NSU NOVA SOUTHEASTERN UNIVERSITY NSU Libraries

Nova Southeastern University
NSUWorks

CEC Theses and Dissertations | College of Engineering and Computing

2014

A Method to Reduce the Cost of Resilience Benchmarking of SelfAdaptive Systems

Steve Hernandez
Nova Southeastern University, steveo.hernandez@gmail.com

This document is a product of extensive research conducted at the Nova Southeastern University College of Engineering and Computing. For more information on research and degree programs at the NSU College of Engineering and Computing, please click here.

Follow this and additional works at: http://nsuworks.nova.edu/gscis_etd

Part of the Computer Sciences Commons

Share Feedback About This Item

NSUWorks Citation

Steve Hernandez. 2014. *A Method to Reduce the Cost of Resilience Benchmarking of SelfAdaptive Systems.* Doctoral dissertation. Nova Southeastern University. Retrieved from NSUWorks, Graduate School of Computer and Information Sciences. (18) http://nsuworks.nova.edu/gscis_etd/18.

This Dissertation is brought to you by the College of Engineering and Computing at NSUWorks. It has been accepted for inclusion in CEC Theses and Dissertations by an authorized administrator of NSUWorks. For more information, please contact nsuworks@nova.edu.

A Method to Reduce the Cost of Resilience Benchmarking of Self-Adaptive Systems

by
Steve O Hernandez

A document submitted in Partial Fulfillment of requirements for the
Degree of Doctor of Philosophy
In
Computer Science

Graduate School of Computer and Information Sciences
Nova Southeastern University

October 29, 2014

We hereby certify that this dissertation, submitted by Steve Hernandez, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

Gregory E. Simco, Ph.D. Date
Chairperson of Dissertation Committee

Francisco J. Mitropoulos, Ph.D. Date
Dissertation Committee Member

Sumitra Mukherjee, Ph.D. Date
Dissertation Committee Member

Approved:

Eric S. Ackerman, Ph.D. Date
Dean, Graduate School of Computer and Information Sciences

SIGNED ORIGINAL ON FILE AT NSU

Graduate School of Computer and Information Sciences
Nova Southeastern University

2014

An Abstract of a Dissertation Idea Paper Submitted to Nova Southeastern University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

A Method to Reduce the Cost of Resilience Benchmarking of Self-Adaptive Systems

By
Steve O Hernandez
October 2014

Ensuring the resilience of self-adaptive systems used in critical infrastructure systems is a concern as their failure has severe societal and financial consequences. The current trends in the growth of the scale and complexity of society's workload demands and the systems built to cope with these demands increases the anxiety surrounding service disruptions. Self-adaptive mechanisms instill dynamic behavior to systems in an effort to improve their resilience to runtime changes that would otherwise result in service disruption or failure, such as faults, errors, and attacks. Thus, the evaluation of a self-adaptive system's resilience is critical to ensure expected operational qualities and elicit trust in their services. However, resilience benchmarking is often overlooked or avoided due to the high cost associated with evaluating the runtime behavior of large and complex self-adaptive systems against an almost infinite number of possible runtime changes.

Researchers have focused on techniques to reduce the overall costs of benchmarking while ensuring the comprehensiveness of the evaluation as testing costs have been found to account for 50 to 80% of total system costs. These test suite minimization techniques include the removal of irrelevant, redundant, and repetitive test cases to ensure that only relevant tests that adequately elicit the expected system responses are enumerated. However, these approaches require an exhaustive test suite be defined first and then the irrelevant tests are filtered out, potentially negating any cost savings.

This dissertation provides a new approach of defining a resilience changeload for self-adaptive systems by incorporating goal-oriented requirements engineering techniques to extract system information and guide the identification of relevant runtime changes. The approach constructs a goal refinement graph consisting of the system's refined goals, runtime actions, self-adaptive agents, and underlying runtime assumptions that is used to identify obstructing conditions to runtime goal attainment. Graph theory is then used to gauge the impact of obstacles on runtime goal attainment and those that exceed the relevance requirement are included in the resilience changeload for enumeration. The use of system knowledge to guide the changeload definition process increased the relevance of the resilience changeload while minimizing the test suite, resulting in a reduction of overall benchmarking costs. Analysis of case study results confirmed that the new approach was more cost effective on the same subject system over previous work. The new approach was shown to reduce the overall costs by 79.65%, increase the relevance of the defined test suite, reduce the amount of wasted effort, and provide a greater return on investment over previous work by a factor of two.

Acknowledgements

This dissertation would not have been possible were it not for the unwavering support and understanding of many individuals.

First and foremost, I must thank my wife Natalie, my other half, the most wonderful and supportive person I could have ever been blessed with. You encouraged, tolerated, and loved me without hesitation or issue. It is because of you that I was able to complete achieve this goal. Thank you for standing by my side and for your guidance and support through the difficult times.

My father, Osvaldo Hernández, and my mother, Sara Rincon, who have supported me throughout my life and made my quest for enlightenment possible, even when it meant theirs were put on hold. A child's achievement is a direct reflection of his parent's love and commitment. I thank you not only for the unwavering support you both have given me but also for defining what a true parent is through your actions. Gracias por todo lo que me has enseñando y por tu apoyo.

My daughters, Sofia, Noelia, and Olivia. I would not have finished this process without having you in my life. Thank you for blessing me. May this small accomplishment pale in comparison with your future achievements.

I was fortunate enough to be welcomed into a study group which became my extended family. They made this process bearable and provided invaluable advice and support throughout. I must personally recognize Dr. Ronald M. Krawitz, my dear friend and colleague, who provided both personal and professional guidance to me. Thank you for everything.

Finally, I must thank the members of my dissertation committee, Dr. Mukherjee and Dr. Mitropoulos, for their valuable feedback throughout the dissertation process. I must especially thank my dissertation committee chair, Dr. Simco, for his patience, guidance, and relentless pursuit of academic and professional excellence without which this dissertation would not have been possible.

Table of Contents

Abstract ii
Table of Contents iii
List of Figures v
List of Equations vi
List of Tables viii

1. **Introduction 1**
   Introduction 1
      Changeloads 2
      Changeload Challenges 2
   Problem Statement 5
      Prior Work 5
         Software Risk Evaluation Steps 6
         Almeida and Vieira Proposed Extension 7
         Contribution Summary 12
      Issues 12
         Vague treatment of System Goals 13
         Vague treatment of Operating Conditions 16
         Cost 19
   Goal 24
   Relevance and Significance 26
   Barriers and Issues 29
   Assumptions, Limitations, and Delimitations 31
   Definition of Terms 32
   Summary 34

2. **Review of the Literature 36**
   Introduction 36
   Benchmarking 36
   Dependability Benchmarking 40
      Faultloads 41
      Faultload Challenges 42
   Self-Adaptive Systems 45
   Resilience Benchmarking 47
   Cost Saving Techniques 48
   Summary 59

3. **Methodology 61**
   Overview of Research Methodology 61
   Approach Overview 62
      Step A: Identification System Goals 63
      Step B: Identification of Obstacles 67

Step B Part 1: Action, Agent, and Assumption Analysis 68
Step B Part 2: Obstacle Analysis 74
Step C: Definition of Obstacle Attributes 77
Step D: Assignment of Obstacle Attributes 80
Step E: Definition of the Changeload 83
Case Study 88
Subject System 89
Analysis of Results 90
Summary 92

4. **Results 94**
Presentation of Data 94
Risk-Based Approach Data 94
Step A: Identification of the Base Scenario 94
Step B: Identification of Change Scenarios 94
Goal-Oriented Approach Data 96
Step A: Identification of System Goals 96
Step B: Identification of Obstacles 96
Step C: Definition of Obstacle Attributes 98
Step D: Evaluation of Obstacle Attributes and Step E: Definition of the Changeload 99
Presentation of Results 101
Results Analysis 102
Cost Savings 102
Effectiveness 103
Wastefulness 105
Return on Investment 106
Summary 106

5. **Conclusions 107**
Implications 108
Recommendations 109
Summary 110

Appendix A 116
Appendix B 119
References 125

List of Figures

Figure 1: Mapping of the phases of the changeload definition with risk analysis phases 7
Figure 2: Algorithm *ReduceTestSuite* for finding a representative set from a group of sets 54
Figure 3: KAOS Glyph Specification 65
Figure 4: Initial Goal Refinement Graph Format 66
Figure 5: Initial Goal Graph of example Self-System A 66
Figure 6: HOW Goal Refinement Graph for example Self-System A 66
Figure 7: WHY Goal Refinement Graph for example Self-System A 67
Figure 8: Expanded Goal Refinement Graph with Actions and Agents Format 70
Figure 9: Expanded Goal Refinement Graph with Actions and Agents example Self-System A 70
Figure 10: Expanded Refinement Goal Graph with Actions, Agents, and Assumptions Format 73
Figure 11: Expanded Goal Refinement Graph with Actions, Agents, and Assumptions for example Self-System A 73
Figure 12: Expanded Goal Refinement Graph with Actions, Agents, Assumptions, and Obstacles Format 76
Figure 13: Expanded Goal Refinement Graph with Obstacles, Assumptions, Agents, and Actions for example Self-System A 76
Figure 14: Goal Refinement Graph of Self-System A – Resource Exhaustion (CPU) Obstacle Impact 82
Figure 15: Goal Refinement Graph of Self-System A – Locked Configuration File Obstacle Impact 82
Figure 16: Considered Obstacles for example Self-System A 87
Figure 17: RAINBOW Framework 89
Figure 18: ZNN.com System Architecture 90
Figure 19: Goal-Oriented Approach Goal Refinement Graph Data 96
Figure 20: Goal-Oriented Approach Expanded Goal Refinement Graph with Obstacles, Assumptions, Agents, and Actions Data 97
Figure 21: Test Suite Relevance Distribution of Identified Changes in the Resulting Test Suites 104

List of Equations

Equation 1: Leung and White (1991) Cost Model 4
Equation 2: Base Scenario Specification 8
Equation 3: Change Specification 9
Equation 4: Change Scenario Specification 10
Equation 5: Modified Change Scenario Specification 10
Equation 6: Changeload Specification 11
Equation 7: Definition of Change 19
Equation 8: Definition of Environment Changes 19
Equation 9: Definition of the Change Space 20
Equation 10: Definition of High-Level System Capabilities 20
Equation 11: Definition of High-Level System Goals 20
Equation 12: Changes Considered by the Risk-Based Approach 20
Equation 13: Enumerated Changes in the Risk-Based Approach 22
Equation 14: Leung and White (1991) Cost Model Strategy Comparison Inequality 25
Equation 15: Simplified Test Suite Cost Comparison Inequality (rewritten) 26
Equation 16: SSR Metric 58
Equation 17: Definition of Low-Level System Goals 63
Equation 18: Low-Level System Goal Definition for example Self-System A 63
Equation 19: Goal Attainment Verification 64
Equation 20: Definition of Self-Adaptive Action 68
Equation 21: Self-Adaptive Action Definition for example Self-System A 68
Equation 22: Definition of Self-Adaptive Agents 69
Equation 23: Self-Adaptive Agent definition for example Self-System A 69
Equation 24: Definition of an Assumption 72
Equation 25: Assumption and Node Satisfaction Relationship 72
Equation 26: Assumption Definition for example Self-System A 72
Equation 27: Definition of an Obstacle 74
Equation 28: Obstacle Satisfaction Relationship 74
Equation 29: Assumption Definition for example Self-System A 75
Equation 30: Obstacle's Shortest Distance to a Goal (OSDG) 78
Equation 31: Obstacle's Breadth of Impact 78
Equation 32: Definition of the Relevance Scale 83
Equation 33: Definition of the Relevance Cut-Off 85
Equation 34: Changeload Definition 86
Equation 35: Changeload Definition for example Self-System A 86
Equation 36: Reduced Test Suite Cost Inequality 91
Equation 37: Test Selection Strategy's Return on Investment 92
Equation 38: Cost Savings Inequality Results 102
Equation 39: Cost of Testing Strategies 116
Equation 40: Cost Savings Inequality as proposed by Leung and White (1991) 116
Equation 41: Cost Savings Inequality with specific costs and different Selection Costs 116
Equation 42: Analysis and Understanding Costs Equivalence 117

Equation 43: Simplified Savings Inequality with different Selection Costs 117
Equation 44: Reduction of Cost Terms 117
Equation 45: Simplified Test Suite Cost Comparison Inequality 118

List of Tables

Table 1: Fault-Space Optimization Results 50
Table 2: Run-times for ReduceTestSuite for Actual and Constructed Associated Testing Sets 55
Table 3: Experiment 1 - Reduction during Program Development 56
Table 4: Experiment 2 - Reduction during program maintenance for performance improvement 56
Table 5: Reduction during program maintenance for program enhancements 57
Table 6: Test Case Coverage Matrix 58
Table 7: Change Scenario Attributes defined in the Risk-Based Approach 77
Table 8: Risk-Based Change Scenario Impact Attribute mapping to Goal-Oriented Obstacle OSDG Attribute 79
Table 9: Risk-Based Change Scenario Probability Attribute mapping to Goal-Oriented Obstacle OB Attribute 79
Table 10: OSDG Attribute for example Self-System A 80
Table 11: OB Attribute for example Self-System A 80
Table 12: Relevance Level Numeric Mapping 84
Table 13: Exposure Matrix for the Goal-Oriented Approach 84
Table 14: Exposure Matrix for example Self-System A 84
Table 15: Exposure Matrix with Cut-Off Level Applied 85
Table 16: Exposure Matrix with Cut-Off Level Applied for example Self-System A 86
Table 17: Concrete Obstacles in the final Changeload generated by the Goal-Oriented Approach 87
Table 18: Final Changeload with Concrete Obstacles for example Self-System A 88
Table 19: Risk-Based Approach Base Scenario Definition Data 94
Table 20: Risk-Based Approach Change Class and High-Level Change Mapping to Base Scenario Elements Data 95
Table 21: Risk-Based Approach Change Scenario Definitions Sample Data 95
Table 22: Expanded Goal Refinement Graph Composition Summary Data 98
Table 23: Goal-Oriented Approach OSDG Attribute Data 98
Table 24: Goal-Oriented Approach OB Attribute Data 98
Table 25: Goal-Oriented Approach Final Changeload with Concrete Obstacles Results 100
Table 26: Test Suite Construction and Total Size Comparison Results 101
Table 27: Included Change Scenarios and Final Changeload Size Comparison after Cut-Off Results 101
Table 28: Test Suite Relevance Distribution Summary 104
Table 29: Risk-Based Approach Change Scenario Definitions Full Results 124

Chapter 1

Introduction

Introduction

The growing heterogeneity, scale, and dynamism of modern systems has the research community and industry turning to self-adaptive systems to deal with their resulting complexity and unmanageability (Almeida & Vieira, 2012a; Ganek & Corbi, 2003). The autonomic functionality of self-adaptive systems reduces the burden on human operators to manage, configure, and troubleshoot them as they can self-configure, self-optimize, self-heal, or self-protect to internal and external changes with greater speed and precision and with little or no human intervention (Almeida & Vieira, 2011; Ganek & Corbi, 2003; IBM, 2003). The goal of resilience benchmarking is to evaluate and validate a system's persistence of service delivery in the presence of changes (i.e. its resilience) in a reproducible and cost-effective manner (Almeida & Vieira, 2012a). However, there are several open research challenges related to resilience benchmarking, with the definition of a representative changeload being the most obscure.

Almeida and Vieira (2012a) proposed a risk-based approach that reduced the considered change space and identified the relevant changes to include in a representative changeload. However, not all included changes fulfilled the purpose of disturbing the system and evoked its adaptive capabilities, resulting in a high cost of benchmarking. This study addressed this issue by extending the risk-based approach to utilize system knowledge to further reduce the considered change space and overall cost of resilience benchmarking.

The rest of this section introduces the changeload and discusses the open research challenge of defining a balanced and cost-effective changeload.

Changeloads

Resilience benchmarking requires the inclusion of a well-defined and relevant set of changes that include the runtime system dynamics that are not considered in traditional dependability evaluation (Almeida & Vieira, 2012a; A. B. Brown et al., 2004; Madeira et al., 2002; Meyer, 2009; Salehie & Tahvildari, 2009). The workload and operating environment cannot be static and must include changes that employ the SUB's self-adaptive capabilities as real-world operating conditions would (Almeida & Vieira, 2011; A. B. Brown et al., 2004). Changes include faults, attacks, failures, expected and unforeseen variations of internal (e.g. resource exhaustion, availability of new features) and external (e.g. network congestion, sub-system changes) contexts of a system, or its components, that may impact its ability to maintain runtime goals (Almeida & Vieira, 2012a; Huebscher & McCann, 2004). Therefore, the changeload must model the fluctuations and variations of the system's overall stress to provide a realistic use-case for evaluation purposes (Almeida & Vieira, 2012b).

Changeloads encompass faultloads, extend their modeling, and their application, to characterize the dimension of change within dynamic systems. Thus, they share several open research challenges, which are discussed below.

Changeload Challenges

Defining a relevant changeload for the evaluation of self-adaptive system resilience is a daunting research challenge due to the complexity of self-adaptive systems and the large number of potential changes that may impact their attainment of goals, which may also be dynamic at runtime (Almeida & Vieira, 2011; Andersson, Lemos, Malek, & Weyns, 2009; Bondavalli et al., 2009; Brun et al., 2009; B. Cheng et al., 2009; Salehie & Tahvildari, 2009). Defining a resilience benchmark for all system-types is an unachievable goal (Almeida, Madeira, & Vieira,

2010), therefore, the benchmarking domain is divided to reduce the problem space into tractable and tenable segments (Bondavalli et al., 2009). However, this is a difficult task as the domain boundaries may not be obvious, such as components, systems (e.g. large, complex, or distributed systems), runtime behavior, and application types (Bondavalli et al., 2009).

While defining changeloads utilizing field data is ideal, accessing such data may not be possible for many systems as runtime changes may not be recorded or shared due to intellectual property concerns (Almeida & Vieira, 2012a). Evaluators experience the same challenges defining changeloads as they do with faultloads, specifically the lack of strict and systematic approaches for their definition (Moorsel et al., 2009) and an absence of standardized metrics and procedures for their utilization (Almeida & Vieira, 2011; Bondavalli et al., 2009). This leads to a reliance on unstructured expert analysis, the utilization of inconsistent field data, the inclusion of loosely related reports, and ad-hoc / system-specific evaluations that increase the overall cost of resilience benchmarking (Almeida & Vieira, 2012a; Barbosa, Vinter, Folkesson, & Karlsson, 2005; Moorsel et al., 2009; Xavier, Hanazumi, & Melo, 2008) by incorporating test cases that are repetitive, irrelevant, and unrepresentative (Barbosa et al., 2005; Jorgensen, 2002). For example, Barbosa et al. (2005) demonstrated that ineffective faults can account for up to 85% of a defined faultload for memory and CPU bit-flip faults.

Identifying the most realistic and relevant changes from the change space is particularly challenging due to the consideration of the many dimensions of variability (such as those affecting resources, interfaces, hardware, and so on) that directly and indirectly affect the SUB's runtime behavior (Almeida & Vieira, 2012a; B. Cheng et al., 2009) while ensuring they are sufficiently representative, reproducible, scalable, portable, and cost-effective (Almeida & Vieira, 2011; Bondavalli et al., 2009; Moorsel et al., 2009). A system's change space extends

the fault space, which is typically extremely large (Barbosa et al.), by encompassing any and all possible variations in the operating environment, internal conditions, inputs, workloads, faultloads, attackloads, and user interactions, or sequences and combinations thereof, that may subject the system to any type of stress which may or may not result in failure (Almeida et al., 2010; Almeida & Vieira, 2012b). The high degree of complexity and runtime dynamics of self-adaptive systems and their environments (Bondavalli et al., 2009; Ganek & Corbi, 2003) makes the number of potential runtime changes virtually unbounded (Almeida & Vieira, 2012b).

The cost of benchmarking is directly related to the number of test cases (e.g. faults or changes) that are considered, included, and ultimately enumerated in the benchmarking process (Cin et al., 2002; Xavier et al., 2008). This relationship can be shown by using the cost model defined in Equation 1, where the total cost of a software testing strategy, $C(Strategy)$, against a set of test cases, T, is comprised of the costs of system analysis, Ca, test selection, Cs, test execution, Ce, result analysis and understanding, Cu, and result checking, Cc.

$$C(strategy) = Ca(T) + Cs(T) + Ce(T) + Cu(T) + Cc(T)$$

Equation 1: Leung and White (1991) Cost Model

Thus, the cost of attaining full change space coverage by utilizing an exhaustive changeload is impractical and unreasonably expensive due to the extremely large change space (Almeida & Vieira, 2012b; Barbosa et al., 2005).

More practical approaches were required to enable the reproducible definition of changeloads consisting of a minimal set of changes required for resilience benchmarking of self-adaptive systems (Almeida & Vieira, 2012a) as it remained labor intensive and costly (Moorsel et al., 2009). The lack of standarized methods resulted in the challenges described were

addressed by the Almeida and Vieira (2012a) and their risk-based approach. Their contribution is described in the following section followed by a description of this study's goal.

Problem Statement

Resilience benchmarking of self-adaptive systems is critical due to their use for mission critical and infrastructure services. However, benchmarking and testing is often avoided due to the high cost and labor required to identify all of a system's potential runtime changes and test the system against them (Quadri & Farooq, 2010). Almeida and Vieira (2012a) proposed a method for identifying relevant changes and defining resilience changeloads for self-adaptive systems. However, their technique suffered from high cost due to the consideration of the entire change space caused by the use of vague constructs for the system's goals and operating conditions. This study extended prior work and addressed the problem of high evaluation costs and labor associated with resilience benchmarking of self-adaptive systems by utilizing system knowledge to reduce the considered change space. The following section presents the risk-based approach followed by a discussion of the approach's limitations.

Prior Work

Almeida and Vieira (2012a) proposed a risk-based approach for defining changeloads in which Software Risk Evaluation (SRE) techniques were extended and adapted to identify and analyze the potential risks to the self-adaptive system goals. The techniques were borrowed from the identification and analysis phases of SRE, which focus on identifying and characterizing the risks that may prevent a development team from accomplishing project goals. The original SRE steps are outlined below followed by the Almeida and Vieira (2012a) extension.

Software Risk Evaluation Steps

The first step in SRE is to define a general criterion against which the results of changes can be measured prior to the project's commencement, called the Threshold of Success (ToS), which defines the boundary between success and failure of the project. Next, the risks to the ToS are captured in risk statements, written in prose, that include the negative conditions under which the project may be classified as unsuccessful. The risk statements are elicited in a condition / consequence format that describes the potential conditions, or circumstances, which cause anxiety to project participants and their negative consequences. Risk attributes are then defined to provide greater understanding of risk conditions and their consequences and serve as a useful method for their prioritization. Risk attributes typically include the impact of the risk to the ToS (e.g. Catastrophic, critical, or marginal), timeframe of identification (e.g. Long, medium, or short), and its probability of occurrence (e.g. High, medium, or low). Once general risk attributes have been identified, they are associated with the risk statements (from the first step) and assigned attribute levels (e.g. Catastrophic impact, Short identification interval, Low probability of occurrence). Finally, the identified risks are prioritized based on their associated attributes. The prioritization can be done using a multi-voting scheme, Pareto Top-N (risk exposure cut-off such as impact vs. probability), or comparison ranking (using pair-wise comparison of defined risk statements).

Ultimately, the definition and prioritization of risks associated with a project rely almost exclusively on the experience of the involved experts. These activities are typically conducted using free-form brainstorming (i.e. informally) or utilizing a taxonomy of risks and determining their applicability to the specific project as defined by the Software Engineering Institute (SEI)'s Taxonomy of Software Development for risk identification (Almeida & Vieira, 2012a). The project's personnel use this information to create risk management and mitigation plans for the

identified risks in an effort to ensure the project's successful completion (Williams, Behrens, & Pandelios, 1999).

Almeida and Vieira Proposed Extension

Almeida and Vieira (2012a) extended and adapted the SRE steps to resilience benchmarking by applying the identification and analysis techniques to a self-adaptive system's operation. Specifically, they adapted the threshold of success (ToS) definition, applied the SRE risk categorization and prioritization to the SUB, and then mapped the SRE risk analysis phases to the changeload definition process, as depicted below in Figure 1.

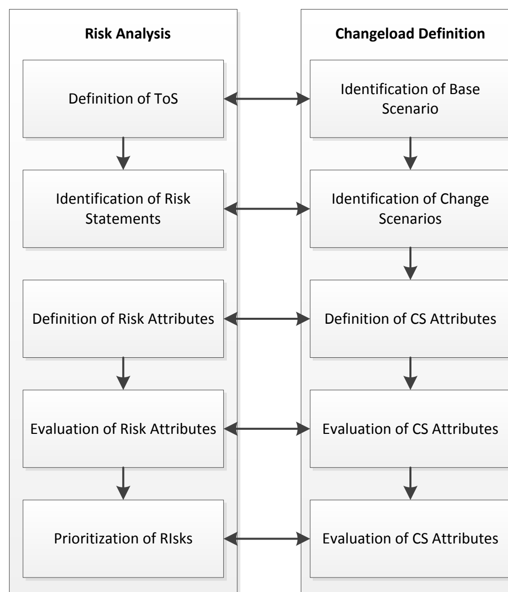

Figure 1: Mapping of the phases of the changeload definition with risk analysis phases

Of particular importance is the identification of the basic drivers of the SUB, or its high-level goals, which is vital for the identification and characterization of the change scenarios, described below (Almeida & Vieira, 2012a). Almeida and Vieira (2012a) argued that detailed descriptions of the SUB's goals, workload, and operational conditions are not necessary to define the changeload. Instead, abstract characterizations of these elements are all that is needed, though they did concede that having detailed descriptions of the goals might assist the

changeload definition process. Thus, the first step of the changeload definition process is to identify and prioritize the generic goals of systems in the benchmark domain (i.e. the specific system-type) in an effort to cope with the diversity of applications and guide system analysis, as conducted in dependability and performance benchmarking (Madeira et al., 2002; Moorsel et al., 2009). An example used within their study, and throughout this paper, is adaptive database management systems (ADBMS) which are typically governed by the following prioritized goals: throughput, availability, and response time (Almeida & Vieira, 2012a).

The ToS was defined as the base scenario under which all identified goals are maintained, a typical workload is executed, and the operational context of the SUB is static; essentially its "golden run". It is independent of the changeload and offers a baseline against which the system evaluator can compare metric values obtained in the presence of the changeload. The base scenario specification is defined below in Equation 2:

$$Base_Scenario = \left\{ workload_{typical}, operating_conditions_{typical}, goals_{fixed} \right\}$$

Equation 2: Base Scenario Specification

The specification defined the base scenario as a set of three elements: the typical workload, $workload_{typical}$, the typical operating conditions and resources (hardware and software) within which the goals are obtained and the workload executed, $operating_conditions_{typical}$, and the fixed goals of the SUB, $goals_{fixed}$. The goals were predefined by a Service Level Agreement (SLA), were fixed, or defined by some other specification that described the attributes or requirements the SUB must fulfill during runtime (e.g. minimize response time, maximize throughput).

The SRE procedure for identifying risk statements was utilized to define change scenarios. The change scenarios are derived from the base scenario and defined a set of each possible representative change, or sequence of changes, that may affect the SUB's ability to achieve and maintain the runtime goals specified in the base scenario. To identify the relevant classes and types of changes Almeida and Vieira (2012a) proposed the following methodology:

1. Identify and select the potential sources of changes, which may include internal or external hardware, software, and operational environment.
2. Identify the classes of changes that may originate from the previously identified change sources. For example, an ADBMS whose potential source of change are its human operators, may have a potential change class of "administrative mistakes" or "variation in service requests."
3. Identify the specific types of changes that may impact the base scenario's defined goals. For example, an ADBMS may have a specific change of "increase in the number of requests per second" for the "variation in service requests" change class.

The change specification is shown in Equation 3:

$$Change = \{source, type\}$$

Equation 3: Change Specification

The source of the change, *source*, and the change type previously defined, *type*, represent a single change the SUB may experience. The evaluator then converts the defined changes into concrete system changes once the relevant classes and change types have been defined for the SUB. For example, "increase in the number of requests" can be converted into a more specific change, such as "15% increase in requests per second."

The specific changes are specified using the following format (added for clarity and not included in the original specification), $specific_change = (change, ti, duration, amount)$, where the trigger instant, ti, determined the predefined instant the SUB would be subjected to the change, $duration$, which specified the amount of time it was affected by the particular change, and the relative quantity, $amount$, of the change to subject the SUB. Examples of change amounts are "50% available throughput", "100% connectivity loss", and a "90% reduction of available memory" (Almeida & Vieira, 2012a). These additional details are required to ensure each change scenario is unique as the same change triggered at different moments may result in different behaviors depending on the SUB's context (Almeida & Vieira, 2011, 2012a; B. Cheng et al., 2009; Huebscher & McCann, 2004).

The evaluator then used the set of specific changes in the change scenario specification, outlined below in Equation 4, and more clearly in Equation 5, where the change scenario is a set of specific changes that are experienced by the SUB from a base scenario context.

$$Change_Scenario = \left\{ \begin{matrix} Base_Scenario, \\ \{(change, ti, duration, amount)\} \end{matrix} \right\}$$

Equation 4: Change Scenario Specification

$$Change_Scenario = \{Base_Scenario, \{specific_change\}\}$$

Equation 5: Modified Change Scenario Specification

Change scenario attributes were then defined in a similar fashion as risk attributes, where expert analysis and voting schemes were utilized (in the absence of available field data) to assign relative impact and probability to each change scenario. The association of change scenario attributes provided a manner of characterizing each change scenario and a means of establishing their relevance.

The changeload was then defined by selecting the most relevant and representative change scenarios defined in the previous steps. To facilitate this process the authors proposed using expert judgment, a multi-voting scheme, or creating an exposure matrix and defining a cut-off level. The later consisted of a matrix based on two or more dimensions of change scenario attributes and their associated scales. Then the change scenario attributes were correlated and an associated level of representativeness (e.g. “Mandatory” inclusion in the changeload, “Very High” representativeness, etc.) was assigned to each potential combination of attributes. The evaluator then defined the cut-off level as the minimum level of representativeness a change scenario had to possess for inclusion in the changeload, which followed the initial definition of the ToS.

For instance, all scenarios with a “Medium” or higher representativeness ranking were included. In that case a scenarios with “Very High” probability of occurrence, a “Catastrophic” impact, and a “Mandatory” ranking would be included in the changeload, while a change scenario with a “Low” probability of occurrence, “Marginal” impact, and “Low” representativeness ranking would be omitted as its attributes did not warrant the resource investment in its evaluation. Finally, the changeload was defined as a set of the most relevant and probable change scenarios, depicted below in Equation 6.

$$ChangeLoad = \{ChangeScenarios\}$$

Equation 6: Changeload Specification

An important consideration is that the order of the change scenarios that comprised the changeload was significant as each variation may result in significantly different adaptive behavior (Almeida & Vieira, 2012; B. Cheng et al., 2009). The evaluator then took the changeload specification and implemented the changes for the specific system. That is, the

changeload and its corresponding change scenarios were converted into executable code that the benchmarking system could execute against the SUB. Almeida and Vieira (2012a) presented a simple case study of an adaptive database management system (ADBMS) to demonstrate the applicability of their approach.

Contribution Summary

The Almeida and Vieira (2012a) approach provided a procedure for identifying the potential risks associated with a system without utilizing any details of the target system or its self-adaptive capabilities. The procedure utilized a step-wise refinement approach, starting with high-level generic context and using deductive reasoning to develop more detailed descriptions of risks to the SUB's general goals. Their approach also provided a specification with which to develop a standardized changeload definition. The specification provided a methodological approach to defining change within dynamic systems.

Issues

As previously mentioned, defining a relevant changeload for benchmarking the resilience of self-adaptive systems has several open research challenges. Almeida and Vieira (2012b) identified the most pressing issues of changeload definitions, which included the selection of specific changes that exercise the adaptive mechanisms of interest within a system, the reduction of the considered change space due to the exponential growth in the number of changes that a system may encounter, the identification of the relevant sequences of changes to mimic their occurrence in the real-world, and the definition of the specific timing and scheduling of change injection into the SUB (and workload) to represent real-world operating conditions. Their approach addressed the identification of relevant changes and the reduction of the considered

change space issues by adapting established software engineering and project management techniques to identify and analyze potential risks to include in a changeload.

The risk-based approach suffered from several shortcomings, including the utilization of highly abstract goals and operating conditions to determine the drivers, and ultimately the behavior, of the SUB. The use of vague and high-level constructs lead to several challenges in the identification and analysis phases of the approach, the process of identifying the SUB's context, and the associated changes that may affect it (B. Cheng et al., 2009). These shortcomings resulted in the consideration of an extremely large change space which significantly increased the benchmark's scope, overall time and labor required to conduct it, and the total cost of the benchmarking procedure (Pressman, 2005). The following section provides a discussion of each of the shortcomings listed above followed by a summary of the resulting issue present in the risk-based approach.

Vague treatment of System Goals

Almeida and Vieira (2012a) stated that the identification of the SUB's goals is the most important aspect of defining its base scenario, and ultimately a relevant changeload, as the base scenario is the baseline from which all changes are identified and against which all self-adaptive values are compared. The authors affirmed that only a high-level understanding of the SUB's generic goals was sufficient and that detailed knowledge was not necessary, though it may aid the process.

Further, Almeida and Vieira (2012a) postulated that only a high-level understanding of typical goals of the class to which the SUB pertains was required, and that this provided sufficient information to identify runtime changes that would effectively evaluate a self-adaptive system's resilience. However, the use of high-level goals to define changeloads did not provide

sufficient insight into the SUB to allow analysis of its runtime behavior, discovery of the specific causes of system change, and the characteristics of the SUB's response using engineering principles (B. Cheng et al., 2009). This was caused by a lack of detail, and ultimately understanding, between the specific goals of the SUB, its capabilities, and its behavior associated with ensuring goal attainment in a dynamic environment (B. Cheng et al., 2009).

The use of high-level goals to drive the changeload definition process, coupled with the complex nature of self-adaptive systems and their interactions with the operating environment (B. Cheng et al., 2009), abstracted complex relationships which made their analysis difficult (Lorenzoli, Tosi, Venticinque, & Micillo, 2007). This practice may have also introduced inaccuracies into the changeload definition process (Moorsel et al., 2009) that can compound with each subsequent step. Further, the evaluator had the daunting responsibility of defining the benchmark domain (components, system, application domain) and key benchmark elements such as measures, workload, faultload, attackload (all components of the changeload), while considering the possible trade-offs between representativeness, portability, practicality, and cost of the benchmark (Almeida & Vieira, 2012a; Bondavalli et al., 2009). Analysis of their approach and issues that existed with the vague treatment of system goals are discussed below.

Abstraction is used to focus on a limited number of details at a time (Almeida & Vieira, 2011). The original study used this technique in an attempt to reduce the number of goals to consider, and ultimately, the total number of risks to be enumerated by only using high-level aspects of the SUB in Step A. However, vital details are lost when the level of abstraction is too high, especially when there is a high degree of variability, complexity, and uncertainty present within the SUB (B. Cheng & Atlee, 2007).

Almeida and Vieira (2012a) stated that goal definition and prioritization should occur prior to defining the changeload. However, it was not clear how the evaluator should deal with goals that may conflict at runtime or have complex relationships common to large self-adaptive systems (B. Cheng et al., 2009) since the number of goals, functionality, features, relationships, and interactions grew with the size of the SUB (Bondavalli et al., 2009). For instance, a web server may be configured to maximize its performance by reducing its availability, such as its maximum number of connections (Hellerstein, Diao, Parekh, & Tilbury, 2004). It's unclear how the base scenario would be defined without knowledge of the underlying conditions (B. Cheng et al., 2009) that trigger its multiple adaptive trajectories (Almeida & Vieira, 2011).

Another example is that of the Znn.com, a self-optimizing web server built on the RAINBOW framework, which optimizes its performance, cost, and content fidelity in response to its workload (S. W. Cheng, Garlan, & Schmerl, 2009). Defining a base scenario based on a simple list of these goals would be a daunting task without understanding their underlying relationship (Bondavalli et al., 2009; B. Cheng et al., 2009). It is difficult to define changes to the SUB's generic operating conditions and goals (B. Cheng et al., 2009; Tamura et al., 2012) as its relevant operations and interactions (Pressman, 2005) may not be apparent due to the abstraction of fine-grained self-adaptive capabilities (B. Cheng et al., 2009).

Thus, the lack of detail regarding the goals and their relationships caused the evaluator to consider a significantly larger change space (i.e. all combinations of goal relationships and their underlying requirements) due to the inability of filtering out those that are not applicable to the SUB (Pressman, 2005). This fact introduced additional issues when trying to define a ToS relative to the SUB, and even more so when multiple goals must be attained concurrently (e.g. minimum throughput, maximum response time, minimum latency), which is typically the case

with complex self-adaptive systems and further expands the number of changes considered (Brun et al., 2009; Weyns, Iftikhar, Iglesia, & Ahmad, 2012).

Vague treatment of Operating Conditions

The risk-based approach also presented similar issues with the treatment of operating conditions. Runtime goal achievement is dependent on the current operating conditions of the SUB (e.g. internal and external context) (B. Cheng et al., 2009), and therefore, detailed knowledge of its operating conditions is necessary to define the base scenario, evaluate goal attainment, and correlate system context to runtime behavior (Pressman, 2005; Tamura et al., 2012).

In the case of dependability benchmarking of static systems, the base scenario would be defined as an absence of faults (Kanoun, Madeira, & Arlat, 2002). That is, the SUB is operating within anticipated conditions (such as resources and workload) and services are being provided at expected levels (Bondavalli et al., 2009). In the case of resilience benchmarking of self-adaptive systems, these operating conditions are defined as those in which the SUB runs a typical workload and does not need to adapt (i.e. self-configure, self-optimize, self-heal, or self-protect) to attain and maintain runtime goals (Almeida & Vieira, 2012a).

The base scenario must include the specific conditions, such as operational context and system-level properties, under which all runtime goals are obtained without employing self-adaptive capabilities so that deviations from that state are identifiable (B. Cheng et al., 2009; Tamura et al., 2012). The SUB's determination of whether it should adapt is dependent on its goals and its changing context, so they must be well understood by the evaluator to fully characterize its response to a change (Bondavalli et al., 2009; B. Cheng et al., 2009). An

analysis of the issues resulting from the vague treatment of operating conditions is discussed below.

The approach did not clarify how an evaluator would define the operating conditions for a system whose self-adaptive capabilities include a self-optimization mechanism, such as throughput awareness. It was unclear if the operating conditions would have to guarantee a static context for all systems in the class (or for the specific system), if the operating conditions were considered before or after optimization, what degree of granularity and detail was required for the operating conditions and their relationship to the goals, how variations to the operating conditions (i.e. change scenarios) that would elicit an adaptive response were defined, and finally, how the SUB's adaptive responses affected its operating conditions (B. Cheng et al., 2009).

Further, the identification of change scenarios, Step B in the approach, considered the possible sources of change to the SUB (e.g. hardware, environment), defined and classified specific change classes (e.g. software and hardware changes, human interaction), and finally extracted specific change types from the defined classes (e.g. database table drops, software updates) (Almeida & Vieira, 2012a). The risk-based approach did not utilize constraining properties to reduce the change space and benchmark's scope (Robert Laddaga & Robertson, 2000; Pressman, 2005). Thus, the evaluator considered all possible sources of change that may affect the system-type which unnecessarily considered the entire change space consisting of any and all changes in its hardware, software, component, sub-system, interaction point, and workload the system-type may encounter (Bondavalli et al., 2009; van Lamsweerde & Letier, 1998).

Detailed knowledge of the SUB's goals was necessary to be able to define changes that deviate from the base scenario's context and employ its self-adaptive capabilities (B. Cheng et al., 2009; Tamura et al., 2012), instead of arbitrarily extending the considered change space by defining any possible changes that may have caused it to do so (Barbosa et al., 2005; Jorgensen, 2002). Similarly, there was no way of determining when a sufficient number of changes were identified (i.e. change coverage), if vital changes were ignored, or if the identified changes were even possible or pertinent given the SUB's capabilities (Moorsel et al., 2009).

The identification of relevant changes posed a significant challenge (Almeida & Vieira, 2012a), especially if insufficiently guided. A single change may have introduced unanticipated side effects and indirectly affect other runtime goals (B. Cheng et al., 2009). There was no way to systematically determine the extent of a change's effects on the SUB without knowledge of its operational context and their relationship to its goals (B. Cheng et al., 2009), leaving the evaluator little option but to define and enumerate the large number of test cases (Robert Laddaga & Robertson, 2000; Pressman, 2005). The evaluator then translated the identified changes into concrete changes (i.e. executable code) and determined the appropriate trigger instant, duration, and amount for each (Almeida & Vieira, 2012a). However it was not clear how these details were being determined, or how the changes were being selected for translation when field data was not available, leaving little option but to translate them all.

For instance, Almeida and Vieira (2012a) used the example of a "10% increase in the number of requests per second commencing 5 minutes after starting execution of the workload and ceasing 2 minutes thereafter" which may or may not have resulted in any self-adaptive capabilities being employed to maintain goals. The assumption was that it would result in activation of self-adaptive mechanisms and provide relevant feedback to the evaluator, however,

this may not have been the case, and instead it may result in additional costs (e.g. labor and effort) of analyzing and enumerating a larger number of changes than was necessary.

The shortcomings of the risk-based approach outlined in the previous section resulted in high benchmarking costs, discussed below.

Cost

The risk-based approach suffered from high cost due to the consideration of the entire change space resulting from the use of imprecise constraints (vague goals and operating conditions) throughout the procedure (Pressman, 2005). The risk-based approach's identification of changes and their correlation to the change space is illustrated in the following example.

Consider a self-adaptive HTTP Web Server, Self-System A, which possesses self-optimizing mechanisms that adjust the number of allowed connections to ensure QoS requirements of low response times. The risk-based approach considered the high-level goal of "self-optimization" and any change that may affect the SUB in any way. These changes were defined as those affecting hardware, H, software, S, or the SUB's internal context, I, as defined in Equation 7.

$$H = hardware_{changes}$$
$$S = software_{changes}$$
$$I = \text{internalContext}_{changes}$$

Equation 7: Definition of Change

Changes originating in the SUB's environment (i.e. external to the system), E, are defined as all possible hardware or software changes, as defined in Equation 8.

$$E = environment_{changes} = H \cup S$$

Equation 8: Definition of Environment Changes

Finally, the change space, CS , was defined as all possible internal and external changes that may affect the SUB, as depicted in Equation 9.

$$CS = E \cup I$$

Equation 9: Definition of the Change Space

The risk-based approach considered only the SUB's high-level capabilities, $SC_{high-level}$, that relate to the self-optimizing mechanisms, as defined in Equation 10.

$$SC_{high-level} = \{x \mid x \text{ is any capability relating to the SUB}\}$$
$$\therefore \{x \mid x \text{ is a self-optimizing capability of the SUB}\}$$

Equation 10: Definition of High-Level System Capabilities

Further, the high-level goals, $G_{high-level}$, were identified for the SUB, as shown in Equation 11.

$$G_{high-level} = \{x \mid x \text{ is any goal that genrally relates to the SUB}\}$$
$$\therefore \; \{x \mid x \text{ is any goal that relates to QoS}\}$$

Equation 11: Definition of High-Level System Goals

Finally, the considered changes, $CC_{risk-based}$, was defined by the risk-based approach as those changes that affect the SUB's high-level capabilities, $SC_{high-level}$, from attaining and maintaining its high-level goals, $G_{high-level}$, as depicted in Equation 12.

$$CC_{risk-based} = \{x \mid x \in CS,\; x \text{ affects } SC_{high-level} \text{ maintenance of } G_{high-level}\}$$
$$\therefore \{x \mid x \in CS,\; x \text{ affects the self-optimizing capabilities maintenance of the QoS goal}\}$$

Equation 12: Changes Considered by the Risk-Based Approach

As illustrated, the risk-based approach considered any possible change that may have affected Self-System A, or its goals, regardless of if the SUB could detect the change, if the

change would elicit an adaptive response, or if the goal was maintained by a self-adaptive mechanism, which made $CC_{risk-based}$ very large. The issue became very prevalent for large complex self-adaptive systems (Bondavalli et al., 2009; Salehie & Tahvildari, 2009) due to the complex interaction between the SUB, its dynamic environment, and its emergent behavior (Bondavalli et al., 2009; B. Cheng et al., 2009).

The use of an exhaustive changeload (a test everything approach) is impractical as it introduces high cost, high labor, and increased difficulty into the resilience benchmark (Salehie & Tahvildari, 2009; Vieira & Madeira, 2004), though it may exhibit a high degree of change coverage (Moorsel et al., 2009). Similarly, the risk-based approach required the evaluator to consider the risks in a general manner, organize them into categories, classes and types, and analyze each individual change to determine its relevance to the SUB based on their expected impact and probability of occurrence (Almeida & Vieira, 2012a).

If few changes were analyzed and deemed irrelevant in the selection phase, $Cs(T)$ mentioned in Equation 1, the evaluator would incur significant cost (Bondavalli et al., 2009) by having to invest time, labor, and other resources to enumerate a larger number of changes against the SUB (Vieira & Madeira, 2004). That is, $Ce(T)$, $Cu(T)$ and $Cc(T)$, would be very large. Conversely, if many changes were deemed irrelevant the evaluator would experience reduced costs associated with the enumerating changes, $Ce(T)$, $Cu(T)$ and $Cc(T)$, but incur a greater cost by manually analyzing the entire risk space, $Cs(T)$. Ultimately, finding the best possible balance between the representativeness of the changeload and the practicality of the benchmark determines the usefulness of the benchmark procedure and is an open research challenge (Almeida & Vieira, 2011, 2012b; A. B. Brown et al., 2004).

Regardless of the outcome of the change analysis, the consideration of the entire change space, or defining an exhaustive changeload, for a large and complex self-adaptive system is very costly and impractical (Kanoun et al., 2002; Salehie & Tahvildari, 2009; Vieira & Madeira, 2004). The authors attempted to reduce the change space by utilizing a cut-off level in Step E of the risk-based approach, described below.

In Step E, the changeload was defined by including only those change scenarios whose representativeness (the combination of the change scenario's impact and probability) superseded the evaluator's cut-off level (defined for the exposure matrix - e.g. High) and directly affected the size of the enumerated changes, $EC_{risk-based}$, in final changeload, depicted in Equation 13.

$$
\begin{aligned}
& EC_{risk-based} = \{x \mid x \in CC_{risk-based}, x_{impact} \geq \text{cut-off}_{impact}\} \\
& \therefore \{x \mid x \in CC_{risk-based}, x_{impact} \geq \text{high}\} \\
& = \left\{x \mid x \in CC_{risk-based}, x_{impact} \in \{\text{high, very high, mandatory}\}\right\}
\end{aligned}
$$

Equation 13: Enumerated Changes in the Risk-Based Approach

The definition of the cut-off level was subjective, based solely on the evaluator's knowledge or via multi-voting when multiple experts were involved, which made it difficult to verify and justify (Burgman, Fidler, Mcbride, Walshe, & Wintle, 2006). There was no way of knowing if the resulting changeload adequately affected the SUB with complex goal relationships (B. Cheng et al., 2009) or if it elicited an adaptive response (Almeida & Vieira, 2012b; Barbosa et al.; Friginal, de Andres, Ruiz, & Gil), save for experimentation (Robert Laddaga & Robertson, 2000), which was not cost effective (Bondavalli et al., 2009).

Furthermore, change scenarios that were under the cut-off level (and excluded from the final changeload) may have actually devastated the SUB even more than those included since they may cause subsequent changes with greater impact resulting in failure (Almeida & Vieira,

2011). The cut-off level needed to be defined in a more objective manner in which the SUB's goals, and the change's impact to those goals, were considered directly to ensure a high degree of change coverage (B. Cheng et al., 2009; Moorsel et al., 2009). Ultimately the cut-off level determined the thoroughness and change coverage of the evaluation (Moorsel et al., 2009; Pressman, 2005) and implied a degree of system robustness (Lemos et al., 2010) but it could not be verified or audited using a systematic approach (Moorsel et al., 2009).

In their follow-up paper, Almeida and Vieira (2012b) concluded that more work was necessary to address these research challenges and adequately reduce the considered change space, provide better insight, knowledge, and modeling of changes in highly dynamic systems and environments (Almeida & Vieira, 2012b). This study extended the risk-based approach to address these issues and reduce the cost of resilience benchmarking of self-adaptive systems, described in the following section.

Goal

The goal of this dissertation consisted of the extension of risk-based approach to further address the open research challenges identified in Almeida and Vieira (2012b), specifically the identification of relevant changes and the reduction in the size of the considered change space, in an effort to reduce the overall cost and labor associated with resilience benchmarking of self-adaptive systems. The study utilized system knowledge, specifically detailed descriptions of the SUB's goals and its self-adaptive capabilities, to identify and analyze only the relevant changes that result in adaptive responses of interest for resilience evaluation (Almeida & Vieira, 2012b). This approach differed from the risk-based approach, which considered the entire change space and gradually filtered out irrelevant changes. Further, this study applied both approaches to a self-adaptive system to provide a basis for comparison and demonstrate the extended changeload definition process.

Discrete mathematics has been used to describe and analyze software testing strategies (Jorgensen, 2002). Its use achieves a high degree of rigor, precision, and efficiency over informal analysis and comparative methods (Jorgensen, 2002). For instance, a set of tests, T, used to evaluate a system, S, can be represented as the test function $S(T)$ (Jorgensen, 2002). Both T and $S(T)$ can be formally defined using declarative statements, logical operations, and then manipulated using set operations (e.g. union, intersection, subset), in a similar manner utilized in the Cost section above (Jorgensen, 2002; Leung & White). The use of set theory, functions, and relationships provide a straightforward method for representing and comparing different testing strategies (Jorgensen, 2002; Leung & White, 1991). In the case of this study, comparison of the risk-based and goal-oriented approaches was straightforward and conducted

using set theory. The similarities between the two approaches allowed for direct comparison of their outputs (Galeebathullah & C.P.Indumathi, 2010; Leung & White, 1991).

The measurement of success for this study was a reduction in overall resilience benchmarking costs which was quantified using the Leung and White (1991) software testing cost model presented in Equation 1. The cost model defined the total cost of a software testing strategy, $C(Strategy)$, against a set of test cases, T, and is comprised of the costs of system analysis, Ca, test selection, Cs, test execution, Ce, result analysis and understanding, Cu, and result checking, Cc.

Cost savings were quantified using an adjusted version of the Leung and White (1991) cost model shown in Equation 1 to compare the costs of the risk-based approach, $C(risk-based)$, and the goal-oriented approach, $C(goal-oriented)$, to satisfy the cost inequality depicted in Equation 14.

$$C(goal-oriented) < C(risk-based)$$

Equation 14: Leung and White (1991) Cost Model Strategy Comparison Inequality

Confirmation of success was attained if the goal-oriented approach reduced the overall cost of resilience benchmarking by ensuring the inequality holds true, that is, it reduced the number of test cases such that any additional selection costs were offset (Leung & White, 1991; Xavier et al., 2008). Thus, the goal-oriented approach was more cost-effective if the cost savings inequality shown in Equation 15 held true. Appendix A contains a detailed description of the inequality and variable definitions.

$$\frac{s'|T_s'|+|T'|(c+e)}{s|T_s|+|T|(c+e)}<1$$

Equation 15: Simplified Test Suite Cost Comparison Inequality (rewritten)

The goal-oriented approach would succeed in reducing the overall cost of resilience benchmarking by decreasing the number of changes that required consideration throughout the process and reducing the total number of changes in the resulting changeload requiring enumeration (Xavier et al., 2008).

Relevance and Significance

This work was relevant due to the growing reliance on self-adaptive systems and the need to ensure the resilience of their services. Businesses, institutions, and governments required their systems to be resilient in dynamic environments with the capability to handle the unpredictable workloads created by our modern information society (IBM, 2003). Development and management of critical systems able to handle the explosion of information requiring storage and computation, while keeping pace with constant demands for increased performance and reduced costs, is an increasingly difficult and complex task (B. Cheng et al., 2009; Ganek & Corbi, 2003; Vieira & Madeira, 2003).

Software developers met these needs by continually exploiting growing computational power, producing more sophisticated software systems that were more versatile, flexible, robust, dependable, energy-efficient, customizable, secure, and configurable (B. Cheng et al., 2009; IBM, 2003; Madeira et al., 2002). The resulting exponential growth in the number, variety, and size of systems, sub-systems, and components created highly distributed and heterogeneous environments which were difficult to maintain and whose runtime behavior was difficult to

predict (IBM, 2003). For example, the value of the Internet has fueled significant growth in storage subsystems (e.g. Database Management Systems) which are now capable of holding petabytes of information and are only a component of an even larger system, or system of systems, requiring its own management, configuration, and tuning (IBM, 2003).

Managing large infrastructure systems became too costly and error prone and resulted in an increase in the frequency and impact of service outages (Ganek & Corbi, 2003). For instance, management and maintenance of critical infrastructure systems grew to 70 – 90 percent of total system cost and up to one-half of an organization's IT budget (Ganek & Corbi, 2003; Group, 2002). Management tasks in these large-scale production systems were too labor-intensive and stressful as they required the operators to decipher large amounts of data and make critical decisions within seconds, resulting in the prevalence of errors, failures, and outages (Ganek & Corbi, 2003). For instance, downtime due to security related service outages at brokerages houses and banking firms were estimated to cost $4,500,000 and $2,600,000 per incident per hour (Group, 2002), respectively, with about 40 percent of these outages resulting from operator error (e.g. poor configuration, tuning, or management) (Ganek & Corbi, 2003). These errors were not caused by poor training or lack of capability but by the inherent complexity of the systems and the pressures of making split-second decisions with a high degree of uncertainty (Ganek & Corbi, 2003).

Further, the economic impact was estimated at almost $3,000,000 per hour for the energy sector and $2,000,000 per hour for the telecommunications industry (Group, 2002) and did not include the societal impact (e.g. pain, suffering, and potential loss of life) experienced by those relying on these critical infrastructure services. Some of the most frequent causes of reported outages were management errors, user error and inadequate change control in systems,

performance overload and insufficient bandwidth in networks, and performance overload and configuration errors in database systems (Ganek & Corbi, 2003). Thus, proactively handling system management and maintenance issues in highly complex systems and environments was a top priority (Ganek & Corbi, 2003).

Industry, governments, and the research community have turned to self-adaptive systems to cope with the growing complexity and manageability of these systems in an effort to reduce errors, failures, and overall downtime (Bondavalli et al., 2009; B. Cheng et al., 2009; Ganek & Corbi, 2003; Group, 2002; IBM, 2003). They incorporated self-adaptive capabilities into their systems as the autonomic responses and mechanisms were better equipped to deal with the uncertainties of the system's operating conditions (Almeida & Vieira, 2012a; Moorsel et al., 2009; Salehie & Tahvildari, 2009). Automating complex management tasks reduced the need for human intervention which liberated the highly skilled technical staff from having to install, configure, operate, tune, and maintain critical systems, enabling them to focus on tasks with higher organizational value (IBM, 2003). Self-adaptive capabilities are found in web and database servers (Graefe, Idreos, Kuno, & Manegold, 2010), multimedia services (Bra et al., 2003), unmanned vehicles (B. Cheng et al., 2009), and are incorporated into large-scale legacy systems to extend their utility passed their end-of-life (Hurtado, Sen, & Casallas, 2011; Parekh, Kaiser, Gross, & Valetto, 2006; Zhang & Cheng, 2007). The increased reliance on self-adaptive systems made their resilience a top priority to those who may experience financial or social impact by their failure (Almeida & Vieira, 2012a; B. Cheng et al., 2009). Evaluation and benchmark methods are vital to instill confidence in the system's safety, quality, and overall resilience, provide methods for verifying claimed properties, reduce long-term system costs, and

reduce the frequency and impact of outages (Bondavalli et al., 2009; Garlan, 2010; Moorsel et al., 2009).

Barriers and Issues

The problem of defining a cost-effective changeload for the resilience benchmarking of self-adaptive systems was, and continues to be, inherently difficult to solve for several reasons.

First, if cost or time were not a concern it would be appropriate to define and enumerate all possible changes in all possible contexts of the system (Vieira & Madeira, 2004). The changeload would grow exponentially due to the scale and complexity of self-adaptive system's behavior, components, and interconnections (B. Cheng et al., 2009; Cin et al., 2002; Vieira & Madeira, 2004), as described in the previous sections. However, defining and enumerating all possible changes in an exhaustive changeload was impractical (Vieira & Madeira, 2004), and potentially impossible in practice (Quadri & Farooq, 2010), due to the costs associated with defining and enumerating a large number of change scenarios (Leung & White, 1991).

A second issue was defining a minimized changeload that provided maximum coverage. This has been shown to be NP-Complete and can be re-expressed as an optimization problem (Harrold, Gupta, & Soffa, 1993; Hemmati, Briand, Arcuri, & Ali, 2010). Therefore, a minimized changeload can only be approximated utilizing heuristics, greedy algorithms, genetic algorithms, and other selection techniques (Galeebathullah & C.P.Indumathi, 2010). These techniques reduce the changeload size by removing redundant, obsolete, and ineffective change scenarios (Barbosa et al., 2005; Galeebathullah & C.P.Indumathi, 2010; Harrold et al., 1993). However, they require the changeload to be defined for the entire change space and are then reduced, wasting resources on the identification of redundant and ineffective change scenarios (Barbosa et al., 2005; Roberto, 2013). The goal-oriented approach utilized system knowledge to guide the

test selection strategy in order to overcome this issue, avoid the identification and definition of irrelevant changes, and produce a minimized changeload.

Another approach, such as model based-testing (MBT), are systematic, generate change scenarios based on models, and can be proven complete (B. Cheng et al., 2009; Hemmati et al., 2010). However, MBT suffers from scalability issues when utilized against complex systems (Hemmati et al., 2010). For instance, thousands of change scenarios can be generated for even modest systems utilizing well-known coverage criteria, such as all transition-pairs or all-roundtrip paths (Hemmati et al., 2010), which is not cost-effective.

A third issue was maximizing the error detection rate during system evaluation while using a minimum number of test cases. Additionally, the changeload's cost-effectiveness must be maximized while ensuring it fully characterizes the system and evaluates goal attainment (Almeida & Vieira, 2011; Hemmati et al., 2010; Quadri & Farooq, 2010; Roberto, 2013; Vieira & Madeira, 2004). Unjustified or unguided test case omission reduced the changeload's error detection rate and can omit tests that are vital to the end-user (Hemmati et al., 2010). Conversely, not removing all ineffective tests resulted in increased cost, which hindered evaluation efforts (Barbosa et al., 2005; Quadri & Farooq, 2010; Vieira & Madeira, 2004). Defining a changeload that balanced coverage, user expectations, real-world conditions, and cost continues to be difficult and labor intensive (Quadri & Farooq, 2010). The goal-oriented approach utilized system knowledge to identify the self-adaptive elements of interest and then defined relevant changes for their direct evaluation in order to ensure test coverage of the system's resilience mechanisms.

A solution that addressed the above concerns would add to the body of knowledge and potentially provide a basis for future research.

Assumptions, Limitations, and Delimitations

The goal of this dissertation was to reduce the overall cost of resilience benchmarking of self-adaptive systems by reducing the considered change space when defining a resilience changeload. The approach utilized system knowledge to limit the identification and definition of changes to those that directly affected a system feature or service protected by a self-adaptive mechanism.

An assumption of this study was that the self-adaptive mechanisms that introduce the systematic or localized change would not introduce additional changes, such as a fault or failure, which would then prompt a series or loop of self-adaptive responses. Furthermore, self-adaptive responses and state transitions occurred within known operational states. These assumptions ensured that all adaptation and system states were fixed and did not involve emergent behavior, allowing behavioral verification and validation. Another assumption was that the defined changes accurately reflected actual changes experienced by the SUB within its production environment and its intended use. These assumptions were in-line with previous studies where the runtime behavior of complex systems was evaluated in the presence faults, failure, and other runtime changes (Almeida & Vieira, 2012a; Bondavalli et al., 2009; Cámara, Lemos, Vieira, Almeida, & Ventura, 2013; Graefe et al., 2010; Khalil, Elmaghraby, & Kumar, 2008; Vieira & Madeira, 2004).

A limitation of the study was the behavior, structure, and functionality of the target system, particularly its self-adaptive mechanisms and capabilities. The analysis, conclusions, and identified changes were only accurate and relevant for the particular implementation, which may limit the applicability of the results. However, the process and approach was generalized and not system-specific. Additionally, some adaptive trajectories or emergent behavior may not

be obvious without in-depth analysis of either documentation or source code, and may not be identifiable without experimentation. For example, a multistep adaptive response (an adaptation triggers another) to a change may be by design, where the system continuously over- and under-compensates to environmental changes until it reaches equilibrium. However, these adaptations were omitted, unless explicitly documented, since the focus of this study is to reduce the cost of resilience benchmarking while ensuring coverage of known adaptive functionality.

A delimitation of this study was that all the self-adaptive capabilities and mechanisms of the target system were fixed and known a priori. This delimitation limited the applicability of the study's results to those systems without evolving capabilities, updatable adaptive mechanisms, or emergent behaviors. Due to the degree of diversity within self-adaptive systems, other studies have also limited their focus to specific system-types or functional-families to increase the feasibility of defining relevant resilience changeloads (Almeida & Vieira, 2012a; Vieira & Madeira, 2004). This study took a similar approach by making the above stated assumptions and delimitations which were reasonable and in line with the previous study.

Definition of Terms

Operating Environment	The environment in which the system operates that cannot be directly managed by the system, such as available system memory, workload, or network connection (Madeira et al., 2002).
Self-Adaptive	A computing environment or software system with the ability to manage aspects of its operation and dynamically adapt to change in accordance to business policies, objectives, and run-time goal attainment. They can be either self-configuring, self-healing, self-optimizing, or self-protecting (Ganek & Corbi, 2003)
Change	Any significant event in the context of a system or environmental resource, internal system state, interface, or component that may affect the system's ability to attain runtime goals. These can

	include attacks, failures, faults, updates, or workload variations (Ganek & Corbi, 2003).
Managed Resource	A system component, module, or resource that can be managed by the system at runtime (Ganek & Corbi, 2003).
Sensor	An interface that provides information about the state and operation of a managed resource (Ganek & Corbi, 2003).
Effector	An interface that allows the system to modify the operational state of a managed resource (Ganek & Corbi, 2003).
Fault	Exceptional conditions that may occur internally, such as hardware or software faults, or externally, such as those that occur within the operating environment, which disrupts expected system operation (Gil et al., 2002; Madeira et al., 2002).
Failure	Is a state in which an error reaches a service interface and alters the offered service in such a way that expected service qualities are no longer met (Gil et al., 2002).
Change Trajectory	The context / operational state of the system as it adapts to a sequence or group of changes. Temporal order of changes often determine specific change trajectories (Almeida & Vieira, 2011).
Functional Testing	Testing in which the only information utilized is the software specification in which inputs are mapped to expected outputs, commonly referred to as black box testing (Jorgensen, 2002).
Black-box Testing	Testing in which the implementation of a system is not known and considered as a black box, where the function of the black box is understood completely in terms of its inputs and outputs (Jorgensen, 2002).
Test	An act of exercising a software system in an effort to find failures or to demonstrate its correct operation (Jorgensen, 2002).
Test Case	A set of inputs and expected outputs used to test program behavior (Jorgensen, 2002).
Fault Space	The set of all possible faults that may affect a system, its components, or its environment (Vieira & Madeira, 2004).
Change Space	The set of all possible changes that may affect a system, its components, or its environment (Almeida & Vieira, 2012a).

Resilience	Encompasses all attributes of quality where a system works well and can be trusted in a changing environment and in the presence of faults, failures, errors, and attacks (Almeida et al., 2010).

Summary

Trends and projections depicted an increase in the need for performance, resilience, and reduced costs of infrastructure systems to meet the growing demand of modern society. However, the increased complexity of these systems in response to growing demand negatively contributed to the management and maintenance of these systems, as they were more prone to outages and errors, which resulted in loss of revenue or disruption in service.

Self-adaptive capabilities endowed a system with autonomic features of self-management or self-healing, which reduced the reliance on human-operators to conduct routine maintenance tasks or troubleshoot issues. Benchmarking and validation of resilience was of utmost importance due the reliance on the critical infrastructure services maintained by self-adaptive mechanisms. However, testing was often labor intensive and cost-prohibitive due to the scale and complexity of these systems. This resulted in insufficient or incomplete testing of runtime functionality, or in many cases, testing was omitted as a cost-saving strategy. Therefore, since software testing can account for 50 to 80% of total system costs, a method for reducing the cost of resilience benchmarking of self-adaptive systems while maintaining test coverage was required.

Barriers existed in achieving this goal. Maximizing the cost-effectiveness of the test suite, while simultaneously maintaining test coverage, was difficult. Additionally, the determination of which tests could be omitted to reduce costs continues to be an open research question. Special care must be observed in maintaining this balance as a solution that does not

sufficiently reduce the cost of benchmarking, or negatively impacted the test coverage of the suite, was unacceptable.

The risk-based approach presented in this chapter is representative of the current research that has attempted to address these problems. It consisted of utilizing Software Risk Evaluation (SRE) techniques to identify the risks threatening the achievement of the system's goals. As such, this research proposed an extension to the risk-based approach to utilize goal-oriented requirements engineering techniques to extract system knowledge and determine if cost-savings and greater effectiveness can be realized over previous research.

The next chapter provides a review of the literature providing an overview of performance benchmarking, dependability benchmarking, and resilience benchmarking as it relates to self-adaptive systems, followed by a discussion of the benchmarking cost saving techniques found within the literature.

Chapter 2

Review of the Literature

Introduction

A comprehensive review of the risk-based approach for defining resilience changeloads has been conducted and its corresponding shortcomings were discussed in the Problem Statement. The discussion has demonstrated the need to extend the risk-based approach to reduce the cost of resilience benchmarking while ensuring the selection of relevant changes that exercise the pertinent system functionality. This section discusses the concepts that were pertinent to this study, such as performance benchmarking, dependability benchmarking, and resilience benchmarking, and then culminating with a discussion of existing cost-savings techniques for system benchmarking.

Benchmarking

Benchmarks are a generic way of characterizing a system's runtime behavior, called the system under benchmark (SUB), by simulating real-world operating conditions (such as expected workloads) and analyzing the quantitative output produced using metrics, which provided a standardized method of evaluating and comparing alternative implementations (Almeida & Vieira, 2012a; Bondavalli et al., 2009; A. B. Brown et al., 2004; Kaddoum, Raibulet, Georg, Picard, & Gleizes, 2010). Their results were used to gauge a system's effectiveness in its intended operating environment, set realistic expectations for its Quality of Service (QoS), provided assurance and verification of key property claims, and abstracted a system's technical details to allow non-technical end-users to compare alternative systems in a straightforward

manner (Almeida & Vieira, 2011; A. B. Brown et al., 2004; Kaddoum et al., 2010; Weicker, 1990).

Work on benchmarking focused primarily on performance aspects of systems, such as CPU, operating system, and file system performance (Almeida & Vieira, 2012a; Traeger, Zadok, Joukov, & Wright, 2008). Performance benchmarks were composed of three major components, the workload, which was the computational load for the SUB (Cin et al., 2002), performance metrics, and execution rules (Almeida & Vieira, 2012a). They were classified as real, ad-hoc, synthetic, application, or trace benchmarks (Agrawal, Arpaci-Dusseau, & Arpaci-Dusseau, 2008).

A real application benchmark was the use of the application that the end-user intended to run on the system as a benchmark for the system, with the obvious advantage that the benchmark results corresponded directly to the actual scenario the end-user cared about, and was the most representative of its real-world performance (Agrawal et al., 2008). However, this technique was impractical as was impossible to determine the specific use of a system for each potential end-user, especially in the case of general-purpose and commercial off-the-shelf (COTS) systems (Traeger et al., 2008).

Ad-hoc benchmarks were created by a system's author for in-house use, were not available to outside parties, and were not reproducible. The code for in-house benchmarks were not widely used or distributed, which resulted in differing implementations, increased errors, and made their results difficult to compare (Traeger et al., 2008).

Synthetic benchmarks were solely written to simulate real-world workloads and performed no useful computations, such as the TPC-C benchmark, by the Transaction Processing Performance Council (TPC) which was used for online transaction processing (OLTP)

benchmarks for database management systems (DBMS) (Weicker, 1990). TPC-C mimicked the activity of a wholesale supplier where multiple users executed data-intensive transactions against a database (Council, 2010). Synthetic benchmarks were widely available, standardized, and were highly reproducible, but their workloads did not always represent real-world conditions accurately (Agrawal et al., 2008; Traeger et al., 2008; Weicker, 1990).

Application benchmarks were distilled from real and purposeful programs that were representative of those used in a particular industry or within a system-type, such as LINPAC, which was a package of libraries used in sophisticated Fortran programs and was originally a major component of a scientific application (Fernandez & Garcia, 1999; Weicker, 1990). They were also widely available and representative, but their results were highly dependent on the language and libraries used in their implementation, which made them prone to gaming (Weicker, 1990).

Finally, trace benchmarks recreated real workloads by logging operations and replaying them under controlled conditions, and if done correctly, they were the most representative benchmark type (Traeger et al., 2008). However, the lack of standardized methods for capturing and replaying traces, coupled with variations in benchmark system setups, made their results difficult to compare and interpret due to the complex interactions of their components (Agrawal et al., 2008). Further, real-world traces were not readily available due to privacy concerns of both the creator (e.g. proprietary technologies) and their users (e.g. capturing of personal information) (Traeger et al., 2008).

Measurements were taken of the SUB while it computed the workload, such as those mentioned above, using performance metrics. A performance metric is a standard method of measuring and quantifying a property of interest, such as bytes per second (bps), millions of

instructions per second (MIPS), millions of floating point operations per second (Mflops), or transactions per minute (tpmC), and allowed the direct comparison of systems (Agrawal et al., 2008; Council, 2010).

There were several challenges with performance benchmarking, such as finding the balance between the representativeness and practicality of the benchmark (A. B. Brown et al., 2004). For instance, a benchmark with a high degree of representativeness (i.e. it represents a production environment and system configuration very well) often resulted in complex and costly benchmarking setups and procedures, which reduced its reproducibility and portability over different systems (Fernandez & Garcia, 1999; Moorsel et al., 2009).

Another challenge was determining the appropriate workload to adequately characterize a system so that the properties of interest were isolated in a realistic manner (Fernandez & Garcia, 1999). For example, CPU benchmark results were often influenced by a number of factors other than the CPU, such as the programming language characteristics of the benchmark, compiler optimizations used, runtime libraries utilized within the benchmark code, and the cache sizes of the involved components (e.g. CPU and disk caches) (Weicker, 1990). Thus, benchmark results must be considered with the context of tasks performed and measurement assumptions to ensure proper interpretation and comparison (Weicker, 1990).

Benchmarks are useful tools that provide means of comparing systems on various performance properties, identify performance problems and bottlenecks, and motivate system design improvements (Fernandez & Garcia, 1999). Useful benchmarks are those that are representative of the system domain, produce expressive results that adequately describe the SUB, are repeatable, portable over different systems, and verifiable (Almeida & Vieira, 2011; A. B. Brown et al., 2004; Fernandez & Garcia, 1999). Despite the large amount of research

focusing on performance benchmarks, researchers continued to address the challenges of defining representative workloads due to the growth in complexity of both modern systems and their usage characteristics (Almeida et al., 2010; IBM, 2003).

Dependability Benchmarking

Society's use of networked devices for critical infrastructure services increased awareness of the importance of failures that resulted in undesirable repercussions, such as loss of revenue, prestige of a company, trust in a service, and even loss of life (Ganek & Corbi, 2003; Madeira & Koopman, 2001). Performance and functionality were no longer the only motivation for improvements in technology products as the technology industry was increasing its emphasis on designing systems that could function in the presence of faults and failures, that is, systems that were dependable (Kanoun et al., 2002; Madeira & Koopman, 2001).

Dependability is an integrating concept that combines the attributes of availability, reliability, safety, integrity, and maintainability of systems that is attained by incorporating fault prevention, fault tolerance, fault removal, and fault forecasting capabilities into a system (A. Avizienis, J. C. Laprie, B. Randell, & C. Landwehr, 2004). Thus, a dependable system is one with ability to delivery services, via fault prevention and tolerance mechanisms, that could be justifiability trusted by avoiding service disruptions due to frequent and severe faults, using fault removal and forecasting features (A. Avizienis et al., 2004; Kanoun et al., 2004). Faults are defined as exceptional, abnormal, or stressful conditions that result in system failure, or more precisely, a state in which a system no longer accomplishes its intended purpose or goals (Vieira & Madeira, 2003).

Thus, the goal of dependability benchmarking was to provide a systematic means of characterizing the behavior of computer systems in the presence of faults, typically evaluated

from the end-user's perspective of their expected services, in a reproducible and cost-effective manner (Cin et al., 2002; Kanoun et al., 2002; Kanoun et al., 2004). Dependability benchmarks extended performance benchmarks by subjecting the SUB to representative faults while it executed workloads typically utilized in performance benchmarks (Almeida & Vieira, 2011; A. B. Brown et al., 2004; Cin et al., 2002; Kanoun et al., 2004). For example, the well-known dependability benchmark DBench used TPC-C as its workload.

The injection of the faults was a critical experimental technique for assessing and verifying dependability (Moorsel et al., 2009; Xavier et al., 2008) as it provided insight into the SUB's tolerance and recovery capabilities in the presence of simulated faults (Kanoun et al., 2002; Salehie & Tahvildari, 2009; Vieira & Madeira, 2003). Faults included internal and external faults affecting software, hardware, network, and human components (Cin et al., 2002).

Faultloads

The faultload captured the additional dimension of fault injection in dependability benchmarking. It was a set of representative faults to be injected into the SUB and included their intended location (e.g. in code, memory, or in hardware), insertion time (e.g. when they should be injected), relative distribution within time and space, and fault type (Cin et al., 2002; Kanoun et al., 2004). Some examples of faults include register bit-flips to simulate CPU hardware faults, data corruption to simulate software faults, read / write timeouts to simulate disk faults, and packet loss to simulate network interface faults (Cin et al., 2002). The SUB's reaction to the faultload was measured utilizing dependability metrics, such as mean time to failure (MTTF) and total uptime, which allowed the direct comparison of systems using quantitative results (A. B. Brown et al., 2004; Madeira et al., 2002).

The faultload was critical to dependability benchmarking but was non-trivial to define. The following section discusses several challenges associated with faultload definitions.

Faultload Challenges

Defining a representative faultload was the most difficult and obscure aspect of dependability benchmarking (Kanoun et al., 2002; Madeira et al., 2002; Madeira & Koopman, 2001) and was more complex than defining workloads for performance benchmarks (Kanoun et al., 2002). In particular, determining the essential elements of the evaluation domain, identifying the features of interest, and defining the most applicable faults of the faultload in a practical and reproducible manner were difficult and labor intensive tasks (Kanoun et al., 2002). This was a result of a lack of available field data (Almeida & Vieira, 2011; Moorsel et al., 2009) and the complex nature of computer faults (Vieira & Madeira, 2004).

Further, the faultload had to portray a high degree of representativeness, completeness, implementability, portability, and repeatability, while being comprised of the minimal number of faults to ensure its cost-effectiveness (Cin et al., 2002). Of particular importance were its representativeness, which directly related to the accuracy of the benchmark results (Cin et al., 2002), portability, which ensured its ability to directly compare different systems (Moorsel et al., 2009; Vieira & Madeira, 2004), and cost-effectiveness, which determined its practicality and reproducibility (Kanoun et al., 2004).

A system's fault space was comprised of all possible sources of faults, affecting any component or interface of the system, that may or may not result in failure (Vieira & Madeira, 2004). The fault space could be very large as it grew exponentially in relation to the number of system components, features, and interfaces (Bondavalli et al., 2009). An exhaustive faultload, which contained all possible faults in the fault space, was often recommended in literature (Cin

et al., 2002; Kanoun et al., 2004) because it ensured a high degree of fault coverage and a greater possibility of uncovering unknown flaws and defects (Vieira & Madeira, 2004). However, this practice became increasingly impractical as the complexity and size of the SUB increased, especially with respect to cost (Cin et al., 2002; Xavier et al., 2008). Cost referred to the overall cost of dependability benchmarking, which included: the time and effort involved in considering and defining the fault space, the analysis and selection of faults to include in the faultload, and the time and resources required to enumerate the faultload in the experimental phase of the benchmark (Ganek & Corbi, 2003; Kanoun et al., 2004). Thus, the cost of a dependability benchmark was directly related to the number of faults considered, included, and enumerated (Xavier et al., 2008).

Several techniques were proposed to reduce the considered fault space and the cost of dependability benchmarking. For example, many classes of low-level hardware faults exhibited similar high-level characteristics, so simulating hardware faults at higher logical layers reduced the number of hardware faults in the faultload (Cin et al., 2002). Similarly, software faults could also be abstracted using established software defect classifications, such as the Orthogonal Defect Classification (ODC), which classified software defects in a set of non-overlapping classes (Cin et al., 2002). Thus, fewer faults needed to be considered and enumerated as the results of a single fault was representative of the entire fault class (Xavier et al., 2008).

The considered fault space could also be filtered (i.e. reduced) using knowledge of the SUB's dependability features, services, and the visibility of a fault's resulting failure (Barbosa et al., 2005; Cin et al., 2002; Friginal et al., 2011). For instance, the fault space for simulating hardware faults (i.e. memory and CPU register bit-flips) could be optimized by eliminating faults with low representativeness (Barbosa et al., 2005). These faults were defined as faults that were

repetitive, such as those that occurred in the same location but at different times, and faults that lacked relevance, such as those that never resulted in a failure (Barbosa et al., 2005; Friginal et al., 2011). Examples of the latter were bit-flips that were injected into a register before a write operation occurred and were subsequently overwritten, and bit-flips that were injected but were never read for useful computations (i.e. activated) (Barbosa et al., 2005).

Evaluators also used system knowledge, its context of use, and properties of the SUB's environment to discern relevant faults from the fault space (Friginal et al., 2011). With this knowledge, the evaluator could determine which faults would actually impact the SUB and the elements of interest, such as those that exercised its dependability mechanisms (Barbosa et al., 2005; Friginal et al., 2011). Selecting faults based on environmental properties significantly reduced the number of considered faults due to its inherent complexities and direct effect on the SUB (B. Cheng & Atlee, 2007; Kanoun et al., 2002; Pressman, 2005). For instance, ambient noise and signal attenuation greatly impacted the availability and integrity of data transfers over wireless networks (Friginal et al., 2011) but had little to no relevance for stationary infrastructure systems. Another example was the risk of physical damage or attack (e.g. hitting or dropping the system) which was very relevant for mobile systems but not for database systems.

The use of system knowledge significantly reduced the considered fault space, increased the relevance of the faults incorporated into the faultload, and reduced the cost of dependability benchmarking (Barbosa et al., 2005). Reducing the considered fault space and overall cost of dependability benchmarking was critical as exhaustive faultload were expensive, labor intensive, and wasted resources by evaluating the SUB against irrelevant faults (Barbosa et al., 2005; Madeira & Koopman, 2001)

Dependability benchmarking focused on measuring and comparing the dependability and performance of systems, with the goal of verifying system behavior and dependability features in the presence of faults (Kanoun et al., 2002; Kanoun et al., 2004). Researchers continue to address the challenges within the n-dimensional problem space of dependability benchmarking that were caused by the huge complexities found within the application domain, operating environment, the very nature of faults, and interaction of all these elements (Kanoun et al., 2004; Madeira & Koopman, 2001). Defining a good workload, and even more so for a good faultload, was a pragmatic process that required observation and analysis of the SUB's functionality, structure, and the constraints and assumptions imposed upon it by its environment (Cin et al., 2002).

Self-Adaptive Systems

Modern systems have increased in complexity and have become unmanageable due to the adoption of heterogeneous, dynamic, and interconnected systems of systems that addressed the growing needs of society (Almeida & Vieira, 2012a; Ganek & Corbi, 2003). As a consequence, industry and the research community focused on developing systems that were capable of performing standard maintenance, optimization tasks, and recovery operations in response to changes within themselves and their operating environment with little or no human intervention, called self-adaptive systems (Almeida & Vieira, 2011; B. Cheng et al., 2009; IBM, 2003). They were organized into four main categories: self-configuring, self-optimizing, self-healing, and self-protecting systems (Almeida & Vieira, 2011; B. Cheng et al., 2009; IBM, 2003).

The autonomic operation of self-adaptive systems allowed them to quickly adapt to highly variable workloads, respond to unpredictable operating conditions, and make performance enhancing changes while reducing system maintenance costs, failures due to operator error, and

overall system downtime (Ganek & Corbi, 2003; IBM, 2003; Kaddoum et al., 2010). They were not bound by predefined execution paths, or the static logic typical of traditional systems, which endowed them with dynamic runtime behavior (Ganek & Corbi, 2003). They gathered and utilized contextual information of their operation and environment to optimize their responses to change and were typically implemented with a closed-loop mechanism (i.e. adaptation loop) called the MAPE-K (Monitoring, Analyzing, Planning, Execution, and Knowledge) loop (Almeida & Vieira, 2012a; Ganek & Corbi, 2003; IBM, 2003; Moorsel et al., 2009). The MAPE-K loop consisted of system capabilities responsible for monitoring its context (internal and external to the system), analyzing changes to its context, planning adaptive responses to those changes using its newly gathered data and its previous knowledge, executing its adaptation plans, and finally updating its knowledgebase with its newly acquired information (IBM, 2003; Robert Laddaga & Robertson, 2000). These systems were expected to be resilient in achieving and maintaining their predefined goals by adapting (proactively and reactively) their behavior and structure in response to runtime changes (Almeida & Vieira, 2012a).

The property of resilience merged concepts of performance, dependability, and security (Almeida et al., 2010). It pertained to a system's persistence of trusted service delivery when faced with circumstances that were beyond its normal (i.e. ideal) operating conditions (Almeida & Vieira, 2011; Laprie, 2008) which inhibited its ability to satisfy runtime requirements and goals (Almeida & Vieira, 2012a; Bondavalli et al., 2009; Ganek & Corbi, 2003; IBM, 2003). Society's reliance on self-adaptive systems for large-scale, mission critical, and infrastructure systems (Bondavalli et al., 2009) increased the urgency of finding methods for the assessing their resilience and other runtime attributes (Almeida & Vieira, 2012b).

Resilience Benchmarking

The need to evaluate a system's ability to maintain expected service levels in the presence of changes other than faults became critical due to the increased reliance of highly complex infrastructure systems designed with self-adaptive capabilities (Almeida & Vieira, 2011; A. B. Brown et al., 2004; IBM, 2003). Benchmarking, which provided methods for evaluating such characteristics, had focused primarily on evaluating the performance and dependability of static systems whose runtime behavior was predictable and constrained to fixed execution paths (Almeida & Vieira, 2012a; Bondavalli et al., 2009; IBM, 2003). However, traditional benchmarking methodologies could not be applied to self-adaptive systems "as-is" because they did not provide insight into their complex runtime behavior and potential variations in system response (Bondavalli et al., 2009).

Further, traditional dependability benchmarks focused on identifying conditions that caused the SUB to enter a failure state (such as an invalid input), while resilience benchmarks focused on the transient behavior of the SUB in response to a change (such as a step variation in workload) and its final operational state (e.g. transient, stable, or a failure state) (Hellerstein et al., 2004). Therefore, dependability benchmarks were extended to include other facets of change experienced by self-adaptive systems, such as internal and environmental variances, to fully assess their capabilities (Almeida & Vieira, 2011; Bondavalli et al., 2009; Salehie & Tahvildari, 2009).

Resilience benchmarking extended dependability benchmarking by providing methods to evaluate and compare the dynamic runtime behavior of self-adaptive systems when faced with changes, which were typically overlooked by traditional dependability benchmarks (Almeida &

Vieira, 2011; Bondavalli et al., 2009). As in dependability benchmarking, resilience was evaluated as the SUB executed a representative workload, such as those used in performance benchmarking (Kanoun et al., 2004; Moorsel et al., 2009). Measurements were taken of specific system attributes, such as behavior and performance characteristics, utilizing specialized resilience metrics (Almeida & Vieira, 2011; Huebscher & McCann, 2004; Kaddoum et al., 2010; Robert Laddaga & Robertson, 2000). Resilience metrics included CPU performance (CPUP), Working vs. Adaptivity Time (WAT), and adaptation latency (Kaddoum et al., 2010). Thus, a resilient system had to be able to adapt to changes in service demands (i.e. workloads), faults and attacks (i.e. faultloads), and other types of perturbations that imposed changes onto the SUB, but may not have necessarily resulted in failure (Almeida et al., 2010).

Just as adaptive capabilities endowed a system with an additional dimension of runtime dynamism, the additional dimension of change was captured to assess a system's effectiveness while coping with change, called the changeload, described in the Chapter 1 (Almeida et al., 2010 2011; Almeida & Vieira, 2012b; A. B. Brown et al., 2004).

Cost Saving Techniques

Testing is the most critical and expensive phase of the Software Development Lifecycle (SDLC). Software maintenance costs, of which testing is a component, can range from 50 to 80% of total software cost over the life of the system (Leung & White, 1991) and can even exceed this range when the system if repeatedly modified and tested (Harrold et al., 1993; Leung & White, 1991). This phase was critical for self-adaptive systems as their complexity and scale required repeated testing to validate their complex runtime characteristics (B. Cheng et al., 2009). This section discusses techniques found in literature aimed at reducing the cost of software testing.

In Barbosa et al. (2005) the authors proposed a fully automated technique of reducing the cost of fault injection that reduced the considered fault-space using assembly-level knowledge of the target system. The technique mapped each register and memory location within the compiled code and determined those injection points that would not result in a system disturbance, that is, the ineffective faults. Only those locations that had a corresponding READ operation immediately after the fault injection point were considered. This was coupled with fault classes being defined and the testing of a single class member in the optimized fault-space to further increase the technique's cost-savings by removing redundant and overlapping test cases.

The authors utilized a Motorola MPC565 microcontroller to facilitate the injection of the bit-flip faults during the execution of two workloads – a quicksort algorithm and a jet engine controller – that demonstrated the technique's feasibility and effectiveness within general computing and mission critical applications. The quicksort application executed within two minutes, its fault-space optimization required only twenty seconds to complete, and each of its fault injection experiments required less than thirty seconds. During the experiment's "golden run," the processor executed 34 distinct assembly opcodes and 815 total instructions. The jet engine controller workload required twelve hours for its golden run, ten minutes for its fault-space optimization, and fault-injection experimentation required less than two minutes per experiment. Its golden run executed an average of 88 unique opcodes and 231 instructions.

The experiments identified three primary outcomes: detected errors, which were those that were signaled by the hardware error detection mechanisms of the processor; wrong outputs, which were errors that were not detected by the processor and resulted in incorrect application output; and non-effective errors, which were errors that did not affect the system's execution during the experiment.

The results of the experiments showed an increase in injected fault effectiveness, which increased from 5% to 47.7% in the optimized fault-space using the quicksort workload and from 4.4% to 38.2% using the jet controller workload. Table 1 summarizes the study's fault-space optimization results.

Workload	Campaign Type	Size of Fault-Space (registers)	Size of Fault-Space (memory)
Jet Engine Controller	Non-optimized	5.0×10^8	1.9×10^{11}
	Optimized	7.7×10^6	3.3×10^6

Table 1: Fault-Space Optimization Results

The technique resulted in a fault-space ratio of only 1.5% and 0.0017% of the original register and memory fault spaces, respectively. These results related to the jet engine controller running on the 32-bit processor utilizing 100 KB of memory during its execution. The optimization technique successfully reduced the fault-space by two orders of magnitude for the registers and five orders of magnitude for memory. The fault-space optimization reduced the total memory fault-space by 99.9983% and the register fault-space by 98.5% while the effectiveness of the considered faults increased by 33.8%.

The optimized fault-space allowed for the consideration and selection of fewer faults but did not reduce the error coverage of the faultload. For example, the optimized faultload included only 1559 faults, a reduction of 72.69%, but increased the fault effectiveness from 2.0% to 19.1%. The reduction of the faultload equated to substantial cost-savings over the non-optimized fault-space since cost is directly tied to the considered fault-space, size of the faultload, and number of executed experiments (Leung & White, 1991).

The authors concluded that further optimization was possible by analyzing error propagation as they observed that faults in some registers had a greater tendency to generate wrong outputs that caused detected errors in other registers. This type of post-injection analysis,

coupled with the techniques pre-injection analysis, could further reduce the fault-space and increase the selected faultload's effectiveness. Finally, specific components could be targeted to evaluate specific error detection or recovery mechanisms directly, speeding the evaluation and further reducing the faultload's size. The study showed that investments in the analysis and selection phases of test suite definition process using pre-injection techniques provided significant cost savings by reducing the considered fault-space, optimizing the test suite, and reducing the total number of tested faults.

In Xavier et al. (2008) the authors proposed a technique that reduced the number of test cases for a program by discarding redundant and repetitive tests from the test suite. This was accomplished by combining automated model checking and program verification that ensured the testing criteria (coverage requirements) were met. The technique first defined testing criteria to guide the test case definition process. The study focused on testing exception handling capabilities of a program, specifically, the detection of an error, the activation of an exception, and finally, the handling of the exception via fault recovery mechanisms. They also defined du-pairs between associated exception objects and their utilization, in addition to exception event activations and deactivations (i.e. exception *throw* and *catch* logic). Thus, the test coverage criteria included all throw commands, all catch commands, all exception definitions, all definition-use pairs, all exception activations, and all exception activation and deactivation (i.e. catches) pairs.

Since the testing criteria related to code coverage, specifically of structural testing, the test cases focused on executing each program command associated with exception handling. The authors constructed an automated tool, called OCongraX, to extract the points and objects of interest. It was guided by the previously defined testing criteria and then generated the

respective test cases. Once the test suite was defined, the authors utilized Java PathFinder to define bad practice properties and check them against the program model, where some bad practices included non-specific exception catches, empty catch statements, and non-specific exception throwing. By combining the tools, they avoided unexpected halts of testing activities that needed manual recovery from unforeseen errors due to bad practices, they replaced the poorly implemented exception handling statements to allow testing to focus on system validation, and they avoided executing redundant test cases that would reevaluate tested code and already satisfied testing criteria. Java PathFinder ensured that system properties were preserved while OCongraX tested the program's fault-tolerance capabilities.

The authors demonstrated their technique and tool in an experiment where the deadlock freedom of a concurrent program was tested. The technique reduced the test-space by 25% and ensured 100% test criteria coverage. The study showed that the combination of pre- and post-injection analysis techniques successfully reduced the programs test-space. Additionally, their tool automated the test case definition process for exception handling mechanisms, which reduced the labor costs of manual transcription. However, the manual analysis required to define the coverage and testing criteria utilized by the tool may add additional costs to the technique, which could potentially negate the cost-savings from the test-space reduction, especially for large-scale self-adaptive systems (B. Cheng et al., 2009; R. Laddaga, 2006). The model-checking step was conducted using the Java PathFinder automated tool, which analyzed the Java byte-code of the test program. However, the tool suffered from known scalability issues which occurred when the test program's size and complexity increased (Visser, Pasareanu, & Khurshid, 2004), which posed significant issues for large-scale self-adaptive systems (B. Cheng et al.,

2009). Finally, not all systems could be modeled, at all or easily, since their complexities may negate any potential cost savings (Andersson et al., 2009; R. Laddaga, 2006).

In Harrold et al. (1993) the authors proposed a technique to reduce the number of test cases within a test suite by removing redundant and obsolete test cases while maintaining test coverage. Their technique could be utilized in several phases of the SDLC, including initial program development, structural changes, and when both structural and functional changes were made to the system. Their technique utilized a heuristic to reduce the number of total test cases by only including those test sets with the greatest cardinality over the tested requirements, described below.

Their algorithm first included all test sets, T_i, in the test suite, TS, associated with at least one valid requirement, r_i, and with a cardinality of one (i.e. containing a single test case t_i). It then marked all test sets within TS containing any of the t_i's within the selected T_i's. Then it processed the higher order cardinalities within TS (e.g. 2, 3, and so on) and selected the T_i's that had not been marked, repeatedly until the maximum cardinality, MAX_CARD, had been evaluated, thus marking all T_i's containing duplicate test cases within TS. Finally, the algorithm returned a representative set, RS, of test sets that satisfactorily covered all valid requirements. In this manner, the algorithm marked and excluded both redundant and obsolete test cases and included only the highest order cardinal test sets that pertained to the requirements and coverage criterion, defined as each definition-usage pair (i.e. du-pair) found within the program code. The algorithm is shown below in Figure 2.

```
algorithm ReduceTestSuite

input       T1, T2, ···, Tn: associated testing sets for r1, r2, ···, rn respectively, containing test cases from t1, t2, ···, tnt
output      RS: a representative set of T1, T2, ···, Tn
declare     MAX_CARD, CUR_CARD: 1 ·· nt
            LIST: list of ti's
            NEXT_TEST: one of t1, t2, ···, tnt
            MARKED: array[1..n] of boolean, initially false
            MAY_REDUCE: boolean
            Max(): returns the maximum of a set of numbers
            Card(): returns the cardinality of a set
begin
/* Step 1: initialization */
      MAX_CARD = Max(Card(Ti))                             /* get the maximum cardinality of the Ti's */
      RS = ∪ Ti, Card(Ti) = 1                              /* take union of all single element Ti's */
      foreach Ti such that Ti ∩ RS ≠ Φ do MARKED[i] := true  /* mark all Ti containing elements in RS */
      CUR_CARD := 1                                        /* consider single element sets first */
/* Step 2: compute RS according to the heuristic for sets of higher cardinality */
      loop
         CUR_CARD := CUR_CARD + 1                          /* consider the sets with next hier cardinality */
         while there are Ti such that (Card(Ti) = CUR_CARD and not Marked[i] do
                                                           /* process all unmarked sets of current cardinality */
            LIST := all tj ∈ Ti where Card(Ti) = CUR_CARD and not Marked[i]
                                                           /* all tj in Ti of size CUR_CARD */
            NEXT_TEST := SelectTest(CUR_CARD,LIST)         /* get another tj to include in RS */
            RS := RS ∪ {NEXT_TEST}                         /* add the test to RS */
            MAY_REDUCE := false
            foreach Ti where NEXT_TEST ∈ Ti do
               MARKED[i] := true                           /* mark Ti containing NEXT_TEST */
               if Card(Ti)= MAX_CARD then MAY_REDUCE := 1
            endfor
            if MAY_REDUCE then                             /* try to reduce MAX_CARD */
               MAX_CARD := Max(Card(Ti)), for all i where MARKED[i] = false
         endwhile
      until CUR_CARD = MAX_CARD
end ReduceTestSuite.
---------
function SelectTest(SIZE, LIST)
/* this function selects the next ti to be included in RS */

declare     COUNT: array[1..nt]
begin
      foreach ti in LIST do compute COUNT[ti], the number of unmarked Tj's of cardinality SIZE containing ti
      Construct TESTLIST consisting of tests from LIST for which COUNT[i] is the maximum
      if Card(TESTLIST)=1 then return(the test case in TESTLIST)
      elseif SIZE = MAX_CARD then return(any test case in TESTLIST)
      else return(SelectTest(SIZE+1, TESTLIST))
end SelectTest.
```

Figure 2: Algorithm *ReduceTestSuite* for finding a representative set from a group of sets

The authors demonstrated the technique's efficiency by analyzing its worst-case run-time. Let n denote the number of tests sets T_i, nt denote the number of test cases t_i, and MAX_CARD the maximum cardinality within the group of sets. *ReduceTestSuite* consisted of two data-intensive steps: computing the occurrences of test cases within test sets of varying cardinality and selecting the next test case to add to the optimized set. The first step took

$O(n * MAX_CARD)$ because there are n sets that were examined once. The second step required examining the occurrences of each test case, which required at most $O(nt * MAX_CARD)$. This was repeated at most n times because the selected test case is covered by at least one other test set. Thus, the overall runtime was $O\left(n\left(n+nt\right)MAX_CARD\right)$. The authors ran simulations of their algorithm against several test programs, which proved its cost-effectiveness as it performed better in practice. Their results are shown in Table 2.

Procedure	Test Cases	Actual Associated Testing Sets	Constructed Associated Testing Sets
trityp	16	1.50	9.28
atof	2	.07	.13
getop	4	.28	.80
calc	7	.23	.60
qsort	5	.10	.30
trityp2	19	.27	2.35
sqroot	6	.07	.35
sqroot2	6	.10	.41
sqroot3	6	.25	.62
sqroot4	5	.08	.20
sqroot5	6	.10	.25

Table 2: Run-times for ReduceTestSuite for Actual and Constructed Associated Testing Sets

In each iteration, they executed the algorithm against a program ("procedure" column) and recorded the actual associated testing sets runtime (i.e. the observed runtime) and the constructed associated testing sets runtime (i.e. worst-case calculated runtime). The results showed that the algorithm's actual runtime was between 46% and 88% better than the estimated worst-case runtime.

Finally, the authors conducted several experiments during the program development, program maintenance for program improvement, and program enhancement phases. The coverage criterion used is the definition-use pair, or *du-pairs*, which consisted of the definition

and use of a variable within its code. They defined full coverage as testing all *du-pairs.* The program development phase consisted of typical functional testing after program development was completed, Experiment 1, and is shown in Table 3. The technique was then used to reduce the test suite and replaced the original test cases in the later experiments.

Procedure	Source Lines	du-pairs	Original Test Cases	Redundant Test Cases	Reduction (%)
trityp	21	39	16	3	18.7
atof	17	63	2	1	50.0
getop	19	33	5	3	60.0
calc	33	3	11	4	36.4
qsort	20	43	4	2	50.0
sqroot	19	13	5	2	40.0

Table 3: Experiment 1 - Reduction during Program Development

The results of Experiment 2, testing after program maintenance for performance improvement, are shown in Table 4. The authors made implementation changes to the programs without changing their functionality, such as making them more efficient or changing their internal structure.

Procedure	Source Lines	du-pairs	Original Test Cases	Redundant Test Cases	Reduction (%)
trityp2	30	42	13	7	54.6
sqroot2	21	25	6	2	33.3
sqroot3	33	44	5	1	20.0
sqroot4	17	17	7	2	28.6
sqroot5	17	24	5	1	20.0

Table 4: Experiment 2 - Reduction during program maintenance for performance improvement

Table 5 depicts the results of Experiment 3, where the technique was used during program maintenance for program enhancements. Here the authors modified the programs, both functionally and structurally, by adding new features and modifying existing ones.

Procedure	Source Lines	du-pairs	Original Test Cases	Redundant Test Cases	Reduction (%)
calc2	41	4	80	0	0.0
calc3	60	4	13	4	30.8
calc4	72	4	14	0	0.0
calc5	86	16	18	3	16.7
getop2	27	57	4	1	25.0
getop3	38	69	5	2	40.0

Table 5: Reduction during program maintenance for program enhancements

The results showed a decrease in the total number of test cases in almost all experiments, with the test suites reduction ranging from 19% to 60% during program development, 20% to 55% when structural changes were introduced during maintenance, and 0% to 40% when functional and structural enhancements were introduced during maintenance.

The study showed that the size of the test suite can be reduced using analysis techniques and coverage criteria in a similar fashion as Xavier et al. (2008). The authors demonstrated that the actual runtime of the algorithm was significantly better than the worst-case $O(n^2)$ time complexity for the small test programs (less than 100 lines of code). However, the technique may not be practical for large systems (e.g. 1 million lines of code) as the time complexity became very large and increasingly significant. Finally, the definition of du-pairs, even if automated, was impractical for a large-scale self-adaptive system due to their dynamic execution paths that were difficult to predict at runtime (IBM, 2003).

In Galeebathullah and C.P.Indumathi (2010) the authors proposed a test suite reduction approach by selecting a minimum set of effective test cases from the application's test space in an effort to reduce the overall cost of software testing, in a similar fashion as Harrold et al. (1993). The technique also omitted redundant test cases and included only those that were the most effective in providing the greatest degree of test coverage. In this instance, coverage was defined as the degree to which a test plan satisfied the greatest number of requirements tested.

The authors utilized set theory to define the minimized test suite, $T_{\min}$, as the intersection between the set of test cases, Ti, satisfying requirements in the requirements set, R, with the set of requirements satisfied, Ri, by the test cases in the original test suite, T. For example, the table below depicts a test case coverage matrix, which contains the relationships identified between requirements and test cases, where test case 1, t_1, satisfies the test coverage of requirements 1, 3, and 5, and so on. As shown, t_1 and t_4 satisfy all requirement testing which results in the omission of t_2 and t_3 from $T_{\min}$.

		Test Case			
Requirement	**Cardinality**	**t_1**	**t_2**	**t_3**	**t_4**
1	2	X		X	
2	2		X		X
3	3	X	X	X	
4	2	X			X
5	2			X	X

Table 6: Test Case Coverage Matrix

The authors utilized the test suite size reduction (SSR) metric to calculate the percentage of overall test suite reduction, defined below in Equation 16:

$$SSR = \frac{T - T_{\min}}{T}$$

Equation 16: SSR Metric

Where T was the number of original test cases, $T_{\min}$ was the number of test cases in the reduced set, and SSR was the reduction percentage, where a larger value denoted greater test suite reduction. They demonstrated the technique's effectiveness using a small case study which produced similar results as traditional greedy and HGS heuristic methods (Chvatal, 1979).

The technique was relatively simple to implement given that all required information, such as the requirement and test case sets, were captured in a machine-readable format so that

the algorithm could determine their relationships. Alternatively, the evaluator could manually complete the preparation step if the number of test cases and relationships was small. Otherwise, the labor required for the automation may not have justified the cost savings, especially if they were large or complex (Cin et al., 2002; Galeebathullah & C.P.Indumathi, 2010). Further, conversion to a machine-readable format may not be possible for all requirements, such as those written in prose (Potts, 1995), or for test cases that required human interaction (A. B. Brown et al., 2004) as they are both have associated challenges.

Finally, the technique presented in this study could be further refined to incorporate test and requirement classes and dependencies, where only a single test case needs to be enumerated to validate a class of tests. Ultimately, the cost savings was directly related to the SSR value, which was dependent on the number of elements in the test set (i.e. the number of tests) and relates to Equation 1. This supported this study's direction to reduce the number of considered and enumerated test cases in an effort to reduce resilience benchmarking costs for self-adaptive systems.

Summary

This section discussed several studies that proposed techniques to address the high costs associated with benchmarking, testing system behavior, and verifying requirements in the presence of exceptional conditions. Studies have been discussed that consider the individual test sets and omit the redundant test cases in an effort to minimize the test suite, while seeking to maximize test set coverage of functional requirements (Galeebathullah & C.P.Indumathi, 2010). Other studies have been presented that utilize test coverage criteria and system analysis to further reduce the size of the test suite by omitting ineffective and redundant test cases from a test suite (Barbosa et al., 2005; Harrold et al., 1993; Xavier et al., 2008). Each technique provided a

method for reducing the size of a test suite in an effort to reduce software-testing costs. A reoccurring theme is to utilize system analysis to guide the selection of test cases, with source code analysis being the most effective. The results showed test suite reductions ranging from 10% to 99%, which validated their effectiveness of test suite minimization. The presented studies reinforced the premise of this study that utilizing system analysis and verification of desired runtime behavior can reduce the cost of resilience benchmarking of self-adaptive systems.

The next chapter describes the methodology used for this study, the goal-oriented approach, and the case study utilized to verify the approach's effectiveness.

Chapter 3

Methodology

Overview of Research Methodology

This study detailed an approach that reduced the cost of resilience benchmarking of self-adaptive systems. The approach built upon the risk-based approach proposed by Almeida and Vieira (2012a) while incorporating goal-oriented requirements engineering techniques and theories proposed by Dardenne, Lamsweerde, and Fickas (1993), Feather, Fickas, Lamsweerde, and Ponsard (1998), and van Lamsweerde and Letier (1998).

The guiding principle of the approach was to minimize the effort invested in the definition and enumeration of ineffective changes during the changeload definition process as they contributed negatively toward the overall cost of evaluation (Barbosa et al., 2005; Roberto, 2013). This differed from the minimization approaches discussed in the preceding sections as they required the definition of an exhaustive changeload first and then discarded the ineffective and redundant changes (Barbosa et al., 2005; Harrold et al., 1993; Quadri & Farooq, 2010). The overhead incurred by defining a large number of changes could outweigh the cost-savings achieved by the minimization technique (B. Cheng et al., 2009; Leung & White, 1991). Therefore, this research proposed a goal-oriented approach that balanced the cost-effectiveness and coverage of resilience evaluation of self-adaptive systems by utilizing system knowledge to avoid the costs incurred by the definition and enumeration of ineffective changes. It is followed by a case study that demonstrated its effectiveness.

This was a valid method as it has been performed in the previous changeload study, the risk-based approach proposed by Almeida and Vieira (2012a), in dependability faultload studies

(Madeira et al., 2002; Vieira & Madeira, 2003, 2004), and in the test suite reduction studies discussed in the Chapter 2 (Barbosa et al., 2005; Galeebathullah & C.P.Indumathi, 2010; Harrold et al., 1993), where the respective techniques were proposed and validated via a case study on a fictitious system, or by experimentation. A description of the goal-oriented approach is presented in the next section followed by a description of the case study that demonstrated its application and the application of the risk-based approach.

Approach Overview

The goal-oriented approach extended the risk-based approach by incorporating additional analysis and identification techniques in each step of the process. The risk-based approach consisted of five primary steps focused on the identification and definition of the system and its relevant changes, as discussed in detail in the Problem Statement. They are:

- Step A: Identification of the Base Scenario
- Step B: Identification of Change Scenarios
- Step C: Definition of Change Scenario Attributes
- Step D: Evaluation of Change Scenario Attributes
- Step E: Definition of the Changeload

The goal-oriented approach mirrored the five-step process of the risk-based approach, with the following steps listed below:

- Step A: Identification of System Goals
- Step B: Identification of Obstacles
- Step C: Definition of Obstacle Attributes
- Step D: Evaluation of Obstacle Attributes
- Step E: Definition of the Changeload

Extensions to each step are described below.

Step A: Identification System Goals

The identification of the system's goals is the most critical milestone of the changeload definition process as they are the driver for the identification and characterization of the change scenarios that may affect the system at runtime (Almeida & Vieira, 2012a). The goal-oriented approach extended the identification of the base scenario, Step A of the risk-based approach, to include elaboration and refinement of the previously defined generic goals using WHY and HOW goal refinement techniques.

The HOW goal refinement technique is a method for refining a goal until concrete sub-goals are identified (van Lamsweerde & Letier, 2000). For example, the evaluator determines HOW the system accomplishes the goal of "maintaining high performance" by analyzing its components and associating the "minimization of response time" sub-goal to it. The low-level goals are specified using a similar notation as proposed by Almeida and Vieira (2012a), shown below in Equation 17 and an example is shown in Equation 18, where attainment of a high-level goals implies attainment of its lower-level goals.

$$G = \{ \mathrm{g} \mid \mathrm{g} \Leftarrow G_{high-level} \}$$

Equation 17: Definition of Low-Level System Goals

$$G = \{ \mathrm{g} \mid \mathrm{g} \Leftarrow G_{high-level} \} = \left\{ \begin{array}{c} \text{minimize response time,} \\ \text{maximize content fidelity} \end{array} \right\}$$

Equation 18: Low-Level System Goal Definition for example Self-System A

If the set of system requirements is represented by R, the set of environmental assumptions, As, the set of domain properties, D, then the following relationship must hold true for each goal, g, in G, as the relationship in Equation 19 shows. Assumptions are defined in Step B.

$$\{R, As, D\} \models g \quad \text{with} \quad \{R, As, D\} \not\models false$$

Equation 19: Goal Attainment Verification

The relationships state that each goal must be attainable by the system within the constraints imposed by its operating environment and requirements (van Lamsweerde, 2000). Domain properties are properties of an object or operation in the environment that holds independent of the system and includes physical laws, regulations, and other constraints imposed by environmental agents (van Lamsweerde & Letier, 2000).

The WHY goal refinement technique provided a method of discovering implicit higher-level goals from stated goals (van Lamsweerde, 2000). Stated goals were analyzed and continually asked WHY the goal is important, necessary, and relevant to the system in order to discover the higher-level goals underpinned by it. This process continued until relationships could be constructed between all stated and identified goals. For example, it was determined that the goal of "maintaining high performance" existed to ensure that more visitors could be served by the system. Therefore, the "serve more visitors" goal was the new root goal and "maximize performance" became its sub-goal. The combination of goal refinement techniques guided the system analysis to determine the underlying sub-goals of the system's generic goals and establish relationships between them. Then the underlying assumptions for attainment and their responsible agents were identified in Step B. The agent is then directly exercised to leverage the cost-reduction technique recommended by Barbosa et al. (2005).

Step A included a visual aid to graphically depict the goal hierarchy and highlight the goal dependencies and relationships, described below. The inclusion of a goal graph provided the basis for goal prioritization, documentation, and additional analysis conducted in the following steps.

In Dardenne, Lamsweerde, and Fickas (1993) the authors proposed the KAOS methodology of goal-oriented requirements engineering, which was later extended in van Lamsweerde (2000) and G. Brown, Cheng, Goldsby, and Zhang (2006) to include obstacles. The extension contained a graphical specification for the representation of goal refinement trees and their relationships. Figure 3 depicts the specification for unrefined / soft goals, refined / formalized goals, sub-goal to goal links, sub-goal to goal OR-refinement links, sub-goal to goal AND-refinement links, goal conflicts, system assumptions, obstacles, agents, and actions.

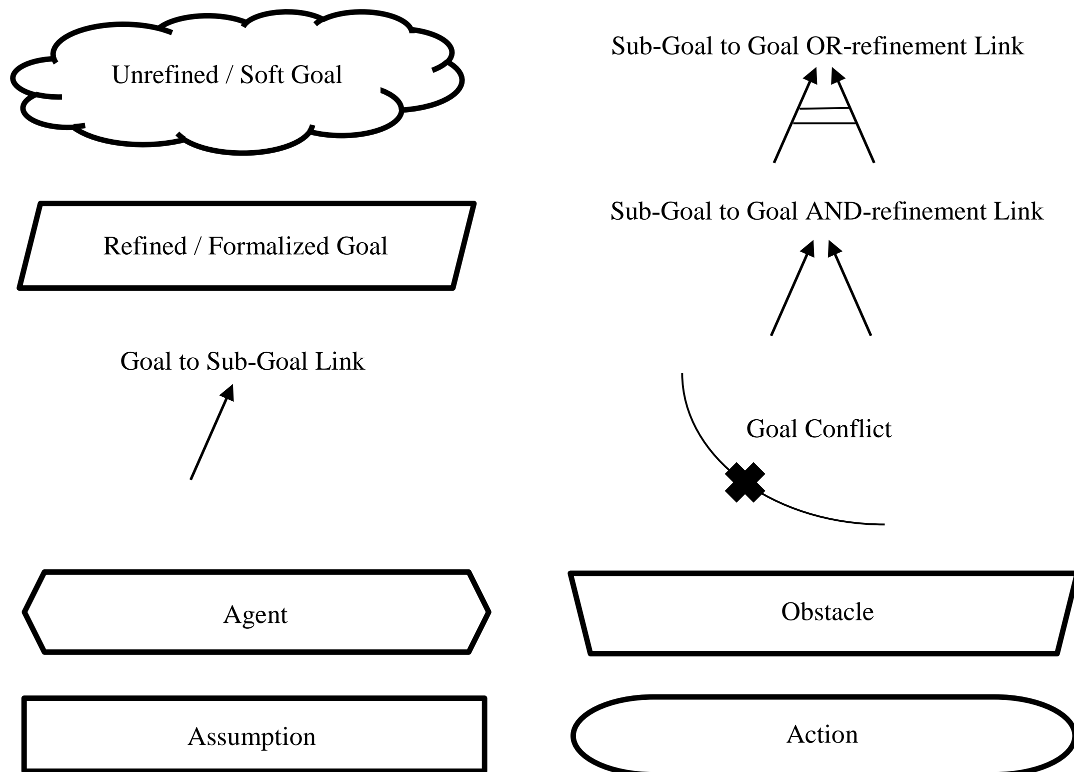

Figure 3: KAOS Glyph Specification

A refined goal graph was created utilizing the KAOS specification and the information derived from the analysis of the system, in the format depicted in Figure 4.

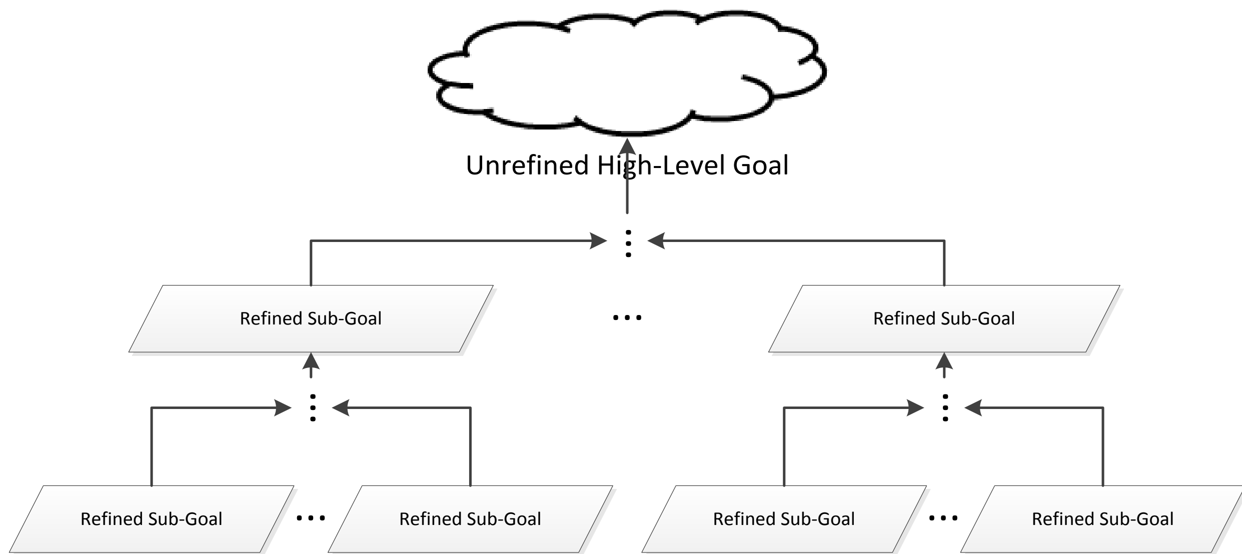

Figure 4: Initial Goal Refinement Graph Format

The initial goal graph for example Self-System A was simply the unrefined goal to "maximize performance," as depicted in Figure 5.

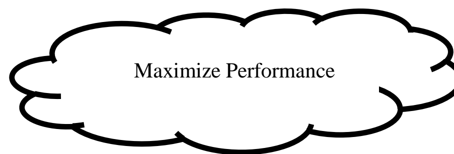

Figure 5: Initial Goal Graph of example Self-System A

The goal-refinement graph illustrated the relationships between the soft goals and their refined sub-goals. Figure 6 and Figure 7 depict the previously refined goals utilizing the HOW and WHY refinement techniques, respectively.

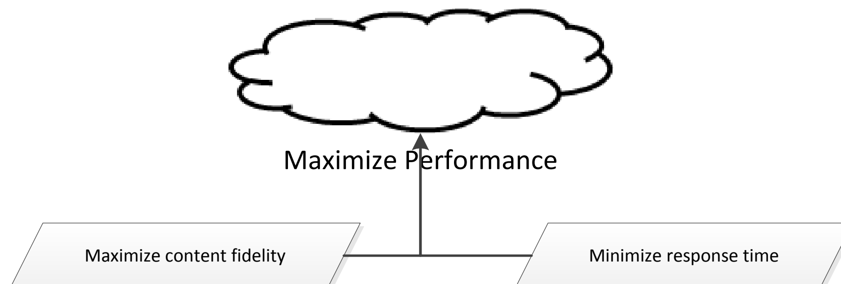

Figure 6: HOW Goal Refinement Graph for example Self-System A

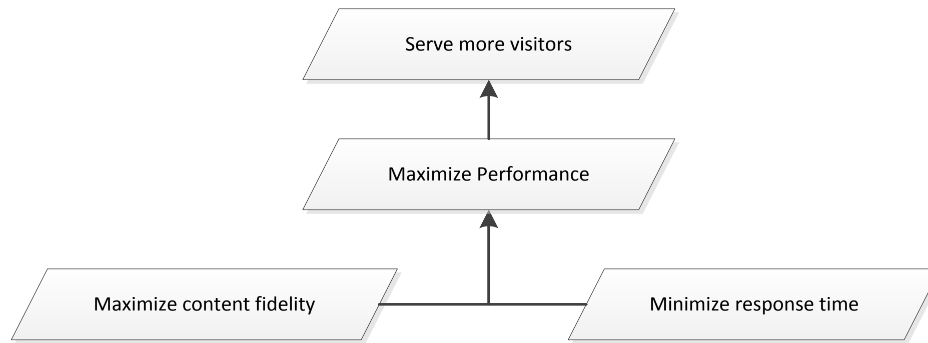

Figure 7: WHY Goal Refinement Graph for example Self-System A

The HOW goal-refinement graph was created in a top-down approach, where the unrefined goal was refined and specified into formalized sub-goals. The WHY goal-refinement graph was created in a bottom-up approach, where the refined and unrefined goals were elaborated and correlated with others to develop higher-level relationships.

The inclusion of a visualization technique improved upon the original approach as it allowed for a more intuitive analysis of the interactions and relationships of the system's goals (Almeida & Vieira, 2012a; Morandini, Penserini, & Perini, 2008; van Lamsweerde, 2001). Further, visualization techniques have been shown to be an essential feature for communicability and understanding of complex systems as they simplify the depiction of complex relationships, dependencies, and logic (G. Brown et al., 2006; B. Cheng et al., 2009; van Lamsweerde, 2000, 2001).

Step B: Identification of Obstacles

Step B, the identification of obstacles, consisted of two sub-steps. The first was the identification of system actions, responsible agents, and assumptions of the system and their incorporation into the initial goal graph created in Step A. The second consisted of expanding the goal refinement graph by identifying and incorporating the obstacles that affected the previously identified actions, agents, assumptions, and goals.

Step B Part 1: Action, Agent, and Assumption Analysis

An action is something the system performs, such as an act or operation, to achieve or maintain a runtime goal in response to a change (Dardenne et al., 1993). The SUB's runtime behavior was revealed by identifying the system's self-adaptive actions. This was accomplished by applying additional HOW refinement to the goal refinement graph defined in Step A and asking HOW the SUB ensures the attainment of each runtime goals. The actions are defined as depicted in Equation 20.

$$A = \{a \mid a \Rightarrow g\}$$

Equation 20: Definition of Self-Adaptive Action

For example, the evaluator reviews example Self-System A's associated documentation, or source code, and identified that it is capable of increasing and decreasing the fidelity of served content in response to measured response time in an effort to ensure the goal of maximum performance (S. W. Cheng et al., 2009). Its self-adaptive actions were captured as shown in Equation 21.

$$A = \{a \mid a \Rightarrow g\} = \left\{ \begin{array}{c} \text{increase content fidelity,} \\ \text{decrease content fidelity,} \\ \text{measure response time} \end{array} \right\}$$

Equation 21: Self-Adaptive Action Definition for example Self-System A

Agent analysis is conducted, followed by assumption analysis, on the SUB's goal refinement graph. An agent is a part of the SUB's operation, including human beings, physical devices, components, and code blocks, that had the ability to make runtime decisions of their behavior based on their operational context (Dardenne et al., 1993). Agent analysis pertained to the review of system actions and the identification of the system's agent responsible for performing each of the identified actions defined in A (Dardenne et al., 1993).

Let Ag be the set of all system agents, ag , which perform an action, a , in the set of identified actions, A , in response to a change, c , in the set of all possible changes, CS , as defined in Equation 22.

$$Ag = \left\{ ag \middle| ag \Rightarrow a, a \in A, c \in CS \right\}$$

Equation 22: Definition of Self-Adaptive Agents

For example, the documentation, or source code, was again reviewed for example Self-System A and asked WHO is responsible for the identified actions in A . Three primary agents were discovered, including a sensor to measure response time, an effector to increase and decrease content fidelity served to users, and a self-adaptive control loop responsible for the coordination of both agents, as shown in Equation 23.

$$Ag = \left\{ ag \middle| ag \Rightarrow a, a \in A, c \in CS \right\} = \left\{ \begin{matrix} \text{response time sensor,} \\ \text{fidelity effector,} \\ \text{self-adaptive control loop} \end{matrix} \right\}$$

Equation 23: Self-Adaptive Agent definition for example Self-System A

The goal refinement graph was then expanded with the identified actions and agents (bold outline) in the format defined in Figure 8. Figure 9 depicts the expanded goal refinement graph for example Self-System A.

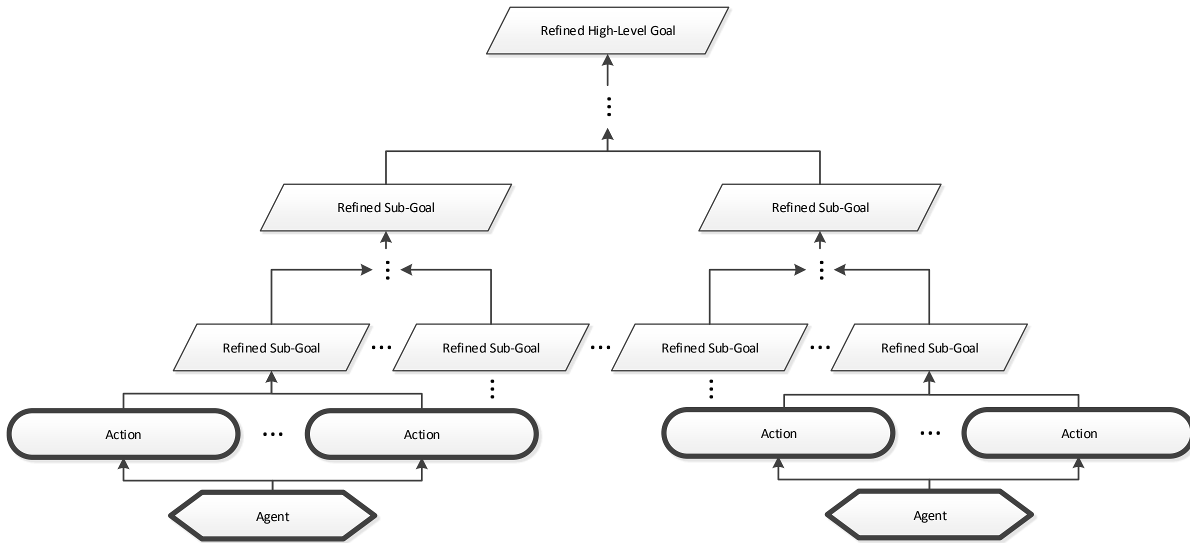

Figure 8: Expanded Goal Refinement Graph with Actions and Agents Format

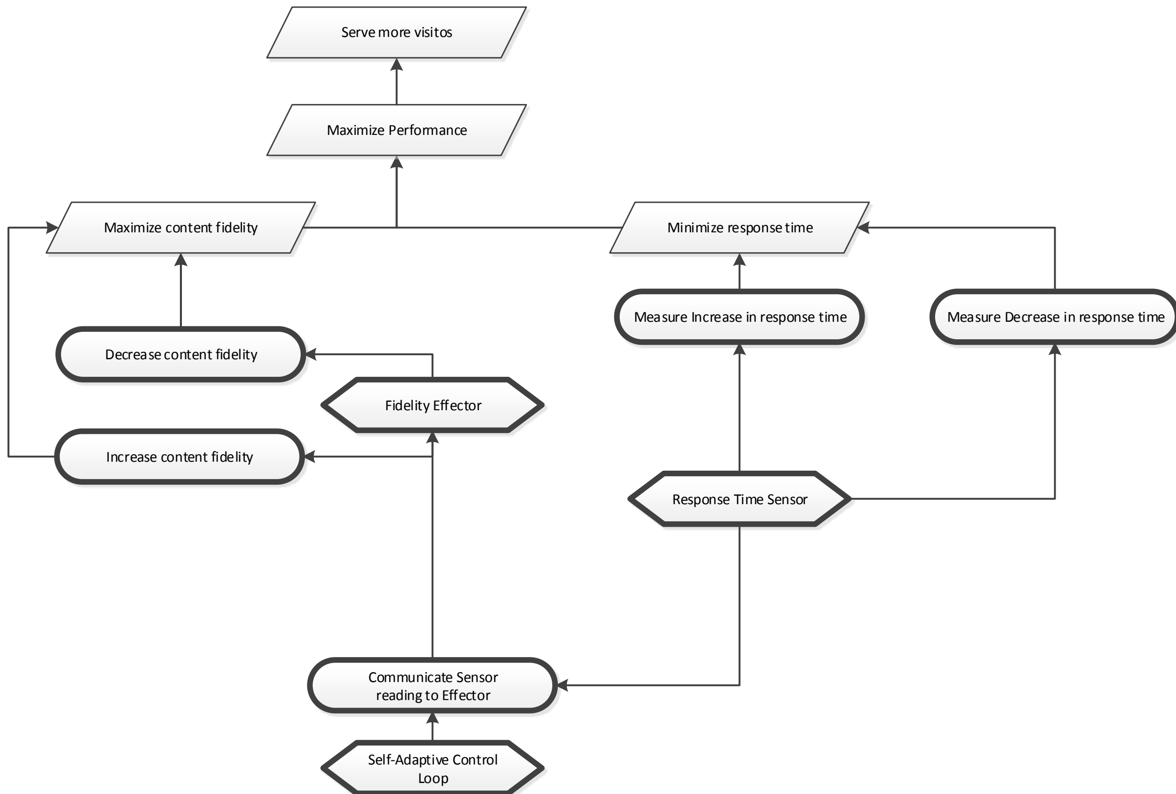

Figure 9: Expanded Goal Refinement Graph with Actions and Agents example Self-System A

Finally, assumption analysis is conducted on the goal refinement graph. Self-adaptive systems are designed to ensure the system's ability to operate as expected while experiencing runtime changes, especially changes in runtime assumptions that are assumed constant throughout its execution (Cámara, Lemos, Laranjeiro, Ventura, & Vieira, 2013). Thus, the inclusion of assumption analysis was vital for the resilience benchmarking of self-adaptive systems as unpredictable and changing assumptions were a source of major problems (van Lamsweerde, 2000).

An assumption is a fact pertaining to the SUB's goals, agents, actions, or their relationships, that is expected to be true at runtime (Feather, Fickas, Lamsweerde, & Ponsard, 1998). While the classic definition of assumptions only included environmental assumptions (van Lamsweerde, 2000), assumptions related to any aspect of the system were considered to ensure coverage of all runtime constraints and possible sources of change.

Assumption analysis is the process of analyzing the goal refinement graph to identify hidden assumptions and operational constraints that are often taken for granted (Feather et al., 1998). Changes in runtime assumptions introduce unforeseen operational conditions, which may lead to unexpected runtime behavior with undesirable results, such as loss of goal attainment or failure (B. Cheng et al., 2009; van Lamsweerde, 2000). Each goal, action, and agent identified in the goal refinement graph was analyzed and asked the question of WHAT conditions needed to exist for a goal to be achieved and maintained, for an action to be performed with the expected outcomes, and an agent to operate as desired.

Let As be the set of all assumption sub-sets, As_i, which contain the set of assumptions, as_i, affecting an action, agent, or goal node, i, in the goal refinement graph, as shown in Equation 24, that satisfies the relationship depicted in Equation 25. Equation 25 states that

agents are able to perform their actions, and those actions are achieve the system's goals, when all assumptions meet expectations.

$$As_i = \{as \mid as \text{ is an assumption on } i, i \in A \vee Ag \vee G\}$$
$$As = \{As_i\}$$

Equation 24: Definition of an Assumption

$$\begin{aligned} \{Ag, As_{Ag}\} &\models A \\ \{A, As_A\} &\models G \\ \{G, As_G\} &\not\models false \end{aligned}$$

Equation 25: Assumption and Node Satisfaction Relationship

For example, the increase and decrease content fidelity actions are analyzed and it is reasoned that access to the configuration file was necessary for this action to occur. Similarly, the fidelity effector was assumed to be functioning properly to perform those actions. Finally, the fidelity effector is assumed to have sufficient resources available to function properly, such as CPU and memory. This process continued for each node until all were analyzed and their assumptions identified, as shown in Equation 26.

$$\begin{aligned} As_A &= \{as \mid as \text{ is an assumption on } a, a \in A\} &&= \begin{Bmatrix} \text{Configuration file is accessible,} \\ \text{Valid Sensor Reading} \end{Bmatrix} \\ As_{Ag} &= \{as \mid as \text{ is an assumption on } ag, ag \in Ag\} &&= \begin{Bmatrix} \text{Effector Operational,} \\ \text{Sensor Operational,} \\ \text{Sufficient Resources Available} \end{Bmatrix} \\ As_G &= \{as \mid as \text{ is an assumption on } g, g \in G\} &&= \{\text{Sufficient Resources Available}\} \\ As &= \{As_A, As_{Ag}, As_G\} \end{aligned}$$

Equation 26: Assumption Definition for example Self-System A

The identified assumptions were incorporated into the goal refinement graph (bold outline) in the format specified in Figure 10 and depicted in Figure 11.

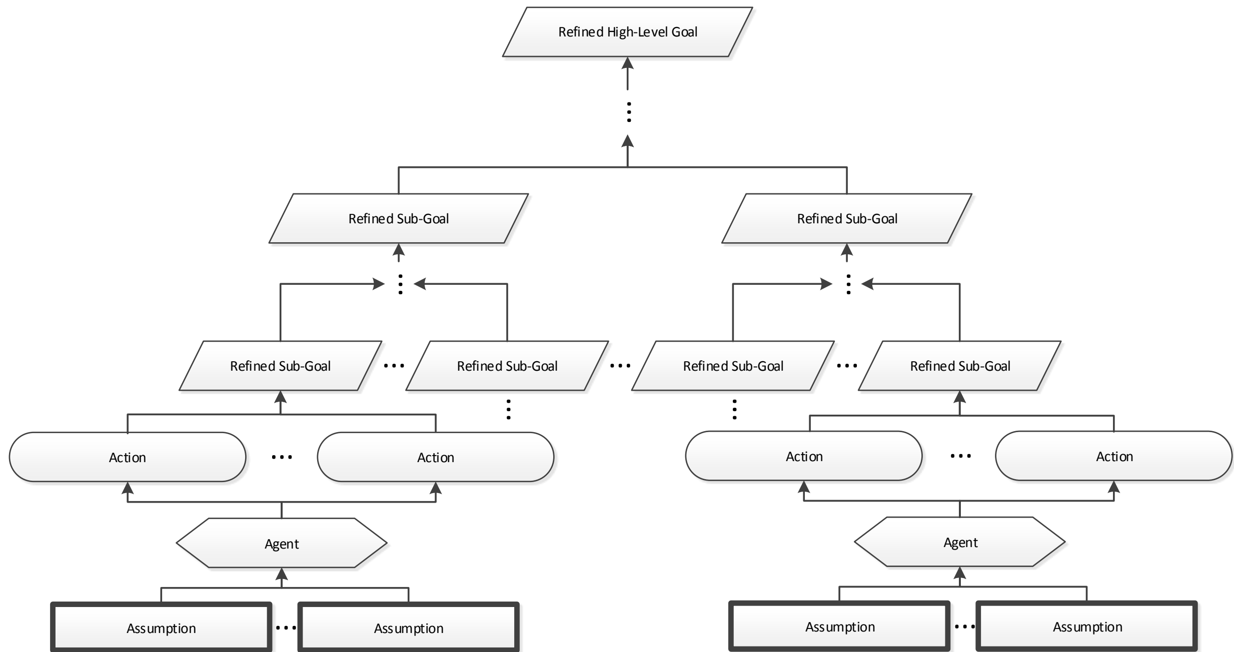

Figure 10: Expanded Refinement Goal Graph with Actions, Agents, and Assumptions Format

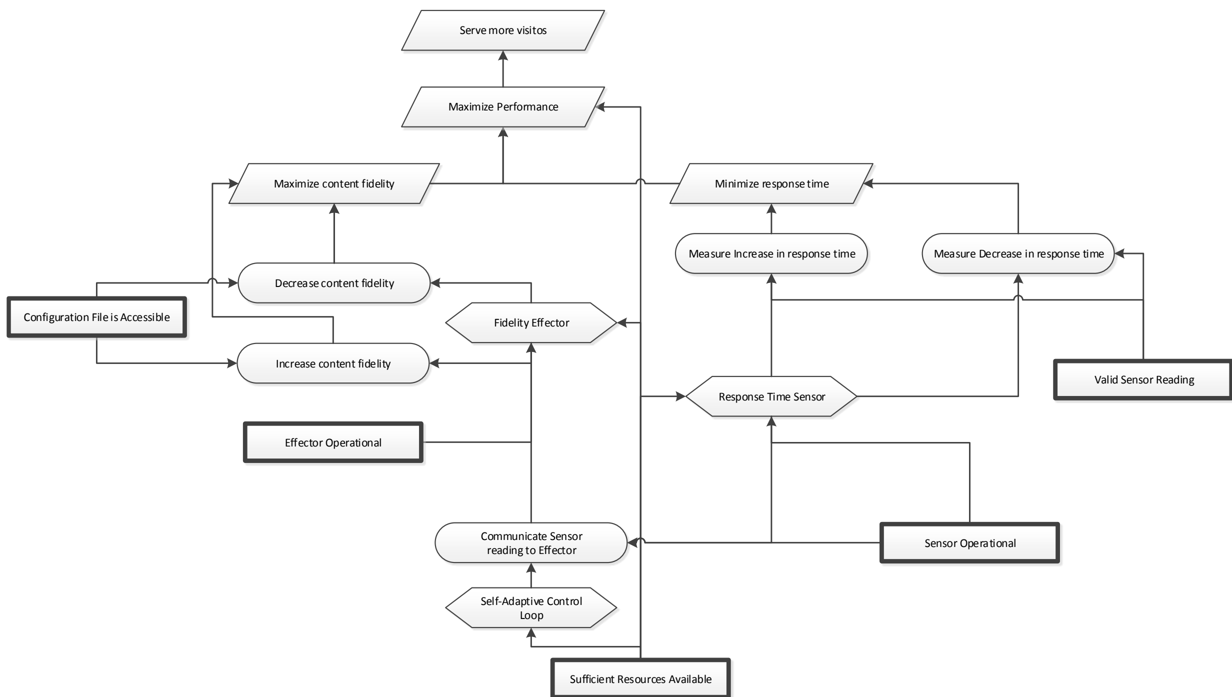

Figure 11: Expanded Goal Refinement Graph with Actions, Agents, and Assumptions for example Self-System A

Step B Part 2: Obstacle Analysis

Obstacle analysis and identification techniques were then employed to identify obstructing conditions under which a goal is unachievable (i.e. Equation 19 was violated). Obstacles may directly obstruct a goal, or indirectly obstruct it, by affecting an assumption, action, or agent required for its attainment (van Lamsweerde & Letier, 2000). Obstacles provided a straightforward method of identifying relevant changes within the system and its environment as they were directly related to the system's runtime goals and changes to runtime assumptions (van Lamsweerde & Letier, 2000). Each assumption, agent, action, and goal identified in the goal refinement graph was analyzed and asked the question of WHAT obstructing conditions may the system face with at runtime that would cause a goal to be unattainable, cause an action to be performed with undesired outcomes or not at all, or cause an agent to operate inconsistently or fail.

Let O be the set of all obstacle sub-sets, O_i, which contain the set of obstacles, o_i, obstructing an assumption, action, agent, and / or a goal node, i, in the goal refinement graph as shown in Equation 27, satisfying the relationship depicted in Equation 28.

$$O_i = \{ o_i \mid o \text{ obstructs } i, i \in As \wedge A \wedge Ag \wedge G \}$$
$$O = \{ O_i \}$$

Equation 27: Definition of an Obstacle

$$\{as, ag, a, g\} \quad \models \neg o \quad \text{(obstruction)}$$
$$\{O, D\} \quad \mid \mathit{false} \quad \text{(domain-consistency)}$$

Equation 28: Obstacle Satisfaction Relationship

The relationship states that the obstacle must be consistent with what is known of the domain (domain-consistency) and that its negation, that is, the absence of obstructing conditions or runtime changes yields the necessary conditions for goal achievement (van Lamsweerde &

Letier, 2000). For instance, an obstacle could not state that the system is simultaneously on- and off-line as such behavior is infeasible.

Example Self-System A's assumption of sufficient resources being available was analyzed and it was reasoned that a lack of available resources, such as CPU or memory exhaustion, would obstruct the agent's ability to function and it's attainment of the goal to maximize performance. This analysis continued until all nodes had been evaluated, as depicted in Equation 29.

$$
\begin{aligned}
O_{As} &= \{ o \mid o \text{ obstructs } as, as \in As \} &&= \left\{ \begin{array}{l} \text{Configuration file locked / Inaccessible,} \\ \text{Resource Exhaustion (CPU),} \\ \text{Resource Exhaustion (Memory),} \\ \text{No Sensor Reading,} \\ \text{Invalid Sensor Reading} \end{array} \right\} \\
O_{Ag} &= \{ o \mid o \text{ obstructs } ag, ag \in Ag \} &&= \left\{ \begin{array}{l} \text{Effector Failure,} \\ \text{Effector Not Available,} \\ \text{Sensor Failure,} \\ \text{Sensor Not Available} \end{array} \right\} \\
O_{A} &= \{ o \mid o \text{ obstructs } a, a \in A \} &&= \{ \text{Communication Error} \} \\
O_{G} &= \{ o \mid o \text{ obstructs } g, g \in G \} &&= \left\{ \begin{array}{l} \text{Resource Exhaustion (CPU),} \\ \text{Resource Exhaustion (Memory)} \end{array} \right\} \\
O &= \{ O_{As}, O_{Ag}, O_{A}, O_{G} \}
\end{aligned}
$$

Equation 29: Assumption Definition for example Self-System A

The identified obstacles were well suited to describe relevant changes to the SUB as they were based on the system's capabilities, goals, assumptions, domain knowledge, and captured its undesirable runtime conditions.

Finally, the identified obstacles were incorporated into the goal refinement graph (bold outline) to provide detail of their interaction and effects on the overall system in the format depicted in

Figure 12. Figure 13 depicts the expanded goal graph for the example Self-System A.

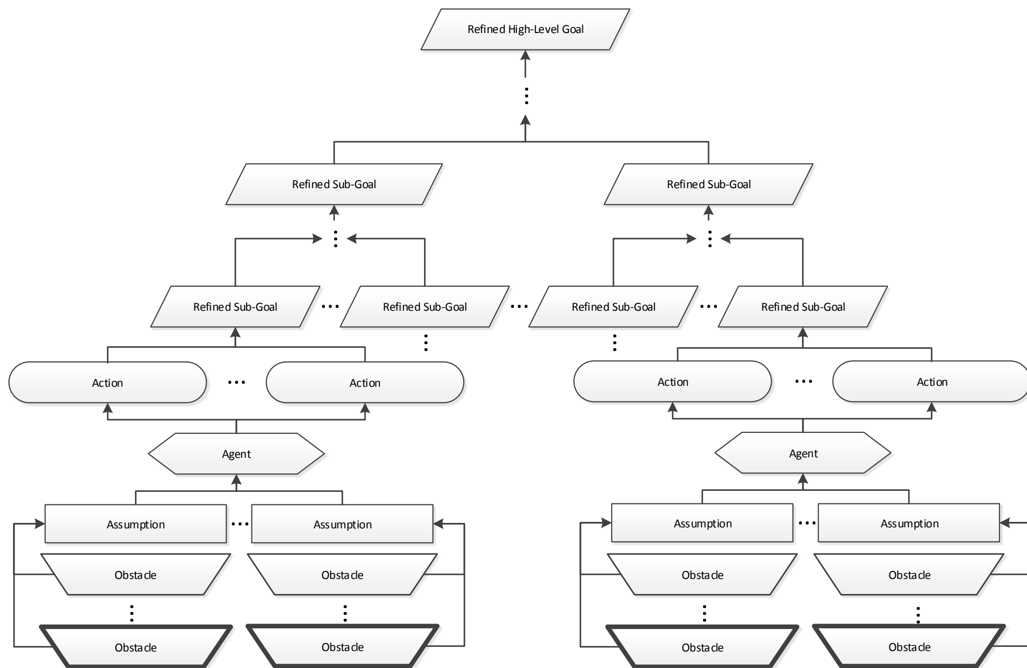

Figure 12: Expanded Goal Refinement Graph with Actions, Agents, Assumptions, and Obstacles Format

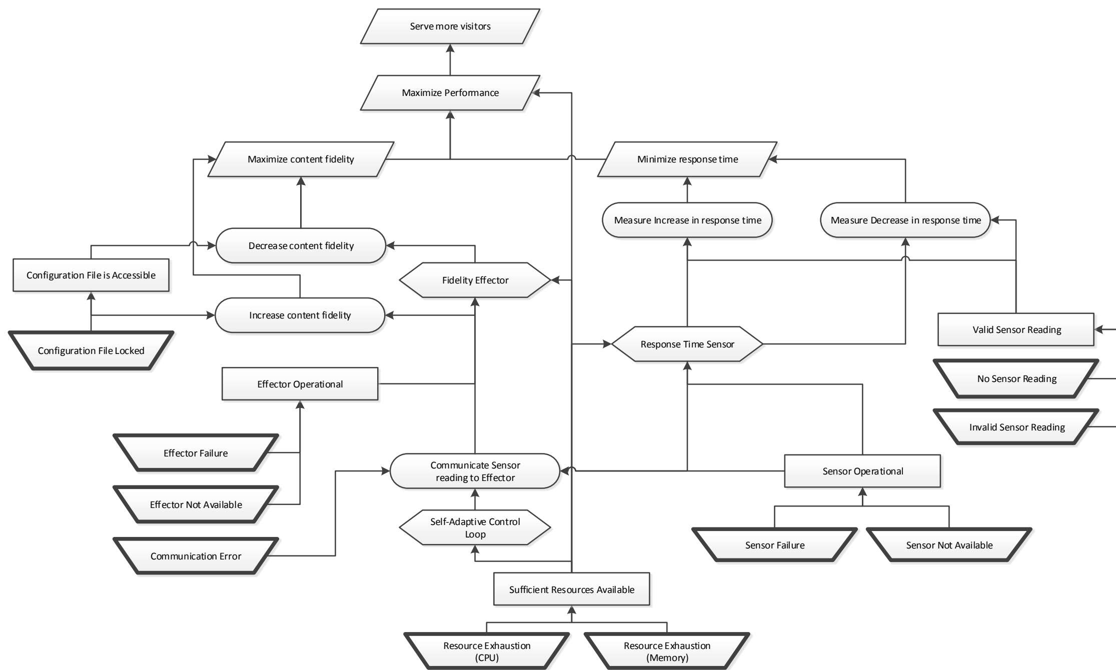

Figure 13: Expanded Goal Refinement Graph with Obstacles, Assumptions, Agents, and Actions for example Self-System A

Step C: Definition of Obstacle Attributes

The definition of change scenario attributes in the risk-based approach, Step C, defined the change scenario attributes of impact and probability utilizing a combination of expert opinion and multi-voting when field data was not available (Almeida & Vieira, 2012a). The risk-based approach also used a qualitative scale for change scenario impacts, such as “medium” and “minimal”, without finite thresholds, as presented in Chapter 1 and shown in Table 7. Each attribute was defined, and assigned in Step D, using expert opinion without clear thresholds or finite boundaries between attribute ranges.

Impact	Probability
Catastrophic	Very High
Critical	High
Marginal	Low
Negligible	Very Low

Table 7: Change Scenario Attributes defined in the Risk-Based Approach

Step C was extended to utilize the previously constructed goal refinement graph to define quantitative measures for each obstacle’s impact attributes utilizing graph theory. Two properties were defined to denote an obstacle’s impact on runtime goals: the obstacle’s shortest distance to a goal (OSDG) and the obstacle’s breadth (OB). The OSDG attribute was defined as the number of graph edges from an identified obstacle to its nearest goal, or the obstacle’s closeness factor to any goal (Kang, Kumar, Harrison, & Yen, 2011).

Let D be the distance matrix of all pair-wise distances, d_{ij}, between each obstacle, o_i, in the set of defined obstacles, O, and each goal, g_j, in the set of defined goals, G. The OSDG value for obstacle o_i, $OSDG_i$, was defined as the minimal element, $d_{ij_{\min}}$ in the partially ordered set $\left(D,\leq\right)$, as shown in Equation 30.

$$OSDG_i = \left\{ d_{ij_{\min}} \mid \forall d_{ij} \in (D, \leq) : d_{ij_{\min}} \leq d_{ij} \right\}$$

Equation 30: Obstacle's Shortest Distance to a Goal (OSDG)

The OSDG attribute represented the relative impact an obstacle would have on the system if experienced at runtime, where a smaller OSDG value denoted a greater impact on that goal (and the overall system) and an increased likelihood of runtime disruptions (Kang et al., 2011).

The OB attribute represented the total number of goals affected by the activation of an obstacle o_i, and was defined as the sum of all reachable goal nodes g_j from o_i, as defined in Equation 31.

$$OB_i = \sum r_{ij}, \text{ where } r_{ij} = \left\{ \begin{array}{l} 1 \text{ if } g_j \text{ is reachable from } o_i \text{ and } g_j \in G \\ 0 \text{ otherwise} \end{array} \right\}$$

Equation 31: Obstacle's Breadth of Impact

The goal-oriented approach utilized the OSDG and OB attributes to define obstacle attribute ranges mathematically. These definitions, as well as the mapping of the goal-oriented OSDG to the risk-based impact and the goal-oriented OB to risk-based probability, are shown in Table 8 and Table 9. Note that the mapping of OB to probability did not imply equivalence and was included for comparative purposes only.

Risk-Based Impact Attribute	Goal-Oriented OSDG Attribute
Catastrophic	$\left[1, \min\left(OSDG\right)\right]$
Critical	$\left(\min(OSDG), \frac{1}{3}\left(2\min(OSDG) + \max(OSDG)\right)\right]$
Marginal	$\left(\frac{1}{3}\left(2\min(OSDG) + \max(OSDG)\right), \frac{1}{3}\left(\min(OSDG) + 2\max(OSDG)\right)\right]$
Negligible	$\left(\frac{1}{3}\left(\min(OSDG) + 2\max(OSDG)\right), \max\left(OSDG\right)\right]$

Table 8: Risk-Based Change Scenario Impact Attribute mapping to Goal-Oriented Obstacle OSDG Attribute

Risk-Based Probability Attribute	Goal-Oriented OB Attribute
Very High	$\left[\lvert G\rvert, \frac{3}{4}\lvert G\rvert\right)$
High	$\left[\frac{3}{4}\lvert G\rvert, \frac{1}{2}\lvert G\rvert\right)$
Low	$\left[\frac{1}{2}\lvert G\rvert, \frac{1}{4}\lvert G\rvert\right)$
Very Low	$\left[\frac{1}{4}\lvert G\rvert, 0\right)$

Table 9: Risk-Based Change Scenario Probability Attribute mapping to Goal-Oriented Obstacle OB Attribute

The OSDG attribute's value range was defined as $\left[1, \max\left(OSDG\right)\right]$, where a value of one described the scenario where an obstacle is a child of a goal node. The value of $\max\left(OSDG\right)$ defined the maximum distance of any obstacle to any goal node for the goal-refinement graph. The OSDG attribute ranges were divided into four uniform ranges to ensure comparability with the risk-based approach's four-value scale. The OB attribute's value range was defined as $\left(0, \lvert G\rvert\right]$, where zero was non-inclusive as an obstacle by definition (Equation 28) must obstruct the attainment of at least one goal. The maximum value for OB was the total number of goals in G. Again, the OB attribute was divided into four uniform ranges to ensure comparability with the risk-based approach. Table 10 and Table 11 illustrate the defined and

effective attribute ranges for the example Self-System A. The effective ranges were included to correspond to the computed OSDG and OB integer values.

Risk-Based Impact Attribute	OSDG Attribute Range	OSDG Attribute Effective Range
Catastrophic	[1.0, 2.0]	1 and 2
Critical	(2.0, 3.3]	3
Marginal	(3.3, 4.7]	4
Negligible	(4.7, 6.0]	5 and 6

Table 10: OSDG Attribute for example Self-System A

Risk-Based Probability Attribute	OB Attribute Range	OB Attribute Effective Range
Very High	[4, 3)	4
High	[3, 2)	3
Low	[2, 1)	2
Very Low	[1, 0)	1

Table 11: OB Attribute for example Self-System A

This approach reduced the dependence on expert opinion and the use of subjective attribute thresholds by leveraging graph theory to calculate the obstacle attributes. This provided a basis for defining objective attributes that could be standardized between systems and experiments (Cailliau & Lamsweerde, 2013) to avoid their misinterpretation and improve result comparison (Almeida & Vieira, 2012a). This step could also be automated to further reduce the labor and cost of resilience benchmarking as the attribute definitions were calculated based on graph characteristics and not by subjective or manual means.

Step D: Assignment of Obstacle Attributes

The evaluation of change scenario attributes, Step D of the risk-based approach, was extended to leverage the attributes defined in Step C by calculating the OSDG and OB attributes for each obstacle and assigning its corresponding impact attributes. This step also lent itself to automation as the evaluation of obstacle attributes and attribute assignments were based on their

computed values derived from the goal-refinement graph, without the need for manual analysis, which could further reduce overall benchmarking costs.

The obstacle attributes assignment provided insight into the overall impact of each obstacle, where the directness of an obstacle's impact was defined as its closeness to goal nodes, its OSDG attribute, and the severity of its impact by the number of goals affected, its OB attribute (Jorgensen, 2002). For instance, the "resource exhaustion" obstacle, with an OSDG value of two (Catastrophic) and OB value of four (Very High), had catastrophic effects on the attainment of runtime goals by directly affecting 100% of all runtime goals (bolded outline), as shown in Figure 14. In contrast, the obstacle "locked configuration file", with an OSDG value of 3 (Critical) and OB value of 3 (High), had less of an impact on the attainment of runtime goals than the previous example as it affected fewer goals and in a less direct manner, as shown in Figure 15.

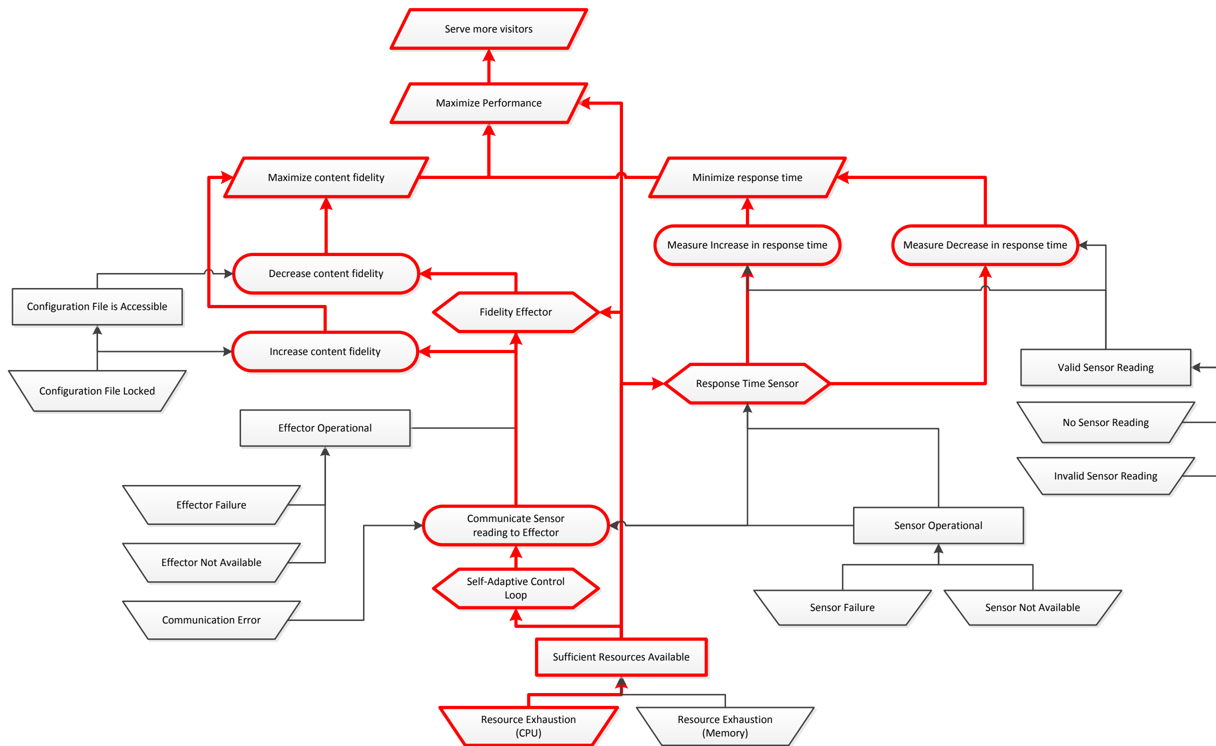

Figure 14: Goal Refinement Graph of Self-System A – Resource Exhaustion (CPU) Obstacle Impact

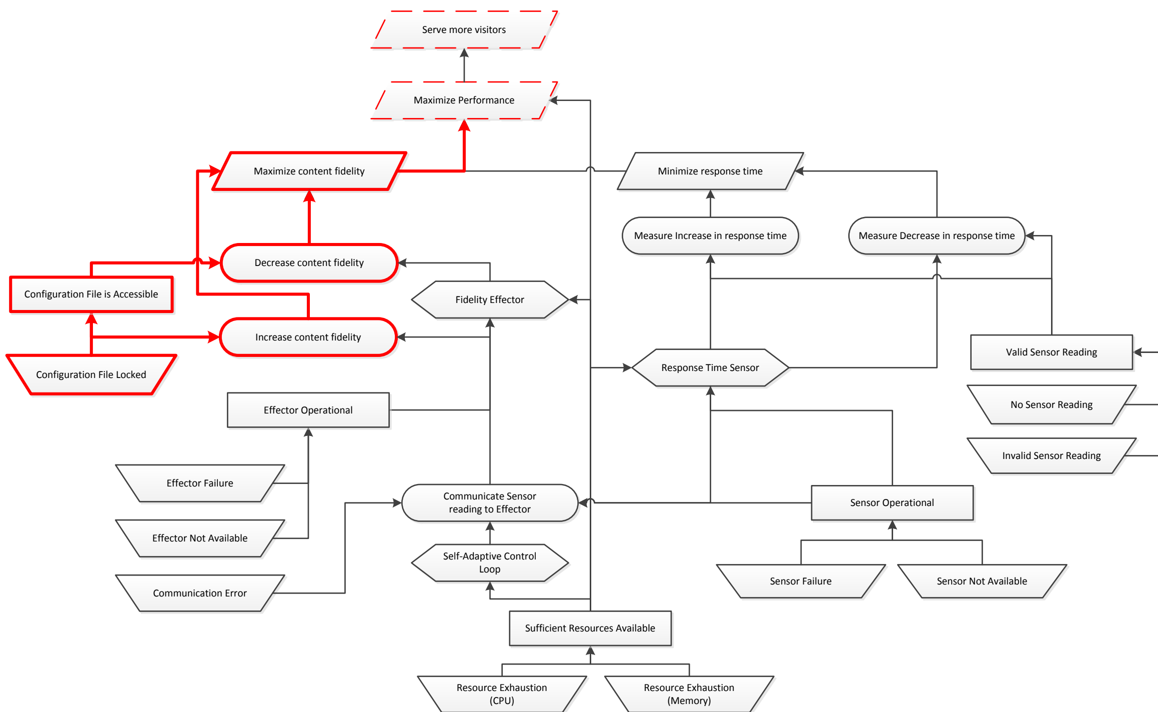

Figure 15: Goal Refinement Graph of Self-System A – Locked Configuration File Obstacle Impact

Step E: Definition of the Changeload

Defining the changeload, Step E, was conducted in the same manner as proposed by Almeida and Vieira (2012a) in which the most relevant obstacles were selected to include in the changeload by defining an exposure matrix and relevancy cut-off level. The goal of the exposure matrix was to prioritize obstacle relevance based on the previously defined obstacle attributes. The combination (i.e. their intersection) of the OB and OSDG attributes corresponded to the obstacle's relevance level in the same way the combination of impact and probability denoted relevance in the risk-based approach. The goal-oriented approach utilized the same relevance defined in the risk-based approach and described in Chapter 1.

Let Rel be the relevance scale for the current evaluation of the SUB, where a "negligible" relevance denoted an obstacle that can be overlooked and "mandatory" relevance denoted an obstacle of obligatory inclusion into the changeload, as defined in Equation 32 (Almeida & Vieira, 2012a).

$$\text{Rel} = \{\text{negligible, very low, low, high, very high, mandatory}\}$$

Equation 32: Definition of the Relevance Scale

Mapping the relevance levels to numeric values provided a method for further automation of the approach by making mathematical comparisons straightforward, as shown in Table 12. The relevance levels were mapped to ascending integers, such as from one (less relevant) to six (most relevant).

Relevance Level	Value
Negligible	1
Very Low	2
Low	3
High	4
Very High	5
Mandatory	6

Table 12: Relevance Level Numeric Mapping

Finally, the exposure matrix was populated as recommended in the risk-based approach, with the OB and OSDG attributes on the axes and relevance levels as their intersection, as shown in Table 13. Table 14 shows the exposure matrix for example Self-System A. The obstacles were only included within the exposure matrix's relevance levels to illustrate their assignment and would not be done in practice.

		OB			
		Very High	**High**	**Low**	**Very Low**
OSDG	**Catastrophic**	Mandatory	Very High	High	Medium
	Critical	Very High	High	Medium	Low
	Marginal	High	Medium	Low	Very Low
	Negligible	Medium	Low	Very Low	Negligible

Table 13: Exposure Matrix for the Goal-Oriented Approach

		OB			
		Very High (4)	**High (3)**	**Low (2)**	**Very Low (1)**
OSDG	**Catastrophic (1 and 2)**	**Mandatory** Resource exhaustion (CPU) Resource exhaustion (Memory)	Very High	High	Medium
	Critical (3)	Very High	**High** Configuration file locked No sensor reading Invalid sensor reading	Medium	Low
	Marginal (4)	**High** Sensor failure Sensor not available	**Medium** Effector failure Effector not available Communication error	Low	Very Low
	Negligible (5 and 6)	Medium	Low	Very Low	Negligible

Table 14: Exposure Matrix for example Self-System A

A relevance cut-off level was then defined in an effort to include only those obstacles deemed relevant to the current evaluation (Almeida & Vieira, 2012a).

Let the defined relevance cut-off level, RCL, be an element in the set of possible relevance levels, Rel, where RCL defines the minimum level of relevance of included obstacles within the changeload, as defined in Equation 33.

$$RCL = \{x \mid x \in \mathrm{Rel}\}$$

Equation 33: Definition of the Relevance Cut-Off

The risk-based approach recommended an RCL of at least "mandatory", however, this study utilized an RCL of "high" to ensure test coverage.

Table 15 shows the previously defined exposure matrix with the relevance cut-off level applied, while Table 16 demonstrates the exposure matrix with the cut-off level applied for the example Self-System A. The obstacles were only included within the exposure matrix's relevance levels to illustrate their assignment and would not be done in practice.

		OB			
		Very High	**High**	**Low**	**Very Low**
OSDG	**Catastrophic**	Mandatory	Very High	High	Medium
	Critical	Very High	High	Medium	Low
	Marginal	High	Medium	Low	Very Low
	Negligible	Medium	Low	Very Low	Negligible

Table 15: Exposure Matrix with Cut-Off Level Applied

		OB			
		Very High (4)	**High (3)**	**Low (2)**	**Very Low (1)**
OSDG	**Catastrophic (1 and 2)**	**Mandatory** Resource exhaustion (CPU) Resource exhaustion (Memory)	Very High	High	Medium
	Critical (3)	Very High	**High** Configuration file locked No sensor reading Invalid sensor reading	Medium	Low
	Marginal (4)	**High** Sensor failure Sensor not available	**Medium** Effector failure Effector not available Communication error	Low	Very Low
	Negligible (5 and 6)	Medium	Low	Very Low	Negligible

Table 16: Exposure Matrix with Cut-Off Level Applied for example Self-System A

The changeload was then defined as the set of enumerated changes, $EC_{goal-oriented}$, which contained obstacles whose relevance met or exceeded the defined relevance cut-off level, RCL, as depicted in Equation 34. The changeload definition for example Self-System A is shown in Equation 35 with a cut-off level of "high".

$$EC_{goal-oriented}\left(RCL\right) = \{o \mid o \in O,\, o_{relevance} \geq RCL\}$$

Equation 34: Changeload Definition

$$EC_{goal-oriented}\left(RCL := \text{high}\right) = \{o \mid o \in O,\, o_{relevance} \geq \text{high}\}$$
$$= \left\{o \mid o \in O,\, o_{relevance} \in \{\text{high, very high, mandatory}\}\right\}$$

Equation 35: Changeload Definition for example Self-System A

The changeload corresponded to a minimized subset of the system's entire change space, whereby only those obstacles of high or greater relevance were included (bold outline), as illustrated in Figure 16. The excluded obstacles are indicated with a dotted outline.

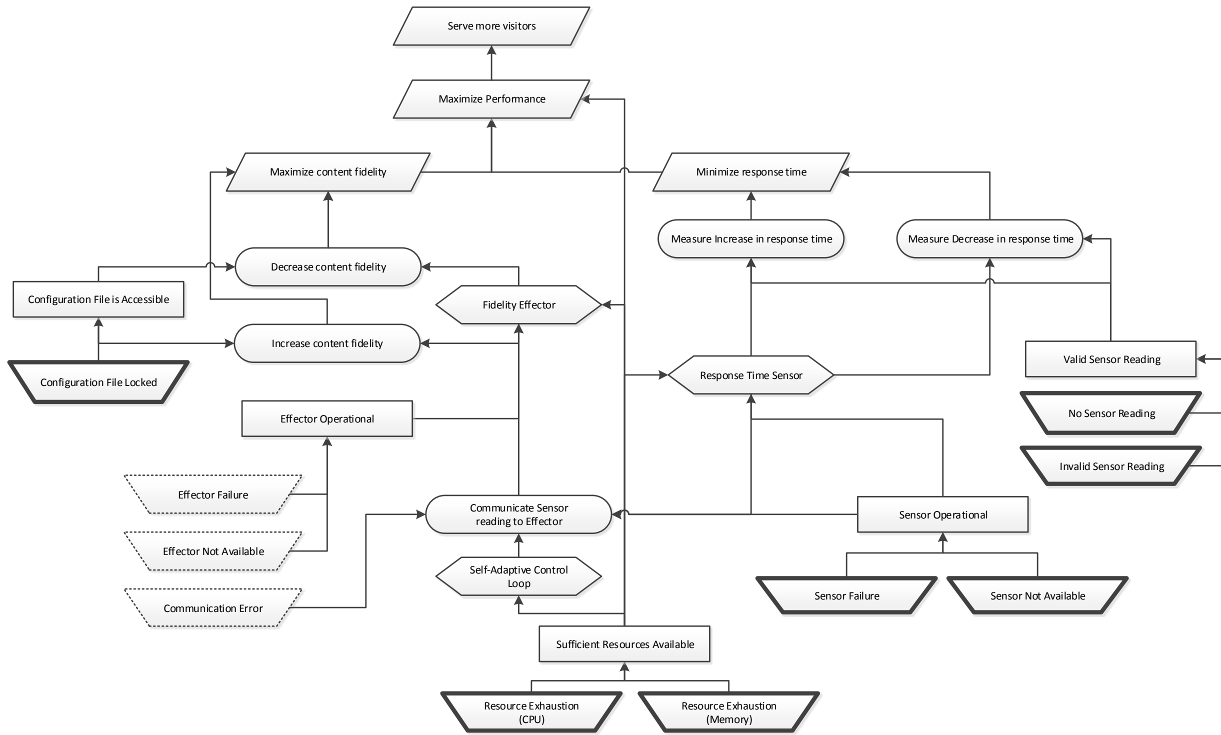

Figure 16: Considered Obstacles for example Self-System A

The obstacles were translated into concrete changes only after the definition of the changeload, as depicted in Table 17. This is in contrast to the risk-based approach, where concrete changes were created for each identified change scenario prior to the cut-off being applied, which resulted in wasted effort and increased costs. Table 18 shows an example of the concrete obstacles within the defined changeload for example Self-System A.

Obstacle	Target	Target Type	Trigger Instant	Duration	Amount	OSDG	OB	Relevance
			ms	ms	%			
			ms	ms	%			

Table 17: Concrete Obstacles in the final Changeload generated by the Goal-Oriented Approach

Obstacle	Target	Target Type	Trigger Instant	Duration	Amount	OSDG	OB	Relevance
Configuration File Locked	Increase / Decrease Content Fidelity	Action	15s	120s	100%	Critical	High	High
No Sensor Reading	Measure Increase / Decrease in Response	Action	60s, 120s, 180s	30s	100%	Critical	High	High
Invalid Sensor Reading	Measure Increase / Decrease in Response Time	Action	100s, 200s, 300s	5s	100%	Critical	High	High
Sensor failure	Response Time Sensor	Agent	500s	60s	100%	Marginal	Very High	High
Sensor not available	Response Time Sensor	Agent	475s	15s	100%	Marginal	Very High	High
Resource Exhaustion (CPU)	Maximize Performance, Self-Adaptive Control Loop, Response Time Sensor, Fidelity Effector	Goal, Agent, Agent, Agent	600s, 700s, 800s	10s, 30s, 90s	75%, 90%, 100%	Catastrophic	Very High	Mandatory
Resource Exhaustion (Memory)	Maximize Performance, Self-Adaptive Control Loop, Response Time Sensor, Fidelity Effector	Goal, Agent, Agent, Agent	700s, 800s, 900s	10s, 30s, 90s	75%, 90%, 100%	Catastrophic	Very High	Mandatory

Table 18: Final Changeload with Concrete Obstacles for example Self-System A

Case Study

A case study was conducted to determine the cost-effectiveness of the goal-oriented approach over the risk-based approach. In Almeida and Vieira (2012a), the authors conducted a case study of a fictitious ADBMS to demonstrate the effectiveness of the risk-based approach to define a suitable changeload. However, they did not provide comprehensive documentation for each step, including those related to discovery, identification, and analysis. To the best of the

author's knowledge, no comprehensive case study utilizing the risk-based approach and focusing on overall costs existed within literature.

Therefore, this research conducted a case study applying the risk-based and goal-oriented approaches against the same subject system. The data from each approach was recorded and compared as described in the following section

Subject System

The ZNN.com system is an N-tier web-based information system designed to reproduce the real-world systems utilized in large-scale online news providers, such as CNN.com. It was built on RAINBOW, an architecture-based platform for self-adaptation, and focused on meeting QoS goals while minimizing server costs (Cámara, Lemos, Vieira, et al., 2013; S. W. Cheng et al., 2009). The RAINBOW framework provided reusable, generic, and cost-effective mechanisms to implement the self-adaptive control loop, the MAPE loop, which monitored the target system, detected changes, planned how to adapt, and executed the adaptation in response to the changes (S. W. Cheng et al., 2009). The RAINBOW framework is depicted below in Figure 17.

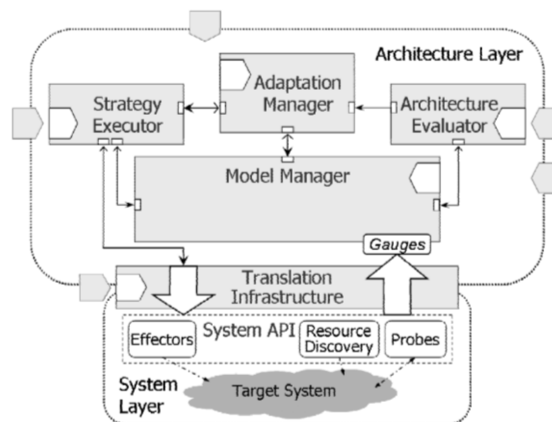

Figure 17: RAINBOW Framework

The ZNN.com system's N-tier architecture consisted of a set of application servers that served web content, such as images, videos, and text, from back-end database servers to clients (c0 – c2) via front-end presentation logic, as shown in Figure 18. It utilized a load balancer (lbproxy) to distribute incoming requests across servers (s0 – s3) based on their utilization.

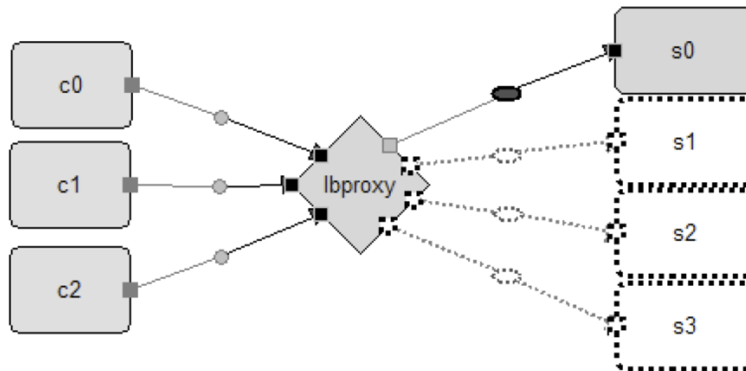

Figure 18: ZNN.com System Architecture

The system's runtime goals were to prevent the loss of customers due to poor performance by reducing content fidelity during peak times. Thus, its high-level goals consisted of performance, cost, and content fidelity, similar to the example utilized throughout this document. The case study analyzed documentation presented in S. W. Cheng, Huang, Garlan, Schmarl, and Steenkiste (2004), S. W. Cheng et al. (2009), and Cámara, Lemos, Vieira, et al. (2013), to determine the characteristics of the ZNN.com system to avoid the need for a physical implementation.

Analysis of Results

The study's results were analyzed to determine the cost-savings provided by the goal-oriented approach over the risk-based approach and to compare the characteristics of the resulting changeloads. Cost savings was determined by utilizing the *Simplified Test Suite Cost*

Comparison Inequality (rewritten) in *Equation 15*. However, further reductions to the inequality were possible based on the values obtained through the case studies.

The cost of a test selection strategy, s and s', which included the costs of personnel, equipment, and resources, the cost of executing a single unattended test against the SUB, e, and the cost of comparing a test's output against the system's specification to analyze its result, c, were the same for both approaches and are constant. Therefore, the inequality was further reduced with the removal of all constants as shown in Equation 37.

$$\frac{|T_s'| + |T'|}{|T_s| + |T|} < 1$$

Equation 36: Reduced Test Suite Cost Inequality

The total number of tests considered throughout the risk-based approach, represented by $|T_s|$, the total number of tests included in the final risk-based approach changeload, $|T|$,, the total number of tests considered throughout the goal-oriented approach, $|T_s'|$, and the total number of tests included in the final risk-based approach changeload, $|T'|$, correspond to the cost of each approach. The goal-oriented approach provided a cost-savings over the risk-based approach if the inequality held true. The value of the ratio (the left side of the inequality) indicated the relative cost savings experienced from the utilization of the goal-oriented approach.

The resulting changeloads were compared to determine the effectiveness of the goal-oriented approach. The number of identified changes for each included relevance level was used to determine the goal-oriented approach's comprehensiveness. A greater distribution of highly relevant changes denoted greater changeload relevance.

The degree to which the changeloads were reduced by the application of the relevance cut-off level was used to determine the wastefulness of the approach by identifying the number of irrelevant changes identified.

The overall effectiveness of the approach was determined by calculating the return on investment for each selection strategy, ROI_s, defined as the quotient of the total number of changes with relevance level of at least "high" identified by the strategy, $|T(RCL := high)|$, and the total number of tests identified by the test selection strategy, $|T_s|$, as shown in Equation 37. A larger ROI_s value implied a greater return and effectiveness of the selection strategy.

$$ROI_s = \frac{|T(RCL := high)|}{|T_s|}$$

Equation 37: Test Selection Strategy's Return on Investment

Summary

This research extended the risk-based approach proposed by Almeida and Vieira (2012a) by incorporating goal-oriented requirements engineering techniques developed by Dardenne et al. (1993). A case study approach was be used to demonstrate the validity and effectiveness of the goal-oriented approach over the risk-based approach, where a target system was analyzed using both approaches and their results compared. This allowed direct comparison of the approaches and enabled future studies to utilize the methodology and results. The results of the case study are be presented in tabular and graphical format to allow direct comparison of their data, discussed in the next section. The hypothesized outcome was the integration and utilization of goal-oriented requirements engineering techniques to analyze the system would result in fewer

test cases being defined and executed for a given target system resulting in lower resilience benchmarking costs of self-adaptive systems.

The following section presents the data produced by the case study, the case study's results, and their analysis.

Chapter 4

Results

The results of the case study demonstrated that the goal-oriented approach minimized the test suite and resulting changeload for the subject system, successfully reducing the cost of resilience benchmarking of self-adaptive systems by over 80%. The case study's produced data is presented in the next section, followed by the presentation of the study's results and their analysis.

Presentation of Data

The following section presents the data produced by the risk-based approach, followed by the data produced by the goal-oriented approach.

Risk-Based Approach Data

The base scenario defined in Step A of the risk-based approach is presented below in Table 19. The high-level goals, operating conditions, and base line workload are taken from the ZNN.com specification (V.-W. Cheng, 2008).

Step A: Identification of the Base Scenario

Goals	Operating Conditions	Workload
Serve news content (content quality) Reasonable response time range (performance) Within operating budget (cost)	Adequate resources	Normal request traffic

Table 19: Risk-Based Approach Base Scenario Definition Data

Step B: Identification of Change Scenarios

The data produced in Step B of the risk-based approach is shown in Table 20 and Table 21 Table 21 only contains a sample of the data produced, and the concrete change details (i.e.

trigger instant, duration, and amount) were omitted, as there were a large number of identified changes. The full list of identified changes can be found in Appendix B.

		Base Scenario Elements		
		Goals	**Operating Conditions**	**Workload**
Sources of Change	Target System (ZNN.com N-tier system)		Internal node connection faults Gauge Issues Adaptive Overhead Effector Issues Configuration	
	Resources (Hardware)		Fluctuations in server resources Fluctuations in network performance New HW Fluctuations in Load Balancer Performance and Availability Backup Issues Faulty HW	
	Resources (Software)		OS Faults File System Faults Fluctuations in service availability OS Updates	
	Environment		Operator Errors Power availability Attack	Fluctuations in request type Fluctuation in number of requests Fluctuation in number of users Content stealing

Table 20: Risk-Based Approach Change Class and High-Level Change Mapping to Base Scenario Elements Data

Specific Change	Class	Impact	Probability	Relevance
Unable to communicate with Server (1… n)	Internal node connection faults	Catastrophic	High	Very High
Unable to communicate x n		Catastrophic	High	Very High
Communication Failure: Server to Load Balancer		Marginal	Low	Medium
Communication Timeout: Server to Load Balancer		Negligible	High	Low
Communication Corruption		Negligible	Very Low	Negligible
Network link saturation		Marginal	High	Medium
Link congestion: Load Balancer to Servers		Marginal	Very High	High
Communication Delay: Load Balancer to Servers		Marginal	Very High	High
Unable to turn server on (stuck off)	Effector	Catastrophic	Very High	Mandatory
Unable to turn server off (stuck on)		Critical	Low	Medium
Unable to reduce content fidelity (stuck high)		Catastrophic	Very Low	Medium
Unable to increase content fidelity (stuck low)		Critical	Very Low	Medium
Unable to increase content fidelity (stuck medium)		Marginal	Very Low	Low
Unable to decrease content fidelity (stuck medium)		Marginal	Very Low	Low
…	…	…	…	…

Table 21: Risk-Based Approach Change Scenario Definitions Sample Data

Goal-Oriented Approach Data

The following section presents the data generated by the goal-oriented approach.

Step A: Identification of System Goals

The initial goal refinement graph produced in Step A of the goal-oriented approach is shown in Figure 19. It is composed of six refined goals and their relationships.

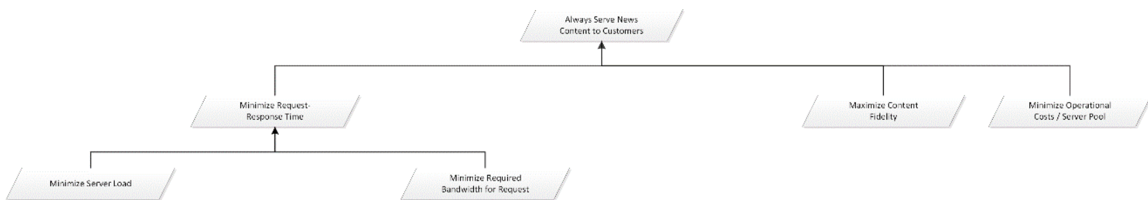

Figure 19: Goal-Oriented Approach Goal Refinement Graph Data

Step B: Identification of Obstacles

The expanded goal refinement graph produced in Step B of the goal-oriented approach is depicted in Figure 20. It contains all identified goals, actions, agents, assumptions, and obstacles. Table 22 contains a summary of the expanded goal refinement graph illustrated in Figure 20, allowing for a straightforward analysis of its composition.

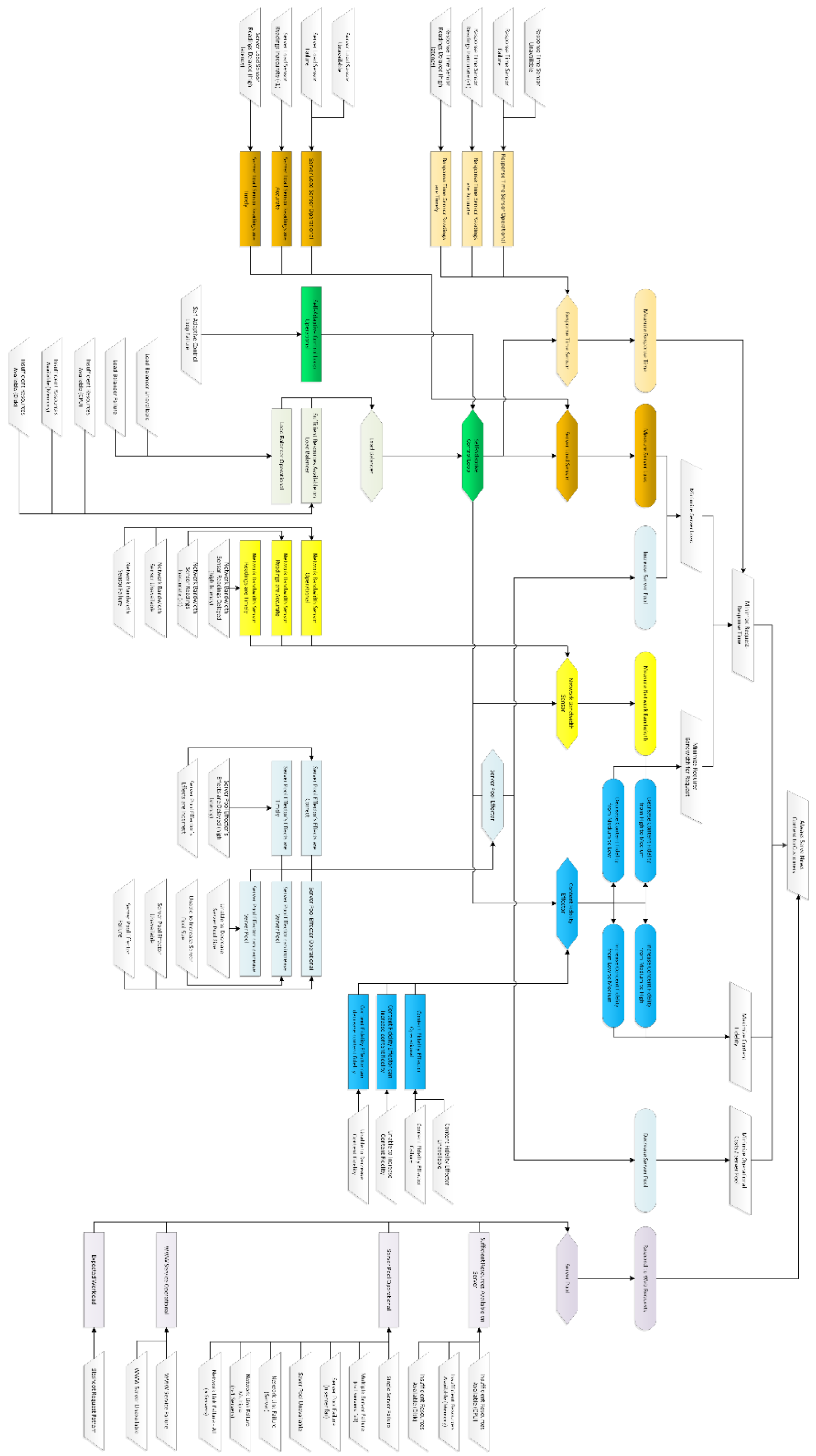

Figure 20: Goal-Oriented Approach Expanded Goal Refinement Graph with Obstacles, Assumptions, Agents, and Actions Data

Expanded Goal Refinement Graph Composition	
Total Number of Goal Nodes	6
Total Number of Actions Nodes	10
Total Number of Assumptions Nodes	24
Total Number of Obstacles Nodes	41
Max Distance (Obstacle to Goal)	8
Min Distance (Obstacle to Goal)	4

Table 22: Expanded Goal Refinement Graph Composition Summary Data

Step C: Definition of Obstacle Attributes

Step C of the goal-oriented approach produced the definition of the OSDG and OB obstacle attributes, as well as their associated and effective ranges, as shown in Table 23 and Table 24.

Risk-Based Impact Attribute	Goal-Oriented OSDG Attribute	Effective Range
Catastrophic	[1, 4]	1, 2, 3, and 4
Critical	(4, 5.3]	5
Marginal	(5.3, 6.7]	6
Negligible	(6.7, 8]	7 and 8

Table 23: Goal-Oriented Approach OSDG Attribute Data

Risk-Based Impact Attribute	Goal-Oriented OB Attribute Range	Effective Range
Very High	[6, 4.5)	5 and 6
High	[4.5, 3)	4
Low	[3, 1.5)	2 and 3
Very Low	[1.5, 0)	1

Table 24: Goal-Oriented Approach OB Attribute Data

Step D: Evaluation of Obstacle Attributes and Step E: Definition of the Changeload

Table 25 shows the test suite produced by the goal-oriented approach with their associated obstacle attributes. Note that the trigger instant, duration, and amount of each obstacle were omitted for ease of review.

Obstacle	Target	Target Type	OSDG	OB	Relevance
Response Time Sensor Unavailable	Response Time Sensor	Agent, Assumption	Catastrophic (4)	Low (2)	High
Response Time Sensor Failure			Catastrophic (4)	Low (2)	High
Response Time Sensor Readings Inaccurate (-1)			Catastrophic (4)	Low (2)	High
Response Time Sensor Readings Delayed (high latency)			Catastrophic (4)	Low (2)	High
Server Load Sensor Unavailable	Server Load Sensor	Agent, Assumption	Catastrophic (4)	Low (3)	High
Server Load Sensor Failure			Catastrophic (4)	Low (3)	High
Server Load Sensor Readings Inaccurate (-1)			Catastrophic (4)	Low (3)	High
Server Load Sensor Readings Delayed (high latency)			Catastrophic (4)	Low (3)	High
Self-Adaptive Control Loop Failure	Self-Adaptive Control	Agent, Assumption	Critical (5)	Very High (6)	Very High
Insufficient Resources Available (CPU)	Load Balancer	Agent, Assumption	Marginal (6)	Very High (6)	High
Insufficient Resources Available (Memory)			Marginal (6)	Very High (6)	High
Insufficient Resources Available (Disk)			Marginal (6)	Very High (6)	High
Load Balancer Unavailable			Marginal (6)	Very High (6)	High
Load Balancer Failure			Marginal (6)	Very High (6)	High
Network Bandwidth Sensor Unavailable	Network Bandwidth Sensor	Agent, Assumption	Catastrophic (4)	Low (3)	High
Network Bandwidth Sensor Failure			Catastrophic (4)	Low (3)	High
Network Bandwidth Sensor Readings Inaccurate (-1)			Catastrophic (4)	Low (3)	High
Network Bandwidth Sensor Readings Delayed (high latency)			Catastrophic (4)	Low (3)	High
Server Pool Effector's Effects are Incorrect	Server Pool Effector	Agent, Assumption	Catastrophic (4)	High (4)	Very High

Server Pool Effector's Effects are Delayed (high latency)			Catastrophic (4)	High (4)	Very High
Server Pool Effector Unavailable			Catastrophic (4)	High (4)	Very High
Server Pool Effector Failure			Catastrophic (4)	High (4)	Very High
Unable to Decrease Server Pool Size			Catastrophic (4)	High (4)	Very High
Unable to Increase Server Pool Size			Catastrophic (4)	High (4)	Very High
Content Fidelity Effector Unavailable	Content Fidelity Effector	Agent, Assumption	Catastrophic (4)	High (4)	Very High
Content Fidelity Effector Failure			Catastrophic (4)	High (4)	Very High
Unable to Increase Content Fidelity			Catastrophic (4)	High (4)	Very High
Unable to Decrease Content Fidelity			Catastrophic (4)	High (4)	Very High
Insufficient Resources Available (CPU)	Server Pool	Agent, Assumption	Catastrophic (4)	Very Low (1)	Medium
Insufficient Resources Available (Memory)			Catastrophic (4)	Very Low (1)	Medium
Insufficient Resources Available (Disk)			Catastrophic (4)	Very Low (1)	Medium
Sever Pool Unavailable			Catastrophic (4)	Very Low (1)	Medium
Single Server Failure			Catastrophic (4)	Very Low (1)	Medium
Multiple Server Failure (n-1 servers fail)			Catastrophic (4)	Very Low (1)	Medium
Server Pool Failure (n server fail)			Catastrophic (4)	Very Low (1)	Medium
WWW Service Failure			Catastrophic (4)	Very Low (1)	Medium
WWW Server Unavailable			Catastrophic (4)	Very Low (1)	Medium
Network Link Failure (Server)			Catastrophic (4)	Very Low (1)	Medium
Network Link Failure – Multiple (n-1 Servers)			Catastrophic (4)	Very Low (1)	Medium
Network Link Failure – All (n Servers)			Catastrophic (4)	Very Low (1)	Medium
Slashdot Request Pattern			Catastrophic (4)	Very Low (1)	Medium

Table 25: Goal-Oriented Approach Final Changeload with Concrete Obstacles Results

Presentation of Results

The following section presents the case study results. Table 26 shows the number of identified changes utilizing the risk-based and goal-oriented approaches and includes the numeric and percent difference for each relevance level. Table 27 shows the number of included changes for each relevance level and the final changeload size produced by each approach.

		Risk-Based Approach	Goal-Oriented Approach	Difference	Percent Difference
1	**Negligible**	4	0	-4	-100%
2	**Very Low**	9	0	-9	-100%
3	**Low**	14	0	-14	-100%
4	**Medium**	138	13	-125	-91%
5	**High**	43	17	-26	-60%
6	**Very High**	35	11	-24	-69%
7	**Mandatory**	9	0	-35	-100%
8	**Total Test Suite Size**	**252**	**41**	**-211**	**-84%**

Table 26: Test Suite Construction and Total Size Comparison Results

		Risk-Based Approach	Goal-Oriented Approach	Difference	Percent Difference
1	**Negligible**	0	0	-	-
2	**Very Low**	0	0	-	-
3	**Low**	0	0	-	-
4	**Medium**	0	0	-	-
5	**High**	43	17	-26	-60%
6	**Very High**	35	11	-24	-69%
7	**Mandatory**	9	0	-9	-100%
8	**Final Changeload Size**	**87**	**28**	**-59**	**-68%**

Table 27: Included Change Scenarios and Final Changeload Size Comparison after Cut-Off Results

Results Analysis

The following section analyzes the results of the case study presented in the previous section based on the qualities outlined in the Analysis of Results section to determine the relative cost savings, effectiveness, wastefulness, and return on investment of the goal-oriented approach.

Cost Savings

Equation 36 was utilized to determine overall cost-savings of the goal-oriented approach and utilized the results presented in Table 26 and Table 27.

$$\frac{41+28}{252+87}<1 \quad \therefore \quad 0.2035<1$$

Equation 38: Cost Savings Inequality Results

The resulting inequality, shown in Equation 38, held true and indicated that the goal-oriented approach provided cost savings over the risk-based approach. The calculated value quantified the extent of the cost savings, where the ratio signified the overall cost of the goal-oriented approach being 20.35% of the overall cost of the risk-based approach. Said differently, the goal-oriented approach reduced the cost of resilience benchmarking by 79.65%. Even if the full goal-oriented test suite were utilized in an effort to ensure maximum test coverage and comprehensiveness of evaluation, the approach would still provide a cost savings of 75.81% over the risk-based approach.

The cost savings was achieved by reducing the number of identified and enumerated changes against the subject system. For example, the risk-based approach's use of a high-level base scenario definition resulted in a large number of workload pattern variations that needed to be defined for the workload, disk utilization, network congestion, and resource utilization to fully evaluate the system on any changes to these aspects. They included steady state, sinusoidal,

stepwise, ramp, exponential, and random request / utilization patterns for the subject system's major components: web server pool CPU, memory, and disk utilization; load balancer CPU, memory, and disk utilization; the internal network's bandwidth and latency patterns; and the web client workload's request type variation and request timing patterns. They totaled seventy distinct changes and constituted 27.78% of the risk-based approach's test suite. However, all of the request changes were found to be irrelevant to the SUB's evaluation, and omitted from the final changeload, since none of them met the high relevancy requirement.

Another example is changes affecting traditional agents, such as faulty hardware and operator error, were not considered in the goal-oriented approach since a self-adaptive agent was not responsible for ensuring their resilience to runtime changes. This contrasts the risk-based approach, which considered runtime changes to all aspects of the system, such as eight faulty hardware changes, six general security changes, eleven common administrative user errors, eight operating system faults, and four electrical system changes. These changes accounted for 14.68% of the risk-based test suite while 62.16% of those defined were omitted from the final risk-based changeload due to low relevance.

Effectiveness

The relevance distribution for each test suite was derived from Table 26 and is presented graphically in Figure 21. Table 28 provides a summary of the test suite distribution relative to the RCL.

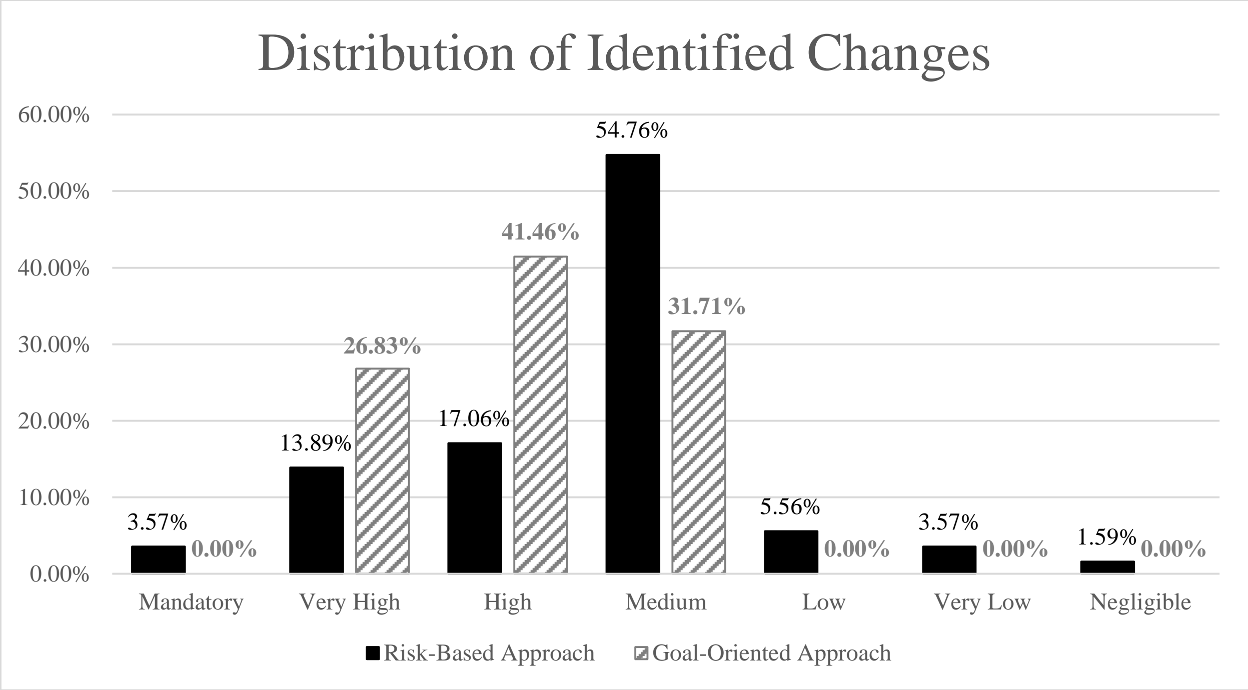

Figure 21: Test Suite Relevance Distribution of Identified Changes in the Resulting Test Suites

Relevance Distribution	Risk-Based Approach	Goal-Oriented Approach
< High	65.48%	31.71%
≥ High	34.52%	68.29%

Table 28: Test Suite Relevance Distribution Summary

The majority of changes identified by the risk-based approach had a relevance level of medium, which comprised 54.76% of the test suite. The test suite also contained 5.56% low, 3.57% very low, and 1.59% negligibly relevant changes. The majority of changes identified by the goal-oriented approach had a relevance level of high, which comprised 41.46% of the test suite. The test suite also contained 31.71% changes of medium relevance and zero low, very low, and negligibly relevant changes.

The results showed that the goal-oriented approach was effective at producing a relevant test suite for the subject system as its resulting test suite was composed of only 31.71% irrelevant

changes and 68.29% relevant changes. This was in contrast to the risk-based test suite was composed of 65.48% irrelevant changes and 34.52% relevant changes.

Examples of irrelevant changes identified by the risk-based approach were power supply failure, operating system updates, and malicious attacks. While the possibility of these changes occurring and ultimately diminishing the system's ability to achieve its goals exists, they did not meet the relevance requirement of the resilience evaluation and therefore provided little value in their consideration. These types of changes are more appropriately evaluated using dependability and security benchmarking as they do not typically consider self-adaptive mechanisms (Almeida & Vieira, 2012a; A. B. Brown et al., 2004; Meyer, 2009).

Wastefulness

The wastefulness of the approach was defined as the ratio of discarded changes to the total number of defined changes. The data was extracted from Table 28, where the risk-based and goal-oriented approaches discarded approximately 65.48% and 37.71% of their defined test suite after the RCL was applied, respectively.

The results indicated that the goal-oriented approach was less wasteful than the risk-based approach since a greater percentage of the identified changes met or exceeded the relevance requirement and were included in the final changeload. Avoiding the wasted effort from the identification, definition, and enumeration of irrelevant changes is a straightforward method of reducing benchmarking costs (Barbosa et al., 2005). In this instance, the goal-oriented approach significantly reduced wasted effort by reducing the amount of irrelevant changes that would ultimately be discarded by the RCL.

Return on Investment

The return on investment of each approach was calculated utilizing Equation 37 and populated with the results presented in Table 26 and Table 27.

The return on investment of the risk-based approach, $ROI_{risk-based}$, was calculated to be 0.3452. This value signified a return of approximately one relevant change for every three changes identified by the risk-oriented approach and corresponded to the roughly 65% wastefulness factor calculated in the previous section. The return on investment of the goal-oriented approach, $ROI_{goal-oriented}$, was calculated to be 0.6829. This value signified a return of approximately two relevant change for every three changes identified by the goal-oriented approach, and correlated to the approximate 32% wastefulness factor of the approach.

The higher return on investment, combined with the lower wastefulness factors, provide a clear picture of goal-oriented approach's value in reducing the cost of resilience benchmarking over the risk-based approach.

Summary

The goal-oriented approach was shown to effectively reduce the cost of defining a resilience changeload for self-adaptive systems. The approach utilized system knowledge to identify the subject system's self-adaptive agents, their operational assumptions, and the obstacles that would hinder the system's ability to attain runtime goals. The results of the case study showed the goal-oriented approach to provide a cost savings by being less wasteful and more effective at defining relevant changeload, thereby providing a greater return on invested effort when compared to the risk-based approach on the same subject system.

Chapter 5

Conclusions

This dissertation demonstrated that the goal-oriented approach for defining resilience changeloads is an effective method for reducing the overall cost of resilience benchmarking of self-adaptive systems over existing approaches. A comparative case study showed that utilizing knowledge of the system's goals and self-adaptive mechanisms is an effective method for identifying relevant runtime changes while simultaneously reducing the overall costs of resilience benchmarking. The incorporation of goal-oriented requirements engineering techniques to extract the pertinent system information from the SUB provided sufficient guidance to avoid the issues associated with existing methods, specifically, the identification of irrelevant and redundant changes.

Incorporating test suite minimization techniques at the onset of benchmarking activities greatly reduces the overall cost and effort required to carry out resilience evaluation, especially for large and complex systems. The cost reduction increases the likelihood of comprehensive verification of runtime behavior and the validation of system capabilities and resilience expectations in dynamic environments. This increases trust in the system and its services, which is especially important due to society's growing reliance on self-adaptive systems for infrastructure and critical services.

The goal-oriented approach entails analyzing, refining, and relating the self-adaptive system's goals in a goal refinement graph to reveal its runtime goals and behavior. The system is further analyzed to incorporate self-adaptive responses (i.e. runtime actions) and their responsible self-adaptive agents into the graph to identify the system components requiring direct

assessment due to their resilience responsibilities. Runtime assumptions are then enumerated for each self-adaptive agent to capture their expected operational and environmental conditions. The test suite is then produced by enumerating all unfavorable runtime conditions, or obstacles, that would contradict an assumption, directly affect a self-adaptive response, or mechanism, and obstruct the attainment of runtime goals. The goal-oriented requirements engineering techniques utilized within the approach were able to extract significant system knowledge that provides guidance for runtime change identification, providing cost savings over existing approaches.

The primary goal of designing an approach that reduces overall benchmarking costs while ensuring test coverage over past work was displayed through the results presented in Chapter 4.

The goal-oriented approach demonstrated greater cost effectiveness than the risk-based approach (Almeida & Vieira, 2012a) by producing a minimized test suite for the subject system and reducing the cost of resilience benchmarking by 79.65%. The goal-oriented approach also achieved a greater degree of return on investment by producing a more favorable relevant to irrelevant change ratio by a factor of two. Additionally, the goal-oriented approach reduced wasted effort and shown to be more effective at identifying highly relevant changes, both by a factor of two. The results demonstrated that the goal-oriented approach is effective in defining a relevant resilience changeload while reducing overall costs by minimizing the total number of identified test cases in the test suite and the number of enumerated changes in the changeload.

Implications

The problem of defining a changeload for the resilience benchmarking of self-adaptive systems has been addressed by previous work (Almeida & Vieira, 2012a) but resulted in extremely large test suites and high costs (Barbosa et al., 2005; Pressman, 2005; Vieira &

Madeira, 2004). The high cost of benchmarking often forced practitioners to omit comprehensive resilience evaluation as a cost-savings strategy since testing and maintenance costs often accounted for up to 80% of total system cost (Jorgensen, 2002). Previous benchmark cost saving techniques focus on minimizing the test suite by removing redundant and irrelevant test cases but they require exhaustive test suites be defined first. This study used system knowledge to guide the definition of test cases and avoided the definition and enumeration of irrelevant test cases by incorporating goal-oriented requirements engineering analysis techniques. The case study showed that the approach was effective at reducing the overall cost of resilience benchmarking while ensuring a high degree of changeload relevance. Refinements to this approach presents the potential for further cost savings while ensuring the relevance of the resulting changeload by further reducing the number of identified irrelevant changes.

Recommendations

The goal-oriented approach was developed in order to demonstrate the ability of system knowledge to reduce resilience benchmarking costs. While the approach was effective in this regard, it has several opportunities for improvement.

First, the definition of the relevance cut-off level (RCL) mirrored the risk-based approach to facilitate result comparison. Refinement of the RCL definition process may result in a cut-off level that is more appropriate to the SUB and its expected operational constraints (Almeida & Vieira, 2012a). For instance, an RCL of high may be too constraining for a military system that may require evaluation that is more comprehensive.

Additionally, refinement to the obstacle attributes, and their associated thresholds, may result in change relevance assignments that are more suitable to a SUB than those used within this study. The total number of attribute values, four for both the OSDG and OB, and the six

change relevance levels, mirrored those utilized in the risk-based approach. Refinement to the attributes and relevance levels may increase their applicability, appropriateness, and expressiveness for other SUBs.

Finally, extension of the goal refinement graph to include additional dimensions of system knowledge may provide additional insight into the system's runtime behavior and should be investigated. For example, additional graph theory analysis techniques, such as node failure modeling (Heegaard & Trivedi, 2009), may provide further insight into an obstacle impact and provide a more appropriate quantification method. Further, goal priorities or weights may provide a more effective method of evaluating obstacle relevance, failure propagation, and perceived failure qualities (Quadri & Farooq, 2010). The incorporation of runtime simulations, documentation review, or adaptive modeling may provide guidance into the evaluation of adaptive strategy and runtime behavior since a system may respond differently to the same changes in a different sequence (Almeida & Vieira, 2011; Andersson et al., 2009; Madan, Goševa-Popstojanova, Vaidyanathan, & Trivedi, 2004). Further, source code analysis may also be useful to determine specific adaptive mechanisms and capabilities, providing greater insight into functionality requiring evaluation and component-specific runtime obstacles that would otherwise go unidentified (Barbosa et al., 2005).

Summary

Society's reliance on software systems to provide mission critical and infrastructure services continues to increase (IBM, 2003). The systems must continue to operate as expected especially when unfavorable or unexpected situations arise, such as attack, power outage, and failure (Almeida & Vieira, 2012a; Huebscher & McCann, 2004; IBM, 2003). This has resulted in a continued increase in system complexity and scale to cope with society's growing

performance, redundancy, robustness, and data demands (IBM, 2003). The management and maintenance of these systems has grown increasingly costly and error prone due to the explosion in their growth and complexity (Ganek & Corbi, 2003), especially when coupled with the unpredictable workloads produced by society (IBM, 2003). The resulting service outages and disruptions negatively affected those reliant on their services with financial and societal consequences (Ganek & Corbi, 2003).

System designers incorporated self-adaptive mechanisms into systems in order to address the problem of ensuring the system's resilience to runtime changes and reducing the reliance on human operators to conduct complex management, configuration, and tuning tasks (Bondavalli et al., 2009; Group, 2002; IBM, 2003; Moorsel et al., 2009). These mechanisms increased a system's resilience to runtime changes and instilled it with dynamic runtime behavior which was able to respond to changes within its operational context with little or no human intervention (Almeida & Vieira, 2011; B. Cheng et al., 2009; IBM, 2003). Consequently, the self-adaptive systems required verification and validation of their runtime behavior in order to elicit a sufficient level of trust for their use in infrastructure and critical systems (A. Avizienis, J.-C. Laprie, B. Randell, & C. Landwehr, 2004; Kanoun et al., 2004). However, resilience evaluation of these systems was often overlooked or avoided (Quadri & Farooq, 2010) because the additional dimension of runtime variability caused the evaluation and verification of runtime requirements and goal attainment to be complex, labor intensive, and costly (Almeida & Vieira, 2012a; Bondavalli et al., 2009; A. B. Brown et al., 2004).

Existing techniques, such as the risk-based approach for defining resilience changeloads of self-adaptive systems, focused on identifying relevant risks that would result in failure to attain runtime goals (Almeida & Vieira, 2012a). The risk-based approach utilized extended of

Software Risk Evaluation (SRE) techniques and deductive reasoning to define a resilience changeload in a five-step process:

- **Step A – Identification of the Base Scenario:** The typical high-level goals, operating conditions, and workload were identified for the system-class.
- **Step B – Identification of Change Scenarios:** The potential sources of risks to the base scenario's high-level goals were identified, mapped to classes of changes, and then specific changes were defined that may directly affect the identified high-level goals.
- **Step C – Definition of Change Scenario Attributes:** Attributes were then defined to qualify the importance and priority of each defined change scenario.
- **Step D – Evaluation of the Change Scenario Attributes:** The defined change scenarios were then evaluated and assigned attributes using expert knowledge and multi-voting schemes. The combination of change scenario attributes corresponded to the change scenario's relevance to the system evaluation.
- **Step E – Definition of the Changeload:** The final changeload was then defined by defining the relevancy cut-off level, or RCL, to omit irrelevant change scenarios from the changeload.

Issues existed, however, as the approach directed the evaluator to consider a very large change space for the system under benchmark by treating the system goals and operating conditions in an abstract manner, resulting in high costs. The authors included a cost minimization technique, the RCL, to reduce the number of enumerated changes by removing irrelevant changes from the changeload. However, the approach resulted in a very large test suite that was labor intensive and costly to define and enumerate on complex self-adaptive systems (Almeida & Vieira, 2012a). The removal of irrelevant, repetitive, and redundant

changes from the test suite has been shown to successfully minimize the test suite and reduce benchmarking costs (Barbosa et al., 2005; Galeebathullah & C.P.Indumathi, 2010; Xavier et al., 2008), however, these techniques require an exhaustive test suite be defined first and then filtered, which resulted in additional labor and costs.

This dissertation was developed to incorporate the use of system knowledge to guide the identification of runtime changes to reduce the number of irrelevant, repetitive, and redundant changes. Its primary goal was to extend past work and develop an approach that reduced the overall costs of resilience benchmarking while maintaining changeload relevance. This dissertation developed a goal-oriented approach, which produced a minimized changeload that indicated it achieved this goal. The goal-oriented approach was developed by leveraging goal-oriented requirements engineering techniques (van Lamsweerde, 2000) to guide the analysis of self-adaptive systems to identify relevant runtime changes.

The basis of the goal-oriented approach is to extract detailed information of the system to identify its runtime goals, their underlying assumptions, and obstructing conditions for goal attainment. The approach consists of a five-step process:

- **Step A – Identification of System Goals:** HOW and WHY goal refinement techniques are used to iteratively refine the system's high-level goals to determine how high-level goals are attained (sub-goals) and why they exist (parent goals) to determine goal relationships and dependencies. A goal refinement graph is created to visualize their relationships using the KAOS specification.
- **Step B – Identification of Obstacles:**
    - **Part 1** consists of analyzing the system to determine the actions conducted to achieve each identified goal, the agent responsible for carrying out the actions,

and underlying assumptions that need to be true at runtime. These nodes are added to the goal refinement graph to provide further insight into the system and its behavior.

    - **Part 2** consists of analyzing the system and the goal refinement graph to identify the obstructing conditions under which goal attainment is unachievable. The obstacles are then incorporated into the goal refinement graph.

- **Step C – Definition of Obstacle Attributes:** Attributes are then defined using graph theory and characteristics of the goal refinement graph to quantify the importance of each obstacle.
- **Step D – Evaluation of Obstacle Attributes:** The defined obstacles are then assigned attributes based on their node characteristics in the graph to determine their relevance to the system evaluation.
- **Step E – Definition of the Changeload:** The final changeload is then defined by using an RCL to further minimize the test suite.

A comparative case study using the risk-based and goal-oriented approaches on the same subject system, ZNN.com (V.-W. Cheng, 2008), was conducted to gauge the approach's effectiveness to define a minimized changeload. The data produced by the approaches, as well as the final resilience changeload, were compared to determine the goal-oriented approach's relative cost savings, wastefulness, effectiveness, and return on investment over the risk-based approach. The results demonstrated that the goal-oriented approach successfully reduced the size of the test suite and final changeload providing an overall cost savings of 79.65% over the risk-based approach while effectively producing a test suite of higher relevance. Additionally,

the goal-oriented approach was shown to be less wasteful and provide a greater return on invested effort, both by a factor of two, over previous work.

This dissertation demonstrated that the utilization of system knowledge to guide the definition of a resilience changeload could result in significant cost savings while producing a highly relevant changeload. It provides a method of defining a cost effective resilience changeload that is widely applicable to address the resilience benchmarking needs of large and complex self-adaptive systems.

Appendix A

Leung and White (1991) proposed a testing cost model for the comparison of selective retesting versus retest-all strategies in regression testing, which was useful when comparing two testing strategies against the same system. The cost model defined the total cost of a software testing strategy, $C(Strategy)$, against a set of test cases, T, which was comprised of the costs of system analysis, Ca, test selection, Cs, test execution, Ce, result analysis and understanding, Cu, and result checking, Cc, as shown in Equation 1 in the Changeload Challenges section.

Thus, the costs of the risk-based and proposed goal-oriented approach are expressed as shown in Equation 39.

$$C(risk-based) = Ca(T) + Cs(T) + Ce(T) + Cu(T) + Cc(T)$$
$$C(goal-oriented) = Ca(T') + Cs(T') + Ce(T') + Cu(T') + Cc(T')$$

Equation 39: Cost of Testing Strategies

The following depicted in Equation 40 must hold true to validate a cost reduction using the goal-oriented approach.

$$C(goal-oriented) < C(risk-based)$$

Equation 40: Cost Savings Inequality as proposed by Leung and White (1991)

More specifically, Equation 41 shows the cost of selection for each approach as being dependent on the number of tests defined in the test suite, T_s, prior to the relevance cut-off being applied.

$$Ca(T') + Cs'(T_s') + Ce(T') + Cu(T') + Cc(T') < Ca(T) + Cs(T_s) + Ce(T) + Cu(T) + Cc(T)$$

Equation 41: Cost Savings Inequality with specific costs and different Selection Costs

Leung and White (1991) mentioned that a thorough analysis of a system has a greater cost, Ca, than a less thorough analysis, however, this cost was offset by the reduction in the cost of results understanding, Cu, due to the additional effort required to understand the system's behavior and its outputs (Leung & White, 1991). Thus, the increased cost of analysis, $Ca(T')$, and reduced cost of result understanding, $Cu(T')$, of the goal-oriented approach was equivalent to the cost of analysis, $Ca(T)$, and results understanding $Cu(T)$ of the risk-based approach, as shown in Equation 42.

$$\left(Ca(T') \geq Ca(T)\right) \wedge \left(Cu(T') \leq Cu(T)\right) \Rightarrow Ca(T') + Cu(T') \approx Ca(T) + Cu(T)$$

Equation 42: Analysis and Understanding Costs Equivalence

The cost savings inequality was combined with the cost equivalence and rewritten as shown in Equation 43.

$$Cs(T_s') + Ce(T') + Cc(T') < Cs(T_s) + Ce(T) + Cc(T)$$

Equation 43: Simplified Savings Inequality with different Selection Costs

The values of Cs, Ce, and Cc were dependent on the number of test cases in T, represented by the cardinal $|T|$, therefore, the cost of each step was rewritten as shown in Equation 44, where, s, e and c were constants and represented the selection cost, execution cost, and result checking cost, respectively.

Risk-Based	Goal-Oriented
$Cs(T_s) = s\lvert T_s\rvert$	$Cs'(T_s') = s'\lvert T_s'\rvert$
$Ce(T) = e\lvert T\rvert$	$Ce(T') = e\lvert T'\rvert$
$Cc(T) = c\lvert T\rvert$	$Cc(T') = c\lvert T'\rvert$

Equation 44: Reduction of Cost Terms

The constant s' represented a different selection cost to capture the cost associated with utilizing the goal-oriented approach due to the extension of the test selection process. The cost of execution of each test case and the cost of resulting checking was fixed for both approaches. The inequality was then simplified as shown in Equation 45.

$$s'|T_s'| + e|T'| + c|T'| < s|T_s| + e|T| + c|T|$$

Equation 45: Simplified Test Suite Cost Comparison Inequality

Appendix B

Specific Change	Target	Impact	Probability	Relevance
Unable to communicate with Server (1… n)	**Internal node connection faults**	Catastrophic	High	Very High
Unable to communicate x n		Catastrophic	High	Very High
Communication Failure: Server to Load Balancer		Marginal	Low	Medium
Communication Timeout: Server to Load Balancer		Negligible	High	Low
Communication Corruption		Negligible	Very Low	Negligible
Network link saturation		Marginal	High	Medium
Link congestion: Load Balancer to Servers		Marginal	Very High	High
Communication Delay: Load Balancer to Servers		Marginal	Very High	High
Unable to turn server on (stuck off)	**Effector**	Catastrophic	Very High	Mandatory
Unable to turn server off (stuck on)		Critical	Low	Medium
Unable to reduce content fidelity (stuck high)		Catastrophic	Very Low	Medium
Unable to increase content fidelity (stuck low)		Critical	Very Low	Medium
Unable to increase content fidelity (stuck medium)		Marginal	Very Low	Low
Unable to decrease content fidelity (stuck medium)		Marginal	Very Low	Low
Unable to measure bandwidth on server	**Gauge**	Critical	Low	Medium
Unable to measure response time from server		Critical	Low	Medium
Unable to measure server load		Marginal	High	Medium
Reported server load is invalid (-1)		Negligible	Very Low	Negligible
Reported server load is incorrect		Negligible	Very Low	Negligible
Reported server load is delayed		Negligible	Very High	Medium
Gauge not updating reading		Negligible	Low	Very Low
Operating Budget set too low	**Configuration**	Critical	Very High	Very High
Operating Budget set too high		Critical	High	High
Response time range too aggressive (too narrow)		Critical	Very High	Very High
Response time range too conservative (too broad)		Critical	Very High	Very High
Operating budget exhaustion (limit reached)		Catastrophic	Very High	Mandatory
Adaptive strategy changed (thresholds have changed during operation)		Marginal	High	Medium
Adapts too slow to fluctuations in server load + response time + bandwidth	**Adaptive Overhead**	Catastrophic	Very High	Mandatory
Adapts too quickly to fluctuations in server load + response time + bandwidth		Marginal	Very High	High
Adaptive functionality causes resource exhaustion		Catastrophic	Low	High
Adaptive thrashing (variables changed repeatedly within a short period of time)		Catastrophic	High	Very High
CPU Utilization Fluctuations: Servers	**Resource Fluctuations**	Marginal	Very High	High
Disk Latency Fluctuations: Servers		Marginal	Very High	High
Low Disk Space		Critical	Very Low	Medium
No disk space		Catastrophic	Very Low	Medium
High disk latency		Critical	High	High

Disk failure	Catastrophic	Very High	Mandatory
RAID Array Failure	Catastrophic	High	Very High
RAID Controller Failure	Catastrophic	High	Very High
Disk thrashing	Critical	Very High	Very High
RAM Utilization Fluctuations: Servers	Marginal	Very High	High
Server CPU Latency Utilization Patterns			
Steady Request Pattern (Start: n)	Marginal	High	Medium
Sinusoid Request Pattern (Trough: n, Peak: m)	Marginal	High	Medium
Stepwise Request Pattern (Start: n, Increment: m, End: p)	Marginal	High	Medium
Ramp Request Pattern (Start: n, End: p)	Marginal	High	Medium
Step Request Pattern (Trough: n, Peak: m)	Marginal	High	Medium
Exponential Request Pattern (Power: 2^p)	Marginal	High	Medium
Random Request Pattern (Min: n, Max: m)	Marginal	High	Medium
Server RAM Latency Utilization Patterns			
Steady Request Pattern (Start: n)	Marginal	High	Medium
Sinusoid Request Pattern (Trough: n, Peak: m)	Marginal	High	Medium
Stepwise Request Pattern (Start: n, Increment: m, End: p)	Marginal	High	Medium
Ramp Request Pattern (Start: n, End: p)	Marginal	High	Medium
Step Request Pattern (Trough: n, Peak: m)	Marginal	High	Medium
Exponential Request Pattern (Power: 2^p)	Marginal	High	Medium
Random Request Pattern (Min: n, Max: m)	Marginal	High	Medium
Server Disk Latency Utilization Patterns	Marginal	High	Medium
Steady Request Pattern (Start: n)	Marginal	High	Medium
Sinusoid Request Pattern (Trough: n, Peak: m)	Marginal	High	Medium
Stepwise Request Pattern (Start: n, Increment: m, End: p)	Marginal	High	Medium
Ramp Request Pattern (Start: n, End: p)	Marginal	High	Medium
Step Request Pattern (Trough: n, Peak: m)	Marginal	High	Medium
Exponential Request Pattern (Power: 2^p)	Marginal	High	Medium
Random Request Pattern (Min: n, Max: m)	Marginal	High	Medium
CPU Utilization Fluctuations: Load Balancer	Marginal	High	Medium
Disk Latency Fluctuations: Load Balancer	Marginal	High	Medium
RAM Utilization Fluctuations: Load Balancer	Marginal	High	Medium
Load Balancer CPU Latency Utilization Patterns			
Steady Request Pattern (Start: n)	Marginal	High	Medium
Sinusoid Request Pattern (Trough: n, Peak: m)	Marginal	High	Medium
Stepwise Request Pattern (Start: n, Increment: m, End: p)	Marginal	High	Medium
Ramp Request Pattern (Start: n, End: p)	Marginal	High	Medium
Step Request Pattern (Trough: n, Peak: m)	Marginal	High	Medium
Exponential Request Pattern (Power: 2^p)	Marginal	High	Medium
Random Request Pattern (Min: n, Max: m)	Marginal	High	Medium
Load Balancer RAM Latency Utilization Patterns			
Steady Request Pattern (Start: n)	Marginal	High	Medium
Sinusoid Request Pattern (Trough: n, Peak: m)	Marginal	High	Medium
Stepwise Request Pattern (Start: n, Increment: m, End: p)	Marginal	High	Medium
Ramp Request Pattern (Start: n, End: p)	Marginal	High	Medium
Step Request Pattern (Trough: n, Peak: m)	Marginal	High	Medium

Exponential Request Pattern (Power: 2^p)		Marginal	High	Medium
Random Request Pattern (Min: n, Max: m)		Marginal	High	Medium
Load Balancer Disk Latency Utilization Patterns				
Steady Request Pattern (Start: n)		Marginal	High	Medium
Sinusoid Request Pattern (Trough: n, Peak: m)		Marginal	High	Medium
Stepwise Request Pattern (Start: n, Increment: m, End: p)		Marginal	High	Medium
Ramp Request Pattern (Start: n, End: p)		Marginal	High	Medium
Step Request Pattern (Trough: n, Peak: m)		Marginal	High	Medium
Exponential Request Pattern (Power: 2^p)		Marginal	High	Medium
Random Request Pattern (Min: n, Max: m)		Marginal	High	Medium
Load Balancer at maximum load		Catastrophic	High	Very High
All Servers at maximum load		Catastrophic	High	Very High
High network congestion	**Fluctuations in network performance**	Critical	High	High
Low bandwidth connection for Servers		Critical	Low	Medium
High latency		Critical	High	High
High response time		Critical	High	High
Request Timeout		Critical	High	High
Low bandwidth connection for Clients		Negligible	High	Low
100% utilization		Catastrophic	Low	High
Network not found		Catastrophic	Very Low	Medium
No Connection		Catastrophic	Low	High
Network Utilization Pattern x 7				
Steady Request Pattern (Start: n)		Marginal	High	Medium
Sinusoid Request Pattern (Trough: n, Peak: m)		Marginal	High	Medium
Stepwise Request Pattern (Start: n, Increment: m, End: p)		Marginal	High	Medium
Ramp Request Pattern (Start: n, End: p)		Marginal	High	Medium
Step Request Pattern (Trough: n, Peak: m)		Marginal	High	Medium
Exponential Request Pattern (Power: 2^p)		Marginal	High	Medium
Random Request Pattern (Min: n, Max: m)		Marginal	High	Medium
Network Latency Pattern x 7				
Steady Request Pattern (Start: n)		Marginal	High	Medium
Sinusoid Request Pattern (Trough: n, Peak: m)		Marginal	High	Medium
Stepwise Request Pattern (Start: n, Increment: m, End: p)		Marginal	High	Medium
Ramp Request Pattern (Start: n, End: p)		Marginal	High	Medium
Step Request Pattern (Trough: n, Peak: m)		Marginal	High	Medium
Exponential Request Pattern (Power: 2^p)		Marginal	High	Medium
Random Request Pattern (Min: n, Max: m)		Marginal	High	Medium
Disk drive added	**New HW**	Negligible	Low	Very Low
RAM added		Negligible	Low	Very Low
NIC added		Negligible	Low	Very Low
RAID controller added		Negligible	Low	Very Low
RAID controller replaced		Marginal	Low	Medium
New network available		Negligible	Low	Very Low
New storage device added (NAS / SAN)		Negligible	Low	Very Low
Server added		Negligible	High	Low
Server removed		Critical	Low	Medium
Content File corruption	**File System Faults**	Catastrophic	Low	High

Content File unavailable		Catastrophic	High	Very High
Access Denied to Content File		Catastrophic	High	Very High
Content File not found (404)		Catastrophic	Low	High
Content File In use / locked		Critical	High	High
File System Corruption (general)		Critical	Very Low	Medium
Configuration File corruption		Catastrophic	Low	High
Configuration File unavailable		Catastrophic	Low	High
Access Denied to Configuration File		Catastrophic	Low	High
Configuration File not found		Catastrophic	Low	High
Configuration File In use / locked		Catastrophic	Low	High
WWW log unavailable		Catastrophic	Low	High
WWW log not found		Catastrophic	Low	High
WWW log corruption		Catastrophic	High	Very High
WWW log full		Catastrophic	Very High	Mandatory
Load Balancer not available	**Fluctuations in Load Balancer Performance and Availability**	Catastrophic	High	Very High
Load Balancer Failure		Catastrophic	High	Very High
Load Balancer misconfigured		Critical	Very High	Very High
Load Balancer high latency to Servers		Critical	Very High	Very High
Load Balancer high latency to Clients		Negligible	High	Low
Load Balancer congestion (internal)		Critical	Very High	Very High
Load Balancer timeout		Catastrophic	High	Very High
RAM bit errors	**Faulty HW**	Marginal	Very Low	Low
CPU bit errors		Marginal	Very Low	Low
NIC fails		Catastrophic	Very Low	Medium
NIC drops packets		Critical	High	High
Disk fails		Catastrophic	Very High	Mandatory
Network Cable faulty		Critical	Very Low	Medium
Power supply failure		Marginal	Low	Medium
Backup battery failure		Critical	Very Low	Medium
Update failed to apply	**OS Faults**	Marginal	Very High	High
Service terminate		Catastrophic	High	Very High
Buffer Overflow		Critical	Very High	Very High
Unexpected Reboot		Catastrophic	Low	High
System unresponsive		Catastrophic	High	Very High
Network Port locked		Catastrophic	Very Low	Medium
OS Corruption		Critical	Low	Medium
Device Driver failure		Critical	High	High
WWW service timeout	**Fluctuations in service availability**	Catastrophic	High	Very High
WWW service stopped		Critical	Low	Medium
WWW service fails		Catastrophic	Very Low	Medium
WWW service restarts unexpectedly		Critical	Low	Medium
WWW service unavailable		Catastrophic	Low	High
New Patch installed on Server	**New SW / OS Updates**	Negligible	Very High	Medium
New patch unsuccessfully installed on Server		Marginal	Very High	High
New patch locks OS files on Server		Critical	High	High
New patch corrupts files on Server		Critical	Low	Medium
New patch resets configuration on Server		Catastrophic	Very High	Mandatory

New patch closes ports on Server		Marginal	High	Medium
New patch affects WWW unexpectedly on Server		Marginal	High	Medium
New patch auto-reboots Server		Marginal	High	Medium
New patch hangs services on Server		Marginal	High	Medium
New Patch installed on Load Balancer		Negligible	Very High	Medium
New patch unsuccessfully installed on Load Balancer		Marginal	Very High	High
New patch locks OS files on Load Balancer		Critical	High	High
New patch corrupts files on Load Balancer		Critical	Low	Medium
New patch resets configuration on Load Balancer		Catastrophic	Very High	Mandatory
New patch closes ports on Load Balancer		Catastrophic	High	Very High
New patch affects WWW unexpectedly on Load Balancer		Catastrophic	High	Very High
New patch auto-reboots Load Balancer		Catastrophic	High	Very High
Additional software added to Server		Negligible	High	Low
WWW Services / Application Updated Successfully		Negligible	High	Low
WWW Services / Application Updated Unsuccessfully		Critical	Low	Medium
WWW Service failure due to failed upgrade on Server		Critical	Low	Medium
WWW Server configuration reset due to patch on Server		Catastrophic	High	Very High
WWW Server configuration reset due to upgrade on Server		Catastrophic	High	Very High
WWW Service fails to start after upgrade on Server		Critical	Low	Medium
New patch hangs services on Load Balancer		Catastrophic	High	Very High
DDoS Attack	**Attack**	Catastrophic	Low	High
Server hacked - content changed		Marginal	High	Medium
Server hacked - page redirects		Marginal	High	Medium
Server hacked - malicious program installed		Critical	Low	Medium
Man in the Middle Attack		Marginal	Low	Medium
0-Day Attack (unknown attack)		Critical	Very Low	Medium
Cross-linking Attack of Text	**Content Stealing**	Critical	Very High	Very High
Cross-linking Attack of Images		Critical	Very High	Very High
Server rebooted	**Operator Errors**	Critical	Very High	Very High
Server turned off		Catastrophic	Very High	Mandatory
Network cable unplugged		Catastrophic	Low	High
Load balancer turned off		Catastrophic	Low	High
Load balancer rebooted		Critical	High	High
Services restarted		Critical	High	High
Services stopped		Catastrophic	Low	High
Permissions changed incorrectly		Critical	Low	Medium
Backup during peak hours		Critical	High	High
Content file deleted		Catastrophic	High	Very High
Configuration file deleted		Catastrophic	High	Very High
Power Loss	**Power availability**	Critical	High	High
Power Overload		Marginal	Low	Medium
Cooling system malfunction		Catastrophic	Low	High
Physical access unavailable		Marginal	Very Low	Low

Unable to backup	**Backup Issues**	Marginal	Low	Medium
Backup medium unavailable		Negligible	Low	Very Low
Backup medium full		Negligible	Very Low	Negligible
Backup medium locked		Negligible	High	Low
Backup medium corrupt		Negligible	High	Low
Backup corrupt		Marginal	High	Medium
Regular requests	**Fluctuations in request type**	Negligible	Very High	Medium
Image only requests		Marginal	Very Low	Low
Text only requests		Negligible	Low	Very Low
Steady Request Pattern (Start: n)		Marginal	High	Medium
Sinusoid Request Pattern (Trough: n, Peak: m)		Marginal	High	Medium
Stepwise Request Pattern (Start: n, Increment: m, End: p)		Marginal	High	Medium
Ramp Request Pattern (Start: n, End: p)		Marginal	High	Medium
Step Request Pattern (Trough: n, Peak: m)		Marginal	High	Medium
Exponential Request Pattern (Power: 2^p)		Marginal	High	Medium
Random Request Pattern (Min: n, Max: m)		Marginal	High	Medium
1 req / min	**Fluctuation in number of requests**	Negligible	Very High	Medium
10 req / min		Negligible	Very High	Medium
50 req / min		Marginal	High	Medium
250 req / min		Marginal	High	Medium
1000 req / min		Critical	Low	Medium
2500 req / min		Critical	Low	Medium
10,000 req / min		Catastrophic	Very Low	Medium
1 users	**Fluctuation in number of users**	Negligible	Very High	Medium
10 users		Negligible	Very High	Medium
50 users		Marginal	High	Medium
250 users		Marginal	High	Medium
1000 users		Critical	Low	Medium
2500 users		Critical	Low	Medium
10,000 users		Catastrophic	Very Low	Medium
Combination of # of users and # of requests		Marginal	High	Medium
Steady Request Pattern (Start: n)	**Workloads**	Marginal	High	Medium
Sinusoid Request Pattern (Trough: n, Peak: m)		Marginal	High	Medium
Stepwise Request Pattern (Start: n, Increment: m, End: p)		Marginal	High	Medium
Ramp Request Pattern (Start: n, End: p)		Marginal	High	Medium
Step Request Pattern (Trough: n, Peak: m)		Marginal	High	Medium
Exponential Request Pattern (Power: 2^p)		Marginal	High	Medium
Random Request Pattern (Min: n, Max: m)		Marginal	High	Medium
392 Workload Variations		Marginal	High	Medium

Table 29: Risk-Based Approach Change Scenario Definitions Full Results

References

Agrawal, N., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2008). Towards realistic file-system benchmarks with CodeMRI. *SIGMETRICS Perform. Eval. Rev., 36*(2), 52-57. doi: 10.1145/1453175.1453184

Almeida, R., Madeira, H., & Vieira, M. (2010, June 21-25 2010). *From Performance to Resilience Benchmarking*. Paper presented at the Conference on Distributed Computing Systems Workshops (ICDCSW), 2010 IEEE 30th International

Almeida, R., & Vieira, M. (2011). *Benchmarking the resilience of self-adaptive software systems: perspectives and challenges*. Paper presented at the Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Waikiki, Honolulu, HI, USA.

Almeida, R., & Vieira, M. (2012a). *Changeloads for Resilience Benchmarking of Self-Adaptive Systems: A Risk-Based Approach.* Paper presented at the 2012 Ninth European Dependable Computing Conference.

Almeida, R., & Vieira, M. (2012b, Sept 10-14 2012). *Changeloads: A Fundamental Piece on the SASO Systems Benchmarking Puzzle.* Paper presented at the Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2012 IEEE Sixth International Conference on.

Andersson, J., Lemos, R., Malek, S., & Weyns, D. (2009). Modeling Dimensions of Self-Adaptive Software Systems. In H. C. Betty, Rog, L. rio, G. Holger, I. Paola & M. Jeff (Eds.), *Software Engineering for Self-Adaptive Systems* (pp. 27-47): Springer-Verlag.

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput., 1*(1), 11-33. doi: 10.1109/tdsc.2004.2

Avizienis, A., Laprie, J. C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on, 1*(1), 11-33. doi: 10.1109/TDSC.2004.2

Barbosa, R., Vinter, J., Folkesson, P., & Karlsson, J. (2005). *An approach to reducing the cost of fault injection.* Paper presented at the in Proceedings of Real-Time in Sweden 2005 (RTiS 2005), Skövde, Sweden.

Bondavalli, A., Lollini, P., Barbosa, R., Ceccarelli, A., L. Falai, Karlsson, J., . . . Vieira, M. (2009). Research Roadmap, Deliverable D3.2, AMBER - Assessing Measuring and Benchmarking Resilience: funded by the European Union.

Bra, P. D., Aerts, A., Berden, B., Lange, B. d., Rousseau, B., Santic, T., . . . Stash, N. (2003). *AHA! The adaptive hypermedia architecture*. Paper presented at the Proceedings of the fourteenth ACM conference on Hypertext and hypermedia, Nottingham, UK.

Brown, A. B., Hellerstein, J., Hogstrom, M., Lau, T., Lightstone, S., Shum, P., & Yost, M. P. (2004). *Benchmarking autonomic capabilities - Promises and Pitfalls.* Paper presented at the Proceedings of the International Conference on Autonomic Computing (ICAC'04).

Brown, G., Cheng, B. H. C., Goldsby, H., & Zhang, J. (2006). *Goal-oriented specification of adaptation requirements engineering in adaptive systems*. Paper presented at the Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, Shanghai, China.

Brun, Y., Serugendo, G. M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., . . . Shaw, M. (2009). Engineering Self-Adaptive Systems through Feedback Loops. In H. C. Betty, Rog, L. rio, G. Holger, I. Paola & M. Jeff (Eds.), *Software Engineering for Self-Adaptive Systems* (pp. 48-70): Springer-Verlag.

Burgman, M., Fidler, F., Mcbride, M., Walshe, T., & Wintle, B. (2006). Eliciting Expert Judgments: Literature Review: Australian Centre for Excellence in Risk Analysis (ACERA) - University of Melbourne.

Cailliau, A., & Lamsweerde, A. (2013). Assessing requirements-related risks through probabilistic goals and obstacles. *Requirements Engineering, 18*(2), 129-146. doi: 10.1007/s00766-013-0168-5

Cámara, J., Lemos, R., Vieira, M., Almeida, R., & Ventura, R. (2013). Architecture-based resilience evaluation for self-adaptive systems. *Computing*, 1-34. doi: 10.1007/s00607-013-0311-7

Cámara, J., Lemos, R. d., Laranjeiro, N., Ventura, R., & Vieira, M. (2013). *Robustness Evaluation of Controllers in Self-Adaptive Software Systems.* Paper presented at the Proceedings of the 6th Latin American Symposium on Dependable Computing (LADC 2013).

Cheng, B., & Atlee, J. M. (2007). *Research directions in requirements engineering.* Paper presented at the Future of Software Engineering (FOSE'07).

Cheng, B., Lemos, R. d., Giese, H., Inverardi, P., Magee, J., Andersson, J., . . . Whittle, J. (2009). *Software Engineering For Self-Adaptive Systems*.

Cheng, S. W., Garlan, D., & Schmerl, B. (2009, 18-19 May 2009). *Evaluating the effectiveness of the Rainbow self-adaptive system.* Paper presented at the Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on.

Cheng, S. W., Huang, A.-C., Garlan, D., Schmarl, B., & Steenkiste, P. (2004). *Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure.* Paper presented at the Proceedings of the International Conference on Autonomic Computing (ICAC'04).

Cheng, V.-W. (2008). *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation.* (Doctor of Philosophy), Carnegie Mellon University. (CMU-ISR-08-113)

Chvatal, V. (1979). A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research, 4*(3), 233-235. doi: 10.2307/3689577

Cin, M. D., Kanoun, K., Buchacker, K., Zuinga, L. L., Lindstrom, R., Johanson, A., . . . Suri, N. (2002). DBench Dependability Benchmark - Workload and Faultload Selection (ETIE3). http://webhost.laas.fr/TSF/DBench/: The European Commission of Community Research in Information Society Technologies (IST).

Council, T. P. P. (2010). TPC Benchmark C, Standard Specification, Version 5.11. Retrieved June 1, 2013, from http://www.tpc.org/tpcc

Dardenne, A., Lamsweerde, A. v., & Fickas, S. (1993). Goal-directed requirements acquisition. *Sci. Comput. Program., 20*(1-2), 3-50. doi: 10.1016/0167-6423(93)90021-g

Feather, M. S., Fickas, S., Lamsweerde, A. V., & Ponsard, C. (1998). *Reconciling System Requirements and Runtime Behavior*. Paper presented at the Proceedings of the 9th international workshop on Software specification and design.

Fernandez, J., & Garcia, J. M. (1999). Representative Benchmarks for Commercial Workloads. *X Jornadas de Paralelismo*(September 1999).

Friginal, J., de Andres, D., Ruiz, J.-C., & Gil, P. (2011). *On Selecting Representative Faultloads to Guide the Evaluation of Ad Hoc Networks.* Paper presented at the 5th Latin-American Symposium on Dependable Computing (LADC), 2011 Sao Jose dos Campos.

Galeebathullah, B., & C.P.Indumathi. (2010). A Novel Approach for Controlling a Size of a Test Suite with Simple Technique. *International Journal on Computer Science and Engineering (IJCSE), Vol 2*(Issue 3), 614-618.

Ganek, A. G., & Corbi, T. A. (2003). The dawning of the autonomic computing era. *IBM Syst. J., 42*(1), 5-18. doi: 10.1147/sj.421.0005

Garlan, D. (2010). *Software engineering in an uncertain world*. Paper presented at the Proceedings of the FSE/SDP workshop on Future of software engineering research, Santa Fe, New Mexico, USA.

Gil, P., Arlat, J., Madeira, H., Crouzet, Y., Jarboui, T., Kanoun, K., . . . Gracia, J. (2002). DBench Dependability Benchmark - Fault Representativeness (ETIE2) *DBench Dependability Benchmark*: The European Commission of Community Research in Information Society Technologies (IST).

Graefe, G., Idreos, S., Kuno, H., & Manegold, S. (2010). *Benchmarking Adaptive Indexing.* Paper presented at the Proceeding TPCTC'10 Proceedings of the Second TPC technology conference on Performance evaluation, measurement and characterization of complex systems

Group, Y. (2002). How much is an hour of downtime worth to you? (pp. 178 - 187). Must-Know Business Continuity Strategies.

Harrold, M. J., Gupta, R., & Soffa, M. L. (1993). A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol., 2*(3), 270-285. doi: 10.1145/152388.152391

Heegaard, P. E., & Trivedi, K. S. (2009). Network survivability modeling. *Computer Networks, 53*(8), 1215-1234. doi: 10.1016/j.comnet.2009.02.014

Hellerstein, J. L., Diao, Y., Parekh, S., & Tilbury, D. M. (2004). *Feedback Control of Computing Systems*: John Wiley \\& Sons.

Hemmati, H., Briand, L., Arcuri, A., & Ali, S. (2010). *An enhanced test case selection approach for model-based testing: an industrial case study*. Paper presented at the Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, Santa Fe, New Mexico, USA.

Huebscher, M. C., & McCann, J. A. (2004). *Evaluation Issues in Autonomic Computing.* Paper presented at the Grid and Cooperative Computing – GCC 2004 Workshops (2004).

Hurtado, S., Sen, S., & Casallas, R. (2011). *Reusing legacy software in a self-adaptive middleware framework*. Paper presented at the Adaptive and Reflective Middleware on Proceedings of the International Workshop, Lisbon, Portugal.

IBM. (2003). An architectural blueprint for autonomic computing. *Tech. rep.*, from http://users.encs.concordia.ca/~ac/ac-resources/AC_Blueprint_White_Paper_4th.pdf

Jorgensen, P. C. (2002). *Software Testing: A Craftsman's Approach*: CRC Press.

Kaddoum, E., Raibulet, C., Georg, J.-P., Picard, G., & Gleizes, M.-P. (2010). *Criteria for the evaluation of self-* systems*. Paper presented at the Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, Cape Town, South Africa.

Kang, Z., Kumar, A., Harrison, T. P., & Yen, J. (2011). Analyzing the Resilience of Complex Supply Network Topologies Against Random and Targeted Disruptions. *Systems Journal, IEEE, 5*(1), 28-39. doi: 10.1109/JSYST.2010.2100192

Kanoun, K., Madeira, H., & Arlat, J. (2002). A Framework for Dependability Benchmarking *in Supplement of the 2002 Int. Conf. on Dependable Systems and Networks (DSN-2002)* (pp. 12-15).

Kanoun, K., Madeira, H., Crouzet, Y., Cin, M. D., Moreira, F., García, J.-C. R., . . . Yuste, P. (2004). DBench Dependability Benchmarks. In B. a. C. Deliverables BDEV3 (Ed.): European Community under the "Information Society Technology" Programme (1998-2002).

Khalil, Y. H., Elmaghraby, A., & Kumar, A. (2008, 6-9 July 2008). *Evaluation of resilience for Data Center systems.* Paper presented at the Computers and Communications, 2008. ISCC 2008. IEEE Symposium on.

Laddaga, R. (2006, 24-24 Sept. 2006). *Self Adaptive Software Problems and Projects.* Paper presented at the Software Evolvability, 2006. SE '06. Second International IEEE Workshop on.

Laddaga, R., & Robertson, P. (2000). Self Adaptive Software: A Position Paper. Retrieved March 1, 2013

Laprie, J.-C. (2008). *From Dependability to Resilience.* Paper presented at the Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN-2008).

Lemos, R. d., Giese, H., Müller, H. A., Shaw, M., Andersson, J., Baresi, L., . . . Wuttke, J. (2010, October 2010). *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap.* Paper presented at the Dagstuhl Seminar Proceedings 10431 on Software Engineering for Self-Adaptive Systems.

Leung, H. K. N., & White, L. (1991, 15-17 Oct 1991). *A cost model to compare regression test strategies.* Paper presented at the Software Maintenance, 1991., Proceedings. Conference on.

Lorenzoli, D., Tosi, D., Venticinque, S., & Micillo, R. A. (2007). *Designing multi-layers self-adaptive complex applications*. Paper presented at the Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting, Dubrovnik, Croatia.

Madan, B. B., Goševa-Popstojanova, K., Vaidyanathan, K., & Trivedi, K. S. (2004). A method for modeling and quantifying the security attributes of intrusion tolerant systems. *Performance Evaluation, 56*(1-4), 167-186. doi: 10.1016/j.peva.2003.07.008

Madeira, H., Kanoun, K., Arlat, J., Costa, D., Crouzet, Y., Cin, M. D., . . . Madeira, H. (2002, October 23-25, 2002). *Towards a Framework for Dependability Benchmarking.* Paper presented at the 4th European Dependable Computing Conference (EDCC4),, Toulouse, France.

Madeira, H., & Koopman, P. (2001). *Dependability Benchmarking: making choices in an n-dimensional problem space.* Paper presented at the Proceedings of the first workshop on Evaluating and Architecting System Dependability.

Meyer, J. F. (2009). *Defining and Evaluating Resilience : A Performability Perspective.* Paper presented at the in Proceedings of the International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-9).

Moorsel, A. v., Alberdi, E., Bondavalli, A., Durães, J., Esposito, R., Falai, L., . . . Zhang, H. (2009). *Final State of the Art, Deliverable D2.2, AMBER - Assessing, Measuring and Benchmarking Resilience*.

Morandini, M., Penserini, L., & Perini, A. (2008). *Towards goal-oriented development of self-adaptive systems*. Paper presented at the Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, Leipzig, Germany.

Parekh, J., Kaiser, G., Gross, P., & Valetto, G. (2006). Retrofitting Autonomic Capabilities onto Legacy Systems. *Cluster Computing, 9*(2), 141-159. doi: 10.1007/s10586-006-7560-6

Potts, C. (1995). *Using schematic scenarios to understand user needs*. Paper presented at the Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques, Ann Arbor, Michigan, United States.

Pressman, R. S. (2005). *Software Engineering: A Practioner's Approach* (6e ed.). New York: McGraw-Hill Education.

Quadri, S. M. K., & Farooq, S. U. (2010). *Software Testing – Goals, Principles, and Limitations.* Paper presented at the International Journal of Computer Applications (0975 – 8887).

Roberto, N. (2013). On Fault Representativeness of Software Fault Injection. *IEEE Transactions on Software Engineering, 39*(1), 80-96.

Salehie, M., & Tahvildari, L. (2009). Self-adaptive software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems, 4*(2), 1-42. doi: 10.1145/1516533.1516538

Tamura, G., Villegas, N. M., Muller, H. A., Sousa, J. P., Becker, B., Karsai, G., . . . Wong, K. (2012). *Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems*: Springer.

Traeger, A., Zadok, E., Joukov, N., & Wright, C. P. (2008). A nine year study of file system and storage benchmarking. *Trans. Storage, 4*(2), 1-56. doi: 10.1145/1367829.1367831

van Lamsweerde, A. (2000). *Requirements engineering in the year 00: a research perspective*. Paper presented at the Proceedings of the 22nd international conference on Software engineering, Limerick, Ireland.

van Lamsweerde, A. (2001, 2001). *Goal-oriented requirements engineering: a guided tour.* Paper presented at the Proceedings of the Fifth IEEE International Symposium on Requirements Engineering.

van Lamsweerde, A., & Letier, E. (1998). *Integrating obstacles in goal-driven requirements engineering*. Paper presented at the Proceedings of the 20th international conference on Software engineering, Kyoto, Japan.

van Lamsweerde, A., & Letier, E. (2000). Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Trans. Softw. Eng., 26*(10), 978-1005. doi: 10.1109/32.879820

Vieira, M., & Madeira, H. (2003). *A dependability benchmark for OLTP application environments*. Paper presented at the Proceedings of the 29th international conference on Very large data bases - Volume 29, Berlin, Germany.

Vieira, M., & Madeira, H. (2004, 28-30 Sept. 2004). *Portable faultloads based on operator faults for DBMS dependability benchmarking.* Paper presented at the Computer Software

and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International.

Visser, W., Pasareanu, C. S., & Khurshid, S. (2004). *Test input generation with java PathFinder*. Paper presented at the Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, Boston, Massachusetts, USA.

Weicker, R. P. (1990). An Overview of Common Benchmarks. *Computer, 23*(12), 65-75. doi: 10.1109/2.62094

Weyns, D., Iftikhar, M. U., Iglesia, D. G. d. l., & Ahmad, T. (2012). *A survey of formal methods in self-adaptive systems*. Paper presented at the Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering, Montreal, Quebec, Canada.

Williams, R., Behrens, S., & Pandelios, G. (1999). SRE Method Description (Version 2.0).

Xavier, K. S., Hanazumi, S., & Melo, A. C. V. d. (2008). *Using Formal Verification to Reduce Test Space of Fault-Tolerant Programs*. Paper presented at the Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods.

Zhang, J., & Cheng, B. H. C. (2007). *Towards Re-engineering Legacy Systems for Assured Dynamic Adaptation*. Paper presented at the Proceedings of the International Workshop on Modeling in Software Engineering.