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. [20], 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 [20]. A *state* s_i maps to one or more components in concurrent execution whose completion is required in order to transfer control over to the next state. A *state transition* with a probability P_{ij} represents the probability of undergoing a transition from state s_i to state s_j. Accordingly, system reliability R is computed as:

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

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

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

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 equation (4). In order to derive these transition probabilities, a HMM is trained using the Baum-Welch algorithm using the observations obtained at the system level.

In order to construct the HMM for the system's SOP, RESIST utilizes the system's structural model, such as the one depicted in Figure 2(a) for the robot. In this scenario, a fireman interacts with the robot using a PDA. The firemen issues a high-level command (e.g., go into the restaurant and extinguish a fire) which is received by the robot's *Controller* through the *Communication Connector*. The *Controller* executes the appropriate sequence of intermediate actions, which will result in the successful completion of (or inability to complete) the original command, which is sent back to the PDA through the *Communication Connector*. To complete the task, the *Controller* makes use of a variety of *Sensors*, which detect obstacles in its environment and heat, a *Navigator* which performs planning for the command being executed, and a mechanical *Actuator* which is used to perform the physical activities.

The state model in Figure 4(a) 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 *Communication Connector* (*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 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 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. In order to derive the SOP for the system, a HMM is constructed by using the information in the state model. Thus, from Figure 4(a), we can identify the states (i.e. set S) and observations (i.e. set O) for the HMM as follows:

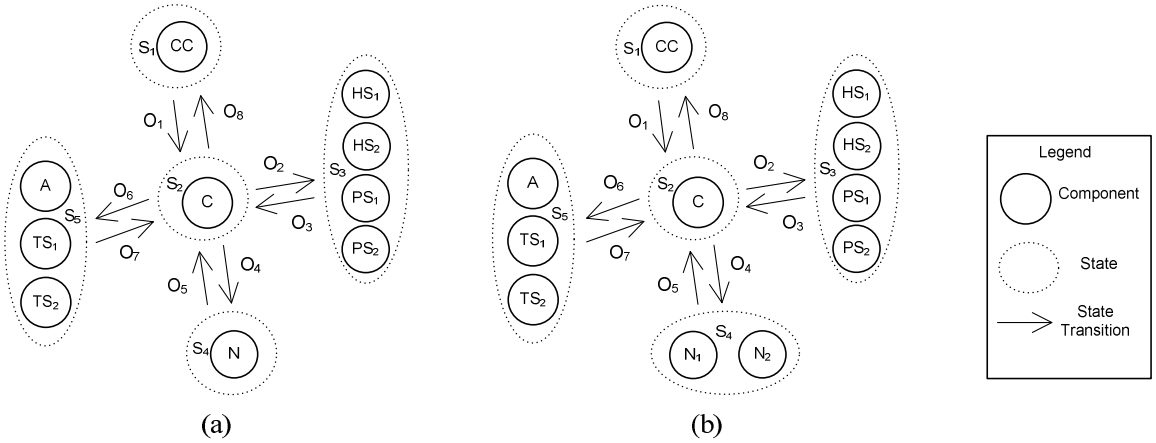

Figure 4. (a) State model for the robot (b) State model with the Navigator replicated.

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

where observations $O_1 \dots O_8$ represent the state transitions between states that result from transfer of control between components (i.e. interactions) as shown in Figure 4(a). The runtime data used to train the HMM consists of these observation sequences, which correspond to state transitions. The following is a transition probability matrix of the HMM derived 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. In order to compute system reliability, a transition

matrix M is derived for the model in equation (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:* C = 0.9915 and *Navigator:* 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% 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 equation (4) yields the system's reliability in its *present* context as 0.9152.

6.4. Context-aware Configuration Reliability Prediction

In order to arrive at a reliability prediction for the system, RESIST utilizes information from its *future* context to determine the behavioral changes that can occur in the near future operation of the system. Similar to the component-level reliability prediction, this analysis is performed by considering the changes that can occur in the system's SOP as a result of the anticipated contextual change. Thus, we update the transition probabilities in A_{System} by utilizing functions of the form (1) to obtain the SOP for its *future* context (recall Section 3).

For the purpose of predicting the system's SOP, we follow an approach similar to the prediction of component's SOP, and model transition probabilities in the system's SOP as architectural parameters. Thus, at each point of interaction $I_n \in I$, the impact of system's contextual parameters C on architectural parameters is given by function μ_n and the system's *future* SOP is derived based on the transition probabilities of the *present* SOP by applying the following two rules $\forall I_n \in I$:

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

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

where $\mathcal{P}(C)$ denotes the power set of contextual parameters. For example, in Figure 4(a) the transition probability from state S_2 to S_3 is impacted by the navigational complexity of the robot's environment. Here μ_n correlates the navigational complexity (i.e. a contextual parameter) to the transition probability from state S_2 to S_3.

2. Given that a_{ij} is a transition probability from state S_i to S_j , which is impacted as a result of contextual change, 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 the row remains at 1, since matrix A is a stochastic matrix. For example in Figure 4(a), 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 will decrease. Thus, all transition probabilities a_{ik} in row i excluding a_{ij} are adjusted so that:

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

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.

$$\mu_{bump}(c) = \begin{cases} 0.125c + 0.2003, & 0 \leq c \leq 0.3 \\ 0.09714c + 0.2142, & 0.3 < c \leq 0.65 \\ 0.07174c + 0.2249, & 0.65 < c \leq 1 \end{cases}$$

Given that the terrain complexity c is expected to increase to 0.45 we can update the transition probability a_{23} based on $\mu_{bump}(c)$ above, and adjust all other transition probabilities in that row using equation (6) to obtain the following SOP for the system:

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

Given 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 equation (4) :

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

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

7. RELIABILITY OF ALTERNATIVE ARCHITECTURES

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

7.1. Impact of Architectural Style

Architectural styles are a set of constraints on the structure and behavior of a system to elicit particular desirable qualities [11][50]. Use of specific architectural styles is a way to apply preconceived solutions to similar recurring software problems. Runtime adaptation and reconfiguration of the system aimed at improving system's quality may often require changes to the system's architectural style. The fault tolerant style, for example, improves reliability by replicating critical components. A fault tolerant connector in the form of middleware can be used to handle component failures and to manage the hot standby copies. In the case of the robot, the original architecture in Figure 1(b) demonstrates the system when the components are allocated to three processes with the *Navigator* and *Controller* components running on separate OS processes. Applying the fault tolerant architectural style in this case can improve the reliability by replicating the *Navigator* component, which represents a critical point of failure. Here, the underlying assumption is that replicas fail independently. Figure 1(c) shows a replicated

Navigator component added to the original architecture while running on a new process. The corresponding state model (Figure 4b) shows the two replicated instances of the *Navigator* N_1 and N_2 both mapped to state S'_4. The reliability of the new state S'_4 can be computed as the probability that at least one *Navigator* component does not fail [20]. Hence the probability of state S'_4 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 S_4 is replaced by the new state S'_4, and the matrix element representing the transition from S_4 (which is nowS'_4) to C increases to 0.9994 from 0.9751. Solving the model above according to equation (4) yields a system reliability of 0.9124. Thus given that in its present configuration, the reliability was predicted to be 0.8736, replication of the *Navigator* results in an improvement of approximately 4.4%.

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 thesis, 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 cause a process failure leading all other components within the process to fail, and

thus impact their reliability. In this case, redeploying components to separate processes could improve a system's reliability. In the case of the robot, consider two deployment configurations of the architecture, one where the *Controller* and the *Navigator* are deployed as two separate processes and another where the two components are deployed as threads sharing the same process.

Let's assume that N and C represent reliability of the *Navigator* and the *Controller* components respectively when they execute on separate processes. When the two components are redeployed to share the same process, the effective reliability of each component is simply $N \times C$, where failure in either N or C will cause both components to fail. For instance, assuming that N and C have been predicted to be 0.9826 and 0.9751 respectively, the effective reliability of the two components would be $N' = C' = 0.9581$. Intuitively, the drop in the two components' effective reliability results in a decrease in the overall system reliability. Therefore, the deployment architecture in which the two components are deployed as separate processes yields better configuration reliability.

8. CONFIGURATION SELECTION

The reliability estimation approach presented earlier can be used to determine the most reliable configuration for a situated software system. However, in practice, reliability estimates are used in conjunction with the estimates of other quality attributes (e.g., efficiency, response time) to determine the *optimal configuration* for the system. As you may recall, the optimal configuration in RESIST is defined as one that satisfies the system's reliability requirement, while improving other quality attributes of concern. In other words, in RESIST, reliability takes precedence over other quality attributes. This is a reasonable objective for the domains targeted by RESIST (i.e., mission critical), but it may not be appropriate for others. Consequently, the configuration selection problem becomes one of an optimization problem. Specifically, RESIST's objective is to find an architectural configuration C^* such that:

$$C^* = argmax_{(C)} \sum_{\forall q \in QualityObjectives} U_q(C) \qquad (7)$$

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

where U_q is a utility function indicating the engineer's preferences for the quality attribute q, R is equation (4) that calculates the expected reliability of a given architecture

C as further detailed below. A utility function is used to perform trade-off analysis between competing (conflicting) quality concerns. In the emergency response system, we would need two utility functions: one specifies the user's preference for improvements in *reliability*, while another one specifies the same for *efficiency*. Elicitation of user's preferences is a topic that has been investigated extensively in the literature (e.g., [21]). RESIST does not place a constraint on the format of utility functions. Arguably any user can specify hard constraints, which can be trivially modeled as step-functions. Alternatively, a utility function may take on more advanced forms (e.g., sigmoid curve), and elicited using the techniques in [21].

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

Thus, for a system with t number of software components (each with a predicted reliability of r_i computed according to the method in Section 6) and h processes, an architectural configuration for the aforementioned optimization problem can be formally specified as follows:

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

The configuration is subject to the following constraints:

- Each component must be placed on a process:

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

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

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

- 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:

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

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

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

Thus, the effective reliability of component i is:

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

where $r_{i_{share}}$ is the effective reliability of component i when the component shares a process with 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 equation (2).

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

Many commonly available algorithms could be used to solve the above optimization problem. For small problems RESIST finds the optimal solution using Integer Programming Solvers. The details of the algorithm used by the solver is outside the scope of this thesis.

9. IMPLEMENTATION AND TOOL SUPPORT

We have developed a prototype implementation of RESIST that integrates an extended version of XTEAM [22] as the environment for maintaining the structural, behavioral, and reliability models, and an open-source HMM toolbox for Matlab. Additionally, we have utilized off-the-shelf tools to perform the runtime reliability analysis and configuration selection.

9.1. Architectural Modeling and Analysis

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. XTEAM uses Finite State Processes (FSP) [37] and xADL [45] for modeling the behavioral and structural properties of a system, respectively.

XTEAM's support for FSP was utilized to implement the state machine model of each component. Additionally, we extended the FSP support to include the specific requirement of RESIST. This includes the ability to define the conditions for failure transitions, and the capability of annotating the behavioral model with the transition probabilities between states in the system. Figure 5 depicts a snapshot of the reliability-annotated FSP models for a subset of the robot's software system.

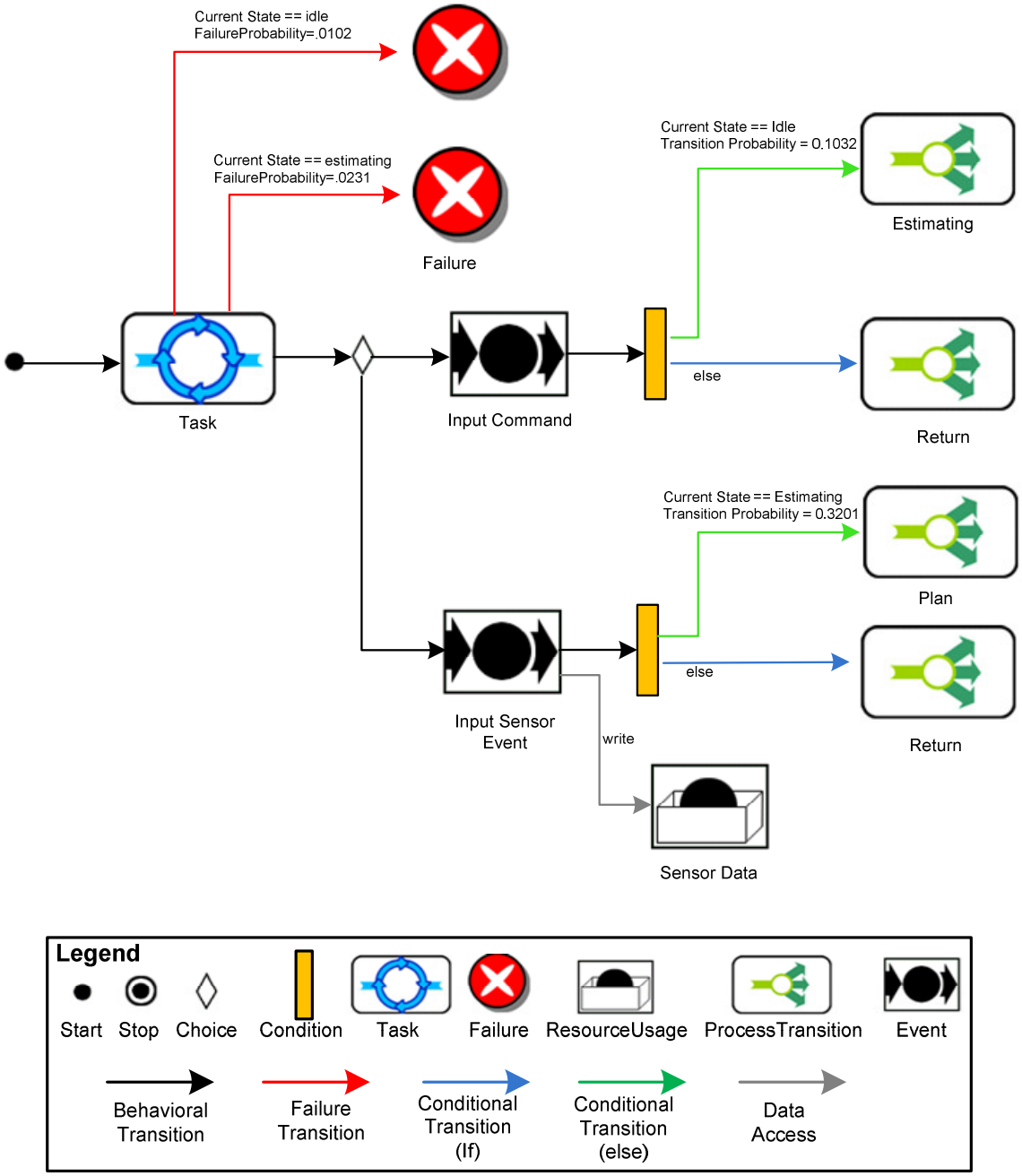

Figure 5. Reliability-annotated architectural model of a portion of Controller's behavioral model. As shown, the annotations in the behavioral model include the transition probabilities into the behavioral and failure states.

We also extended the traditional xADL model support in XTEAM to model reliability properties of the architectural constructs, such as component reliability, and

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

9.2. Simulation and Runtime Monitoring

For the purpose of collecting runtime monitoring data, we used XTEAM to generate simulation code, which was then executed to collect monitoring data consisting of observation data, such as state transitions and component interactions (recall section 6). Executable architectural models were developed for the robotic subsystem using XTEAM, which were programmed to output observation data that correspond to (1) behavioral and failure transitions in the state models and, (2) state transitions in the form of component interactions. This data is written to data files from within the simulation

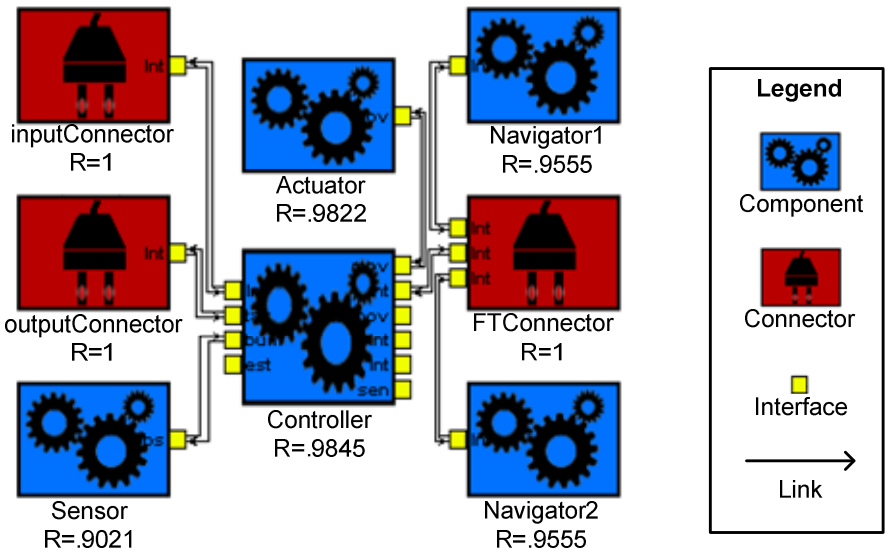

Figure 6. Reliability-annotated architectural model of a portion of robot's structural model. As shown, the architectural constructs such as components and connectors have been annotated with the reliability properties of the system.

code, and serve as input data to the reliability analysis module that performs reliability analysis of the components and the system, as described in Section 9.3.

For simulating failure behavior, we injected faults into the system, which execute as per probabilistic distributions that are available in XTEAM. For example, the following code segment demonstrates an example that forces the system to transition into *failed* state from the *estimating* state with a probability of 0.05:

```
if (NewRandom::uniform(0,1) < 0.05
      && CurrentState -> value().compare("ESTIMATING") == 0)
      NextState -> value()=("FAILED")
```

9.3. Reliability Analysis

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 together with the monitored observation data from the running system is then solved using Matlab's HMM toolbox. An open source Matlab HMM toolbox was used for this purpose [35]. This toolbox provides algorithmic support for Baum-Welch algorithm, steady state vector, etc., which we have used to train and solve the HMM. The estimated reliability values are then used to find an optimal configuration for the system as described in Section 9.5.

9.4. Regression Analysis

We have utilized Matlab's support for regression analysis to obtain functions that correlates the system's context to internal transition behavior (recall Section 6). We have used Matlab's graphical curve-fitting tool to perform regression analysis on data collected from the system [36]. Figure 7 shows an example of the curve fitting toolbox that we have used to perform the regression analysis.

9.5. Configuration Analysis

In order to solve the optimization problem defined in Section 8, we utilized integer programming solver available in Microsoft Excel. Using this tool, the optimization problem is modeled in terms of decision variables and constraints, and solved using the built-in solvers. Since the optimization problem in RESIST is a non-linear integer problem, the solver uses a branch-and-bound algorithm to search for a solution. Figure 8 shows the solver in action before and after the solver is executed.

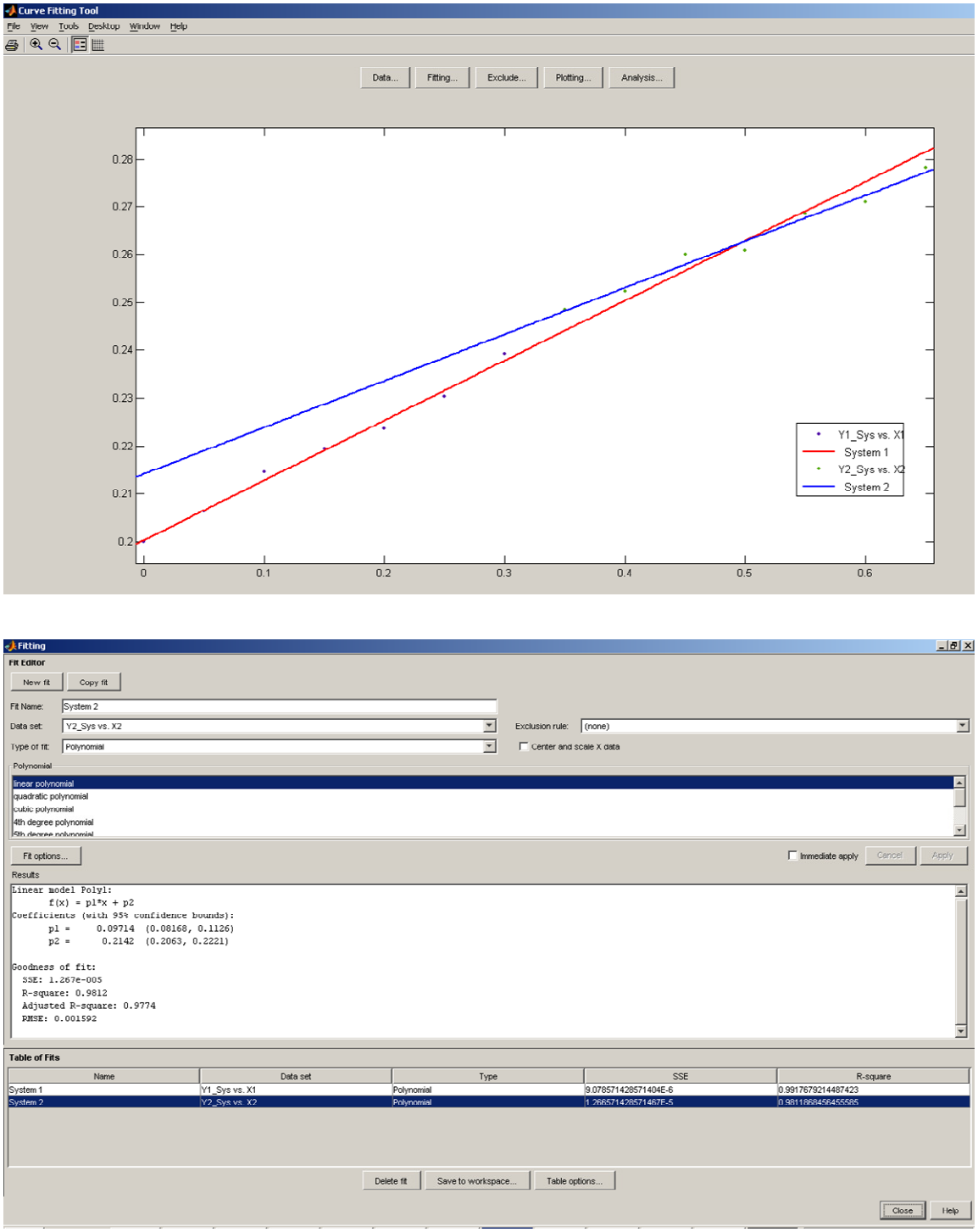

Figure 7. Shown above is the Matlab Curve fitting tool that was used to perform regression analysis in order to correlate navigational complexity of the robot's environment to transition probabilities within the system.

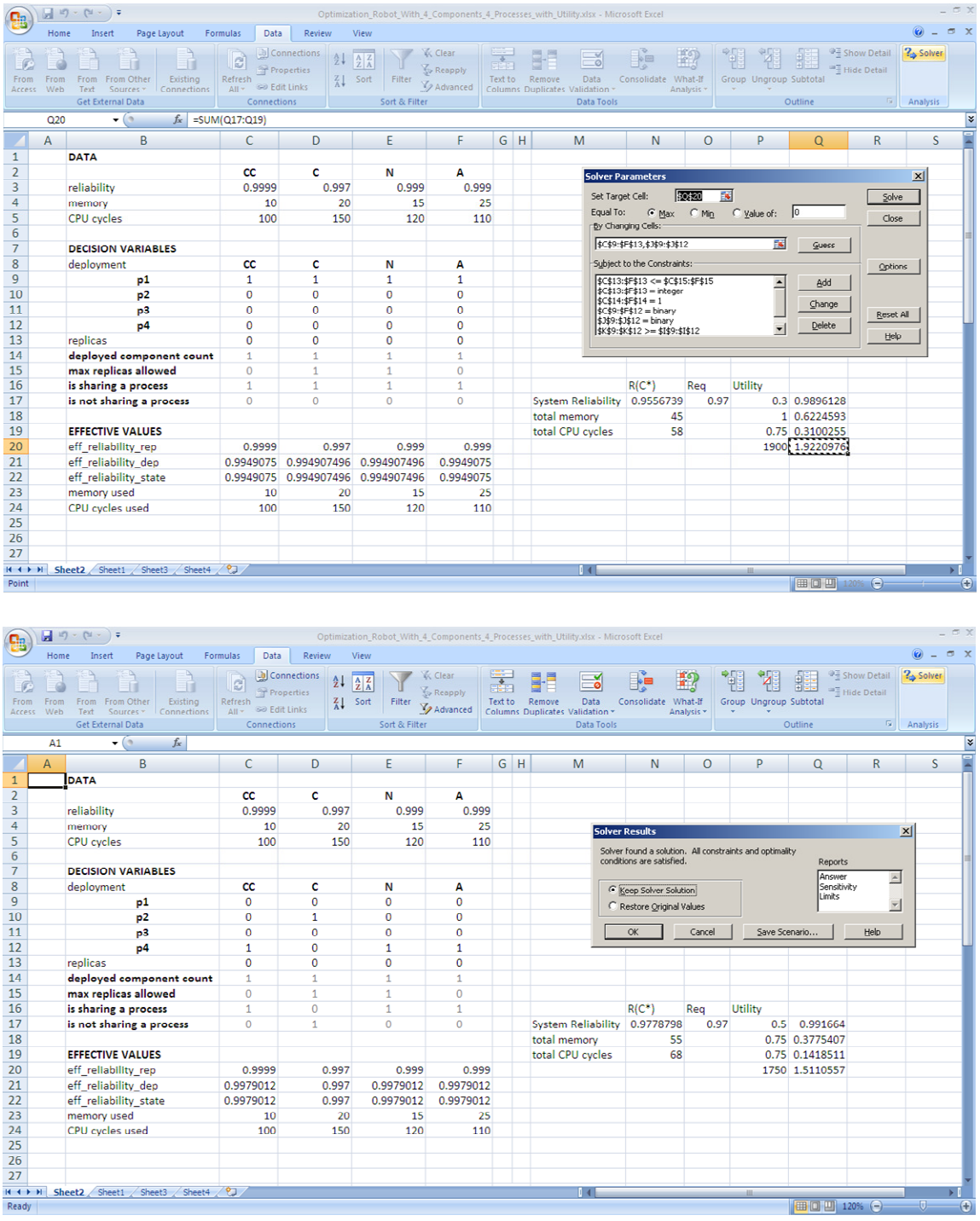

Figure 8. Configuration Analyzer uses Microsoft Excel solver to solve the optimization problem. As shown above, the decision variables, constraints and utility functions are defined in Excel, and solved using the Excel solver.

10. EVALUATION

We have evaluated RESIST using its prototype implementation and the mobile emergency response system described earlier. The evaluation consists of four criteria: (1) the impact of architectural reconfiguration decisions on the reliability of components and the system, (2) the validity of reliability prediction based on expected changes in the context, (3) the effectiveness of proactive system reconfiguration, and (4) the performance overhead of the runtime reliability analysis, and (5) the possibility of accurately correlating a system's context and its architectural parameters. We used XTEAM to control the system's operational profile (i.e., usage) and to gather runtime data. Neither the robotic software nor RESIST was controlled, which allowed them to behave as they would in practice.

10.1. 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.

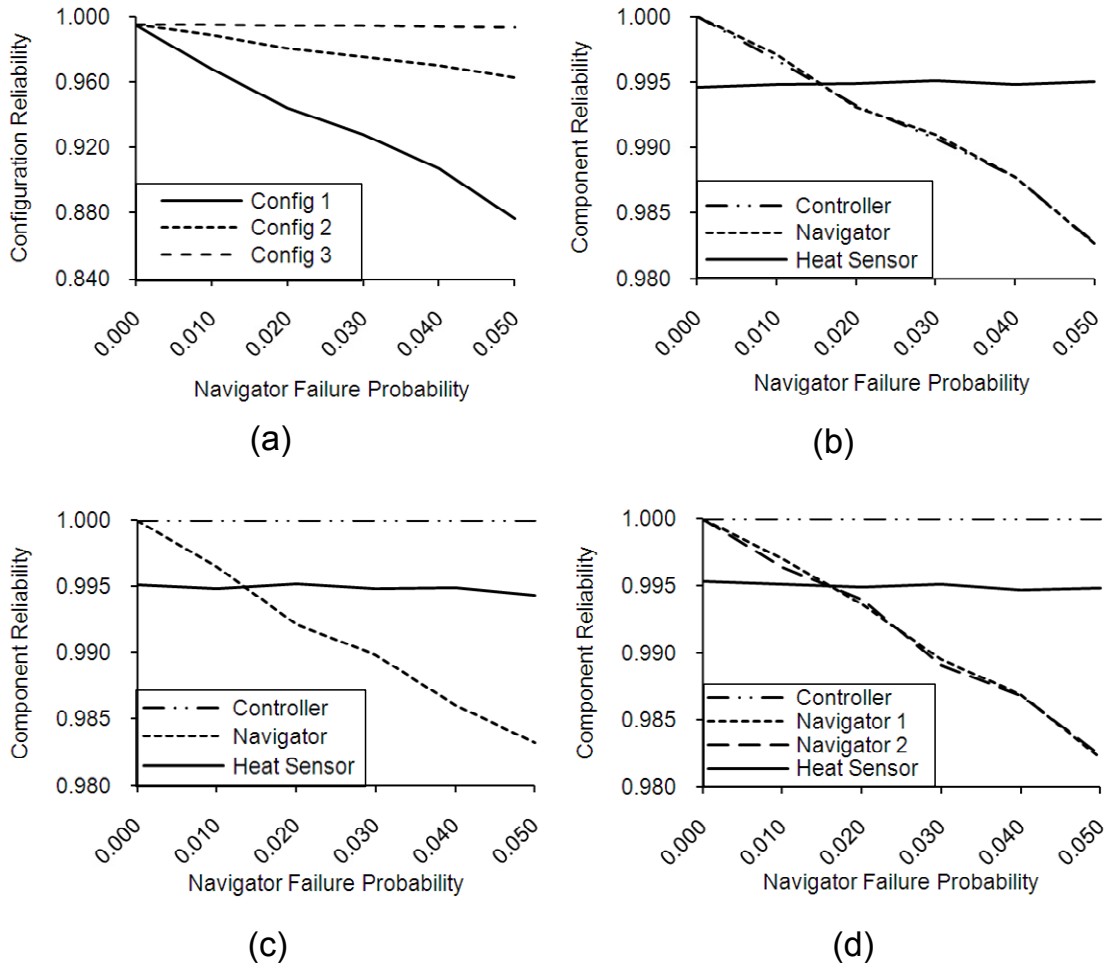

Figure 9. (a) System reliability for 3 architectures; (b), (c), and (d) show component reliabilities for configurations 1, 2, and 3, respectively.

Figure 9 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.

10.2. 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 to the *complexity* of the terrain through which the robot navigates in order to accomplish an assigned task. An increase in the bump probability causes the *Controller* to transition from the *moving* state to the *estimating* state with a higher probability (recall Section 3), thereby altering its operational profile. The techniques presented in [19] together with multi-linear regression were used in our experiments to derive function μ (recall Section 6) that estimates the

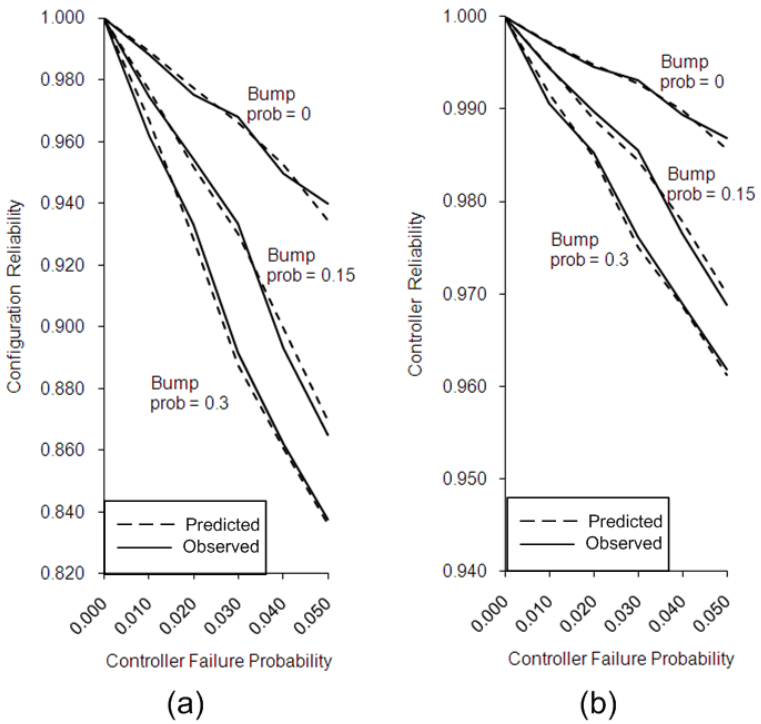

Figure 10. Accuracy of reliability predictions: (a) system reliability (b) Controller's reliability.

impact of change in terrain to change in bump probability with $\pm 2.1\%$ error at 95% confidence level.

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

Figure 10 shows the comparison of predicted reliability and observed reliability in three execution scenarios where different bump probabilities were predicted, and varied the failure probability of the *Controller* component from 0 to 0.05. As shown, the *Controller*'s reliability decreases as the bump probability increases. This is because an increase in transitions to the *estimating* state leads to more failures. Further, the deviation between observed and predicted reliability both at the level of system and *Controller* are extremely small. Note that since the function μ used in the experiment had a 95% likely error bound of 2.1%, small deviation in results is to be expected. However, the deviation is small enough that very accurate adaptation decisions could be made.

10.3. 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 latter successfully maintained the

initial configuration 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% 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.

For the purpose of predicting memory and CPU utilization of a given configuration, we used analytical models where the total memory and CPU utilization are computed in terms 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 $P(C)$:

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

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

where;

$n =$ the number of processes in configuration C

$c =$ number of components in configuration C

$mem_0 =$ average memory required by a process

mem_i = average memory required by component i

mem_{avail} = total memory available

$proc_0$ = average CPU cycles required by a process

$proc_i$ = average CPU cycles required by component i

$proc_{avail}$ = total CPU cycles available

We use sigmoid curve functions to express utility functions for the three quality attributes of concern:

$$U_{Reliability}(R(C)) = \frac{1}{1 + e^{-0.1(100R(C)-50)}}$$

$$U_{MemUtilization}(M(C)) = \frac{1}{1 + e^{0.1(M(C)-50)}}$$

$$U_{ProcUtilization}(P(C)) = \frac{1}{1 + e^{0.1(P(C)-50)}}$$

where $R(C)$, $M(C)$ and $P(C)$ are reliability, memory utilization percentage, and CPU utilization percentage of configuration C.

The global utility function $U_g(C)$ is computed as:

$$U_g(C) = U_{Reliability}(R(C)) + U_{MemUtilization}(M(C)) + U_{ProcUtilization}(P(C))$$

Figure 11(a) 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

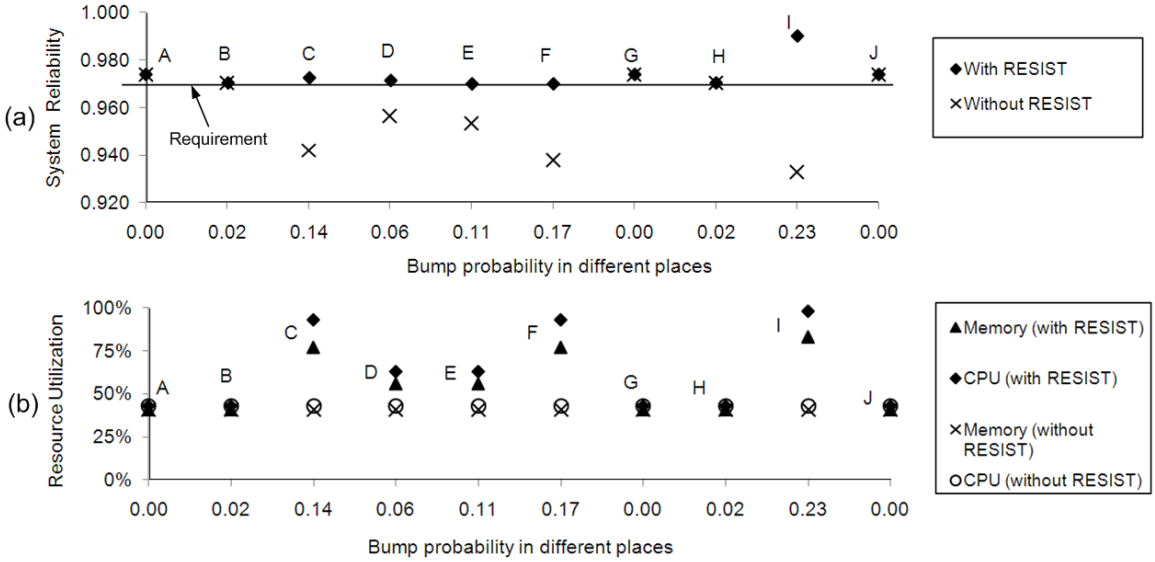

Figure 11. Context-aware proactive reconfiguration. (a) System reliability (b) Resource utilization efficiency.

reliability (since the bump probability increases to 0.14) and proactively adapts the system to maintain its reliability above 97%. As a result, the *Navigator* is replicated and the *Controller* is redeployed to a separate process. This reconfiguration prevents the reliability from falling below the requirement. In contrast, the reliability of the robot without RESIST deteriorates significantly, falling below the 97% requirement.

Figure 11(b) 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% reliability requirement.

10.4. Overhead of Reliability Analysis

Since RESIST is intended to manage situated software systems at runtime, it is important to assess the performance overhead of RESIST's analysis. Table 1 shows the benchmarking results of RESIST's reliability analysis on an Intel Core 2, 2.4 GHz, 2 GB RAM platform, which is representative of the average hardware capability present in modern mobile robots (e.g., [24]). The results show the time it took for performing the reliability analysis for varying number of commands (i.e., tasks sent to the robot). Each command on average resulted in 20 different monitoring observations (e.g., component interface invocations) to be collected and used for training the HMM. The benchmark in

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

Table 1. Execution Time of Reliability Analysis.

the largest scenario, consisting of 2,000 commands and 41,879 observations took 10.45 seconds. However, in practice, our experience with the emergency response robot shows the analysis is often performed on much smaller number of observations, requiring only a fraction of a second for completion.

10.5. Relating Context to Architectural Parameters

We have investigated the feasibility of accurately relating a system's contextual parameters to its architectural parameters by deriving equations of the form (1) described in Section 3. As you may recall, such capability is required for prediction of component and system reliability. In this experiment we used the robotic system and changed its navigational complexity (i.e., a contextual parameter), and observed the changes in the bump probability (i.e., an architectural parameter) in the Controller's operational profile. A correlation of the two parameters was derived by using linear regression.

Figure 12 shows the data points derived for the bump probability, as the navigational complexity of the robot's environment changed. As shown, the robot changes its behavior by employing three navigational strategies (a), (b), and (c), depending on the navigational complexity of its environment. Navigational strategies may correspond to the different types of routing approximation algorithms used by the Controller component as it guides the robot's movement around obstacles. Thus, the relationship between bump probability and navigational complexity is represented in the form of three linear equations, as shown in Table 2. As shown both the goodness of fit error and the percentage error in each of the coefficients are negligible. This set of experiments demonstrated the feasibility of accurately correlating contextual factors to the architectural parameters. We believe a similar approach could be employed in many other domains.

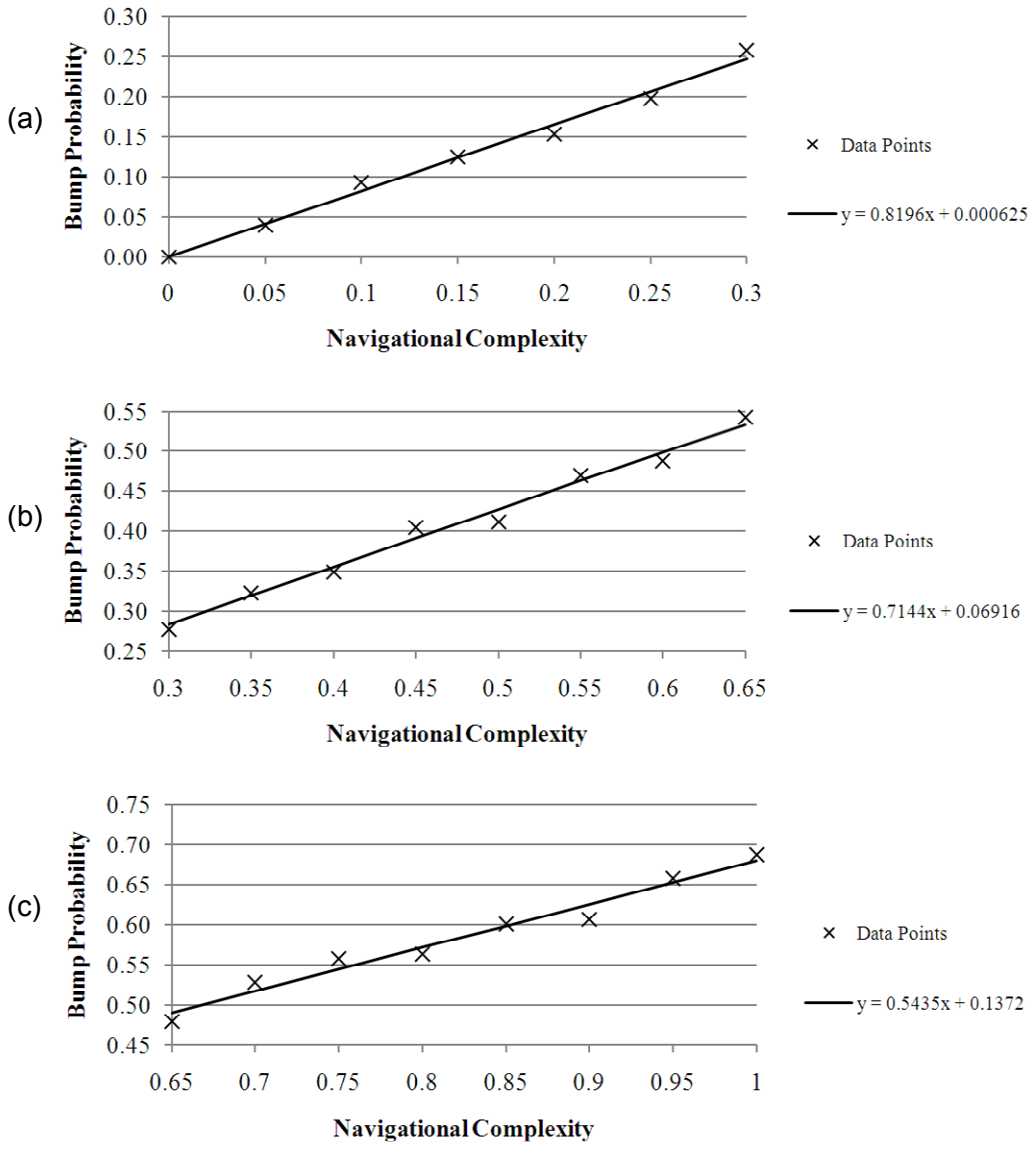

Figure 12. Correlating the navigational complexity to the bump probability of the robot's Controller using linear regression. (a), (b), and (c) show equations derived for three navigation strategies used by the robot.

Navigational Complexity c	**Bump probability $\mu(c)$**	**Goodness of fit Sum of Squares Error**	**Error % of coefficients in $\mu(c)$ (for a 95% confidence level)**
$0 \leq c \leq 0.3$	$0.8196c + 0.00063$	0.0004337	$\pm 2.3\%, \pm 3.1\%$
$0.3 < c \leq 0.65$	$0.7144c + 0.06916$	0.0007125	$\pm 2.9\%, \pm 3.6\%$
$0.65 < c \leq 1$	$0.5435c + 0.1372$	0.0009209	$\pm 2.7\%, \pm 3.3\%$

Table 2. Correlation between the robot's navigational complexity and bump probability.

11. RELATED WORK

In general, architecture-based software reliability analysis and prediction, has been studied extensively by the software engineering research community. However, the methods proposed are not suitable for runtime analysis and for proactive adaptation. In this chapter, we provide an overview of the previous works that have addressed the challenges of analyzing and predicting software reliability, and improving reliability of system through architecture-based adaptation. We also describe other software adaptation frameworks that are related to RESIST, where architecture-based adaptation is utilized in order to improve a system's quality of service. Finally, we described previous work on context-aware middleware intended for situated software systems.

11.1. Architecture-based Reliability Analysis

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. They can broadly be categorized as design-time and runtime analysis.

11.1.1. Design time Analysis

- An approach to reliability analysis described in [13] presents an architecture-level risk analysis framework. The primary purpose of the approach is to help identify the

high-risk components and connectors of a system, based on data that can be collected early in the development process.

- An approach presented in [26] is aimed at reliability analysis of component-based systems. This includes a reliability prediction algorithm which allows system architects to analyze reliability of the system before it is built, taking into account component reliability estimates and their anticipated usage. The approach is intended to guide the process of identifying critical components and analyze the effect of replacing them with the more/less reliable ones.
- System reliability models described in [7][8][20] are targeted towards component-based architectures where system reliability is computed in terms of the reliabilities of its components, (or services provided by components), and the probability of execution. Additionally, [20] presents ways in which a system's architectural style can impact the manner in which a system's reliability is computed, and presents reliability estimation techniques for styles such as pipe-and-filter, batch-sequential and fault-tolerant style.
- An approach presented in [25] is aimed at a scenario-driven approach to computing reliability of software systems early in the lifecycle. Scenarios are partial descriptions of how components interact to provide system level functionality. This reliability model is specifically aimed at concurrent component-based software systems and describes a method to predict software system reliability as a function of component reliability estimates. Scenarios are used to analyze the possible paths in a concurrent software system through the use of probabilistic LTS and FSP. The approach

annotates a LTS graph of the system with probabilities of component failure, and scenario transition probabilities derived from an operational profile of the system. Finally, a Markov-based reliability model described in [38] is used to compute a reliability prediction from the system behavior model.

- [9] Introduces a moving average reliability growth model to describe the evolution of component-based software. In this model, the reliability of a system is a function of the reliabilities of its constituent components. The moving average provides a trend indicator to depict reliability growth movement within the evolution of a series of component enhancements. The input parameters are the components' configurations and individual reliability growths. The output is a vector of moving averaged system reliability growths indicating increasing component enhancement. The application of this model can facilitate cost/performance evaluation and support decision making for future software maintenance.

The underlying assumptions in the above approaches make them unsuitable for use in the domain of situated, dynamic, and mobile systems. Majority of these approaches focus on system-level analysis and assume the reliabilities of the software components are fixed and known. Moreover, many of these approaches assume (sometimes implicitly) that the operational profile of the system is known and does not change at runtime. Finally, none considers the impact of contextual change on the software system's reliability. Three recent surveys [12][14][15] corroborate these observations.

Our past research has addressed some of the uncertainties associated with design-time reliability analysis by incorporating various sources of information [18][27]. We

also identified the challenges of reliability analysis in the mobile domain [28]. 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, in RESIST, we are concerned with runtime reliability of the system and rely on the availability of its implementation. Moreover, we incorporate latest operational and contextual information to predict the system's reliability and proactively place it in the optimal configuration.

11.1.2. Runtime Analysis

Few approaches combine software architecture and reliability analysis using runtime data [29][30].

- An approach presented in [30] addresses the problem of reliability prediction through reliability forecasting. Aimed at distributed computing environments, a statistical model is used for determining the suitable algorithms related to performance requirements for each specific application. While this technique is suitable for traditional desktop systems, it is unsuitable for systems situated in highly dynamic and mobile settings where statistical forecasting offers little help.
- KAMI framework [29] 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.

11.2. Architecture-based Adaptation Frameworks

Run-time adaptation of software systems has been studied extensively by the software engineering research community [2][46]. Of these, the work related to RESIST are the architecture-based adaptation frameworks.

- IBM's Autonomic Computing initiative [39] proposed the MAPE reference model which consists of hierarchically structured feedback control loops. Each loop which is encapsulated within an Autonomic Manager consists of the four phases: Monitor, Analyze, Plan, and Execute. The lowest levels of the Autonomic Managers are responsible for directly managing resources, while the higher levels orchestrate the lower Autonomic Managers in order to meet the system's intended goals.
- A framework presented in [40] by Oreizy et al. describe an architecture-based approach to run-time adaptation and evolution management, in which an explicit architectural model is deployed with the system and is used as a basis for change. Further they highlight the role of software connectors in supporting change management.
- The Rainbow framework [41] present, a style-based approach for developing reusable self-adaptive systems. Rainbow monitors a running system for violation of the invariants imposed by the architectural model, and applies the appropriate adaptation strategy to resolve such violations.

- The Three-Layer architectural model for self-managed systems presented by Kramer and Magee [3] consists of the layers of Component Control, Change Management and Goal Management. The bottom-most layer which is the Component Control layer provides facilities to report the current status of components to higher layers as well as to add, remove components. The Change Management layer reacts to changes reported from the lower layer and executes plans. Goal Management layer, which is the top-most layer produces change management plans in response to requests from the layer below, and in response to the introduction of new system goals.

In contrast to the frameworks described above, RESIST is narrowly aimed at improving the reliability of dynamic situated systems. While none of the existing frameworks directly achieves our objectives, they form the foundation of our research. In fact, our framework is compatible with the widely accepted three layer reference model of self-adaptation [3] as well as the MAPE model [39]. Moreover, in contrast to RESIST, none of the above frameworks are aimed at supporting proactive adaptation of the software system.

In addition the following utility-driven dynamic reconfiguration models are related to RESIST.

- Poladian et al. propose an approach to dynamically configure software based on availability of resources [43]. Given a user's task, the framework selects an appropriate set of services to carry out the task and allocates available computing resources to them. Additionally, the applications or resources assignments are reconfigured as the situation changes.

- A framework for Anticipatory Dynamic Adaptation [44] presented by Poladian et al. is aimed at self adapting a system by allocating resources to applications in *anticipation* of future resource availability. It leverages predictions of future resource availability to improve utility for the user over the duration of the task, rather than reconfiguring reactively. Additionally, the framework considers chooses sequences of configurations over the duration of the task, and maximizes the expected value of utility accrued over the duration of the task.

RESIST primarily differ from the above models it is aimed at arriving at reliability predictions amidst changing context, and placing the system in a more reliable configuration subject to other quality attributes (e.g., resources consumption). In contrast, the above work assumes that the relationship between quality dimensions and resource availability is known *apriori* through the use of application profiles.

11.3. Context-aware Middleware Frameworks

Finally, related is previous research on context-aware middleware intended for mobile and ubiquitous software systems.

- Aura [32] is an architectural style and supporting middleware for ubiquitous computing applications with a special focus on user mobility, context awareness, and context switching. In Aura, a user's task becomes a first class entity, which is represented explicitly in a manner independent from a specific environment. These tasks are represented as a coalition of required services, and the architecture is

equipped with the capability to self-monitor and renegotiate task support as the available resources vary at runtime.

- XMIDDLE [33] is a middleware that supports application engineers to deal with data inconsistency problems caused by mobility, such as low bandwidth, context changes or loss of connectivity. During disconnection, users will typically update local replicas of shared data independently from each other. The resulting inconsistent replicas need to be reconciled upon re-connection. XMIDDLE supports building mobile applications that use replication and reconciliation over ad-hoc networks. XMIDDLE uses reflection capabilities to allow application engineers to influence replication and reconciliation techniques.
- MobiPADS [1] is a reflective middleware that supports active deployment of augmented services for mobile computing. It is designed to support context-aware processing by providing an executing platform to enable active service deployment and reconfiguration of the service composition in response to environments of varying contexts. The adaptation takes place at both the middleware and application layers to provide configuration of resources to optimize the operations of mobile applications.
- Lime [34] 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.
- CARISMA [47], is a mobile computing middleware which exploits the principle of reflection to enhance the construction of adaptive and context-aware mobile

applications. The middleware maintains a valid representation of the execution context, by directly interacting with the underlying network operating system. To enhance the development of context-aware applications, CARISMA provides application engineers with an abstraction of the middleware as a customizable service provider. In particular, the behavior of the middleware with respect to a specific application is described as a set of associations between the services that the middleware customizes, the policies that can be applied to deliver the services, and the context configurations that must hold in order for a policy to be applied.

Unlike RESIST, none of the above frameworks and middleware provides reliability-driven support for optimization of situated software systems through proactive adaptation.

12. CONCLUSIONS

Software systems are increasingly situated in mission critical settings, which present stringent reliability requirements. These systems are predominantly mobile, embedded, and pervasive, which are innately dynamic and unpredictable. In turn, no particular configuration of the system is optimal for the system's entire operational life-time. We presented RESIST, a framework intended to satisfy the reliability requirements, while taking into consideration other quality attributes (e.g., efficiency) through proactive reconfiguration of the software.

12.1. Contributions

The three key contributions of RESIST are: (1) incorporation of multiple sources of information, in particular contextual information, to provide refined reliability predictions at runtime; (2) automatically find the optimal architectural configuration that achieves the appropriate-level of tradeoff between reliability and other quality attributes; and (3) proactively adapt the system by positioning it in the optimal configuration before the system's reliability degrades.

12.2. Future work

In our future work, we intend to evaluate the scalability of RESIST in large-scale software systems comprising of hundreds of components and hardware hosts. We also

intend to increase the types of reconfiguration decisions and dependability tradeoffs that RESIST supports. Finally, we plan to investigate the use of other stochastic approaches (e.g., Dynamic Bayesian Networks, and Hierarchical HMM) and potentially integration with KAMI [29] to support incremental refinement of DTMC parameters, as opposed to periodic assessment of the reliability at runtime.

REFERENCES

REFERENCES

[1]. A. Chan, et al. MobiPADS: Reflective Middleware for Context-Aware Mobile Computing. IEEE TSE, 29(12), Dec. 2003.

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

[3]. J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. ICSE, Minneapolis, MN, May 2007.

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

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

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

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

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

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

[10]. H. Pham, Software Reliability, Springer, 2002.

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

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

[13]. K. Goseva-Popstojanova, et al. Architectural Level Risk Analysis using UML. IEEE TSE, Vol.29, No.10, Oct 2003.

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

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

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

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

[18]. L. Cheung, R. Roshandel, et al. Early Prediction of Software Component Reliability. ICSE, Leipzig, Germany, May 2008.

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

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

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

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

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

[24]. Mobile Robots Inc. http://www.mobilerobots.com/

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

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

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

[28]. S. Malek, et al. Improving the Reliability of Mobile Software Systems through Continuous Analysis and Proactive Reconfiguration. ICSE, Vancouver, Canada, May 2009.

[29]. I. Epifani, et al. Model Evolution by Run-Time Parameter Adaptation. ICSE, Vancouver, Canada, May 2008.

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

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

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

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

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

[35]. MATLAB HMM toolbox
http://www.cs.ubc.ca/~murphyk/Software/HMM/hmm.html

[36]. MATLAB Curve-Fitting toolbox
http://www.mathworks.com/access/helpdesk/help/toolbox/curvefit/cftool.html

[37]. J. Magee, J. Kramer. Concurrency State Models and Java Programming. John Wiley & Sons, 2006.

[38]. R. C. Cheung. A User-Oriented Software Reliability Model. In IEEE Transactions on Software Engineering, volume 6(2), pages 118–125. IEEE, Mar. 1980.

[39]. J. O. Kephart, and D. M. Chess. The Vision of Autonomic Computing. IEEE Computer, vol. 36, 2003, pp. 41-50.

[40]. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Run-time Software Evolution. International Conference on Software Engineering, Kyoto, Japan, May 1998.

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

[42]. P. Oreizy, N. Medvidovic, R. N. Taylor. Runtime software adaptation: framework, approaches, and styles. In Companion of the 30th international Conference on Software Engineering (Leipzig, Germany, May 10 - 18, 2008). ICSE Companion '08. ACM, New York, NY, 899-910.

[43]. V. Poladian, J. P. Sousa, D. Garlan, M. Shaw. Dynamic Configuration of Resource-Aware Services. International Conference on Software Engineering, Edinburgh, Scotland, May 2004.

[44]. V. Poladian et al. Leveraging Resource Prediction for Anticipatory Dynamic Configuration. In Proc. of the First IEEE Conference on Self-Adaptive and Self-Organizing Systems (SASO), Boston, MA, July 2007.

[45]. E. Dashofy, A. van der Hoek. and R. N. Taylor: An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. Proceedings of the 24th International Conference on Software Engineering, pp. 266 - 276, 2002.

[46]. J. Andersson, R. de Lemos, S. Malek, and D. Weyns. "Modeling Dimensions of Self-Adaptive Software Systems." In Software Engineering for Self-Adaptive Systems, Lecture Notes on Computer Science Hot Topics, Springer, 2009.

[47]. L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. IEEE Transactions on Software Engineering, 29(10):929--945, Oct. 2003.

[48]. J. Musa, Software Reliability: Measurement, Prediction, Application. McGraw Hill Software Engineering Series, 1990.

[49]. K. Cai, Software Defect and Operational Profile Modeling. The Kluwer international series in software engineering, 1998.

[50]. R. N. Taylor, N. Medvidovic, E. M. Dashofy. Software Architecture: Foundations, Theory, and Practice. John Wiley, 2010.

CURRICULUM VITAE

Deshan Cooray earned his bachelor's degree in Computer Science and Engineering from the University of Moratuwa, Sri Lanka (2004). Afterwards he was employed at Virtusa Corporation, Sri Lanka (2004-2007) as a Technical Lead where he was engaged in the design and development of enterprise business applications. While a graduate student at George Mason University, he worked at Unisys Corporation where he was involved in designing a SOA based architectural framework for a US Government agency.