12-2010

A Reasoning Framework for Dependability in Software Architectures

Tacksoo Im
Clemson University, tim@cs.clemson.edu

A Reasoning Framework for Dependability in Software Architectures

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

by
Tacksoo Im
August 2010

Accepted by:
Dr. John D. McGregor, Committee Chair
Dr. Harold C. Grossman
Dr. Jason O. Hallstrom
Dr. Pradip K. Srimani

Abstract

The degree to which a software system possesses specified levels of software quality attributes, such as performance and modifiability, often have more influence on the success and failure of those systems than the functional requirements. One method of improving the level of a software quality that a product possesses is to reason about the structure of the software architecture in terms of how well the structure supports the quality. This is accomplished by reasoning through software quality attribute scenarios while designing the software architecture of the system.

As society relies more heavily on software systems, the dependability of those systems becomes critical. In this study, a framework for reasoning about the dependability of a software system is presented. Dependability is a multi-faceted software quality attribute that encompasses reliability, availability, confidentiality, integrity, maintainability and safety. This makes dependability more complex to reason about than other quality attributes. The goal of this reasoning framework is to help software architects build dependable software systems by using quantitative and qualitative techniques to reason about dependability in software architectures.

Dedication

To my father Chungun Im and my mother Youngsoon Ji.

Acknowledgments

I would like to thank the following individuals who have helped me greatly in writing this dissertation: Dr. John D. McGregor, Kyungsoo Im, Changsoo Im, Andres Diaz-Pace, Soujanya Vullam, Sebastien Gagne, and the students of CPSC 875 (Software Architecture) in spring 2010.

Table of Contents

Title Page i

Abstract ii

Dedication iii

Acknowledgments iv

List of Tables viii

List of Figures ix

1 Introduction 1

2 Background 4
- 2.1 Quality Attributes 4
- 2.2 Software Architecture 8
- 2.3 Reasoning Frameworks 11
- 2.4 ArchE Architecture Design Assistant 14
- 2.5 Qualitative Reasoning 17

3 Related Work 20

4 Quantitative Reasoning Frameworks 22
- 4.1 Reliability 24
  - 4.1.1 Analytic Theory 24
  - 4.1.2 Model Representation 26
  - 4.1.3 Interpretation 28
  - 4.1.4 Evaluation Procedure 29
  - 4.1.5 Implementation 32
- 4.2 Availability 38
  - 4.2.1 Problem Description 38
  - 4.2.2 Analytic Theory 38
  - 4.2.3 Analytic Constraints 39
  - 4.2.4 Model Representation 39
  - 4.2.5 Interpretation 39
  - 4.2.6 Evaluation Procedure 40
  - 4.2.7 Discussion of Technique 41
  - 4.2.8 Related Work 42
- 4.3 Maintainability 43
  - 4.3.1 Problem Description 43
  - 4.3.2 Analytic Theory 43

4.3.3 Anaytic Constraints . . . 44
4.3.4 Model Representation . . . 44
4.3.5 Interpretation . . . 44
4.3.6 Evaluation Procedure . . . 45
4.3.7 Example Application of Technique . . . 45
4.3.8 Discussion of Technique . . . 46
4.3.9 Related Work . . . 46

5 Qualitative Reasoning Frameworks . . . 48
5.1 Confidentiality & Integrity . . . 48
5.1.1 Problem Description . . . 50
5.1.2 Analytic Theory . . . 51
5.1.3 Analytic Constraints . . . 52
5.1.4 Model Representation . . . 52
5.1.5 Interpretation . . . 53
5.1.6 Evaluation Procedure . . . 54
5.2 Safety . . . 57
5.2.1 Initial Safety Analyses . . . 58
5.2.2 Problem Description . . . 59
5.2.3 Analytic Theory . . . 60
5.2.4 Analytic Constraints . . . 61
5.2.5 Model Representation . . . 62
5.2.6 Interpretation . . . 62
5.2.7 Evaluation Procedure . . . 62
5.2.8 Summary . . . 64
5.2.9 Example Application of Technique . . . 64
5.2.9.1 Perform Initial Safety Analyses . . . 64
5.2.9.2 Translate into Safety Scenarios . . . 65
5.2.9.3 Apply the Constituent Reasoning Frameworks . . . 66
5.2.9.4 Apply the Safety Reasoning Framework . . . 67
5.2.10 Related Work . . . 68
5.3 Conclusion . . . 70

6 Assembling the Reasoning Frameworks . . . 71
6.1 Background . . . 72
6.2 Attribute Architectures . . . 73
6.3 Combining Attribute Architectures . . . 75
6.3.1 Connection Types and Compositions . . . 75
6.3.2 Composition Operator Defined . . . 76
6.3.3 Effects of Combining Attribute Architectures . . . 79
6.3.4 Summary . . . 80
6.4 Example with AADL . . . 81
6.5 Summary . . . 84
6.6 Assembling the Results from the Reasoning Frameworks . . . 84

7 Case Study on Apache . . . 89
7.1 Apache in AADL . . . 89
7.2 Parameters for Dependability analysis . . . 90
7.3 Dependability analysis of Apache . . . 94
7.3.1 Reliability . . . 95
7.3.2 Availability . . . 97

7.3.3 Confidentiality & Integrity . . . 99
7.3.4 Maintainability . . . 100
7.3.5 Safety . . . 101
7.4 Combination of attrbitue architectures in Apache . . . 102
7.5 Conclusion . . . 103

8 Implementation & Evaluation . . . 104
8.1 Implementation Issues . . . 104
8.1.1 Quantitative Reasoning Frameworks . . . 105
8.1.2 Qualitative Reasoning Frameworks . . . 106
8.2 Evaluation . . . 109
8.2.1 Introduction . . . 109
8.2.2 Efficacy of the Dependability Reasoning Framework . . . 110
8.2.3 Usability of the Dependability Reasoning Framework . . . 112
8.2.4 Suggestions for the Dependability Reasoning Framework . . . 112
8.3 Conclusion . . . 112

9 Conclusions and Future Work . . . 113

Bibliography . . . 115

List of Tables

2.1 Definition of Some Quality Attributes 5
2.2 Quality Priorities by market 7
2.3 Architectures and its affected Quality Attributes 10

4.1 A Reliability Scenario 26
4.2 Responsibilities and Reliability 30
4.3 Reliability Rate for the Responsibilities in Scenario 34
4.4 Responsibilities and Availability 40
4.5 Information of a Maintenance Scenario 46

5.1 Satisficing Authorization Level for CTAS 52
5.2 Some Quality Attributes Used to Analyze Safety 59
5.3 General Scenario for Safety 61
5.4 Results of a Function Hazard Analysis on CTAS 64
5.5 Safety Scenario related to potential confidentiality failure 66
5.6 Confidentiality scenario after mapping from the safety scenario 66

6.1 Connector Types 75
6.2 Combination of Connectors 76
6.3 Dependency Connector Combination Specifics 77
6.4 Effects of Combination on Quality Attributes 80
6.5 Relationship Types in AADL 81
6.6 AADL equivalent of Combined Connectors 82
6.7 Scales of sub-qualities 86
6.8 Sub-qualities mapped to standard scale 86
6.9 Current quality profile 87

7.1 Reliability Rate for the Responsibilities in Scenario 97
7.2 Availability Rate for the Responsibilities in Scenario 98
7.3 Availability Rate for the Responsibilities in Scenario 99
7.4 Values for a Maintainability Scenario 100
7.5 Events affecting Safety 101
7.6 Safety Scenarios and its Analysis 102

8.1 Tactics for Quantitative Reasoning Frameworks 107
8.2 Tactics for Qualitative Reasoning Frameworks 108

List of Figures

2.1 The dependability tree 6
2.2 GameBoard Module Decomposition 9
2.3 Reasoning Framework Diagram 13
2.4 ArchE Process Diagram 16

4.1 CTAS Architecture (Dependency View of Modules) 23
4.2 A Serial Connection 25
4.3 A Parallel Connection 25
4.4 Use Case Diagram for CTAS (Clemson Trasit Assistance System) 27
4.5 A CTAS Activity Diagram 28
4.6 Relationships in a Scenario 30
4.7 Reliability Scenario 33
4.8 Responsibilities and their reliabilities 34
4.9 Mapping from Scenario to Responsibility 35
4.10 The Relationships among the responsibilities 36
4.11 Sequence of responsibilities 36
4.12 Analysis result of the Scenario 37

5.1 Confidentiality Scenario 55
5.2 Safety Analysis 60
5.3 Transformations between theories 63
5.4 Fault Tree for Failure in User Data Management 65
5.5 CTAS Software Architecture 67
5.6 Star Plot of Safety Analysis 68

6.1 Dependency Graph of CTAS 72
6.2 Attribute Architectures 73
6.3 Combining Attribute Architectures 79
6.4 Confidentiality Attribute Architecture 82
6.5 Reliability Attribute Architecture 83
6.6 Combined Architecture 84
6.7 Garp 3 Static Model Fragment 88
6.8 Garp 3 Modified Static Model Fragment 88

7.1 Apache server's APR, MPM and Modules 91
7.2 Apache's server core, client and OS 92
7.3 Apache server's system in AADL 93
7.4 Apache Reliability Attribute Architecture 95
7.5 Apache Availability Attribute Architecture 97
7.6 Apache Confidentiality Attribute Architecture 100
7.7 Apache Maintainability Attribute Architecture 101
7.8 Star Plot of Apache Dependability Analysis 102

7.9 Combined Attribute Architecture of Apache 103

8.1 Self Rated Skill Level as Software Architect 109
8.2 Did the framework help in learning about dependability? 110
8.3 Did the framework help design CMCS with respect to dependability? 111
8.4 Did the framework help in the architecture definition process? 111

Chapter 1

Introduction

Software has become so pervasive that a program director of the F-22 Raptor has commented that "The only thing you can do with an F-22 that does not require software is take a picture of it [42]." As software becomes more pervasive, the need for more dependable software is also growing. This is largely because more and more critical systems such as energy management, health care, finance, transportation and telecommunications are dependent on software. NIST estimates that up to 50 billion dollars a year is wasted because of software errors [75]. As software is increasingly used in life-critical items such as cars and airplanes, the potential for the loss of life is also increasing. But many software systems are not meeting the dependability goals and organizations are forced to depend less on software systems. An infamous example is the FBI's VCF (Virtual Case File) which tried to replace a paper-based work environment, but failed to do so because it was not dependable. VCF was scrapped and cost the US tax payers $170 million dollars [29].

Dependability is defined as a property of a computer system where reliance can be justifiably placed on the service it delivers [61]. It is not a directly measurable quality but the level of dependability of a system is often described using a combination of reliability, availability, safety, confidentiality, integrity and maintainability. These qualities are considered the sub-attributes of dependability and the importance of each attribute to the overall concept of dependability varies from one domain to another. The definition of dependability has been slowly evolving in the Fault-Tolerant Computing community, and especially in the IFIP (International Federation for Information Processing) working group 10.4 [61].

Central to achieving dependability in a software system is the software architecture. Software architecture is an abstraction of a software system and it contains the early design decisions that are critical for the success of a system [6]. This is because the early design decisions provide the initial solutions to achieving the quality attribute requirements of a system. Since most software redesigns occur because the quality attribute requirements of the system have not been satisfied, creating software architectures that help achieve those requirements has become important.

Architectural expertise is used to help design software architectures that can satisfy the quality attribute requirements of a system. For quality attributes such as performance and modifiability, there are reasoning frameworks that capture the expert's knowledge about each architectural attribute [18]. These expert systems do not produce a full architecture for the user but instead assist the user by determining if the design decisions are leading to an architecture that satisfies the user's quality attribute requirements.

The nature of dependability makes it difficult to characterize. What makes dependability different from other qualities such as performance or modifiability is that it is a composite quality. The qualities that comprise dependability are very different from one another and each quality has its own measurement scale. There are also complex trade-offs between the qualities. These issues make reasoning about dependability very difficult.

One aspect of dependability that makes it difficult to reason about is its blend of qualitative and quantitative scales. For example, reliability is a sub-attribute of dependability for which a reasoning framework can be based on a quantitative analytic theory that produces values between 0 and 100. But confidentiality or integrity are unlikely to be based on a quantitative analytic theory. How to combine the results from these two different types of theories and make a judgement on the dependability of the architecture is an open problem.

Reasoning frameworks are also expected to give design suggestions to the user in the form of architectural tactics. The identification of these tactics for achieving dependability is necessary in order to define a reasoning framework for dependability. Also, each architectural tactic represents a positive or negative contribution to satisficing [1] the pertinent qualities. Therefore, an architectural tactic that might increase one aspect of dependability can negatively affect other parts of dependability.

To summarize, there is a need for a reasoning framework for dependability but the knowl-

[1] *Decision making where possible alternatives are considered until a choice that yields a satisfactory result is chosen. [36]*

edge about dependability has not been organized very well to be applied to software architecture. Dependability is an amalgamation of qualitative and quantitative qualities and it is not simple to blend those qualities into one reasoning framework.

The general hypothesis for this research is that a reasoning framework for dependability can be built and that it is possible to blend the qualitative and quantitative attributes of dependability to build a reasoning framework for it.

The contributions that will be made from this research are the following:

- A technique for blending quantitative and qualitative quality attribute measures to provide guidance for software architects.

- A separate reasoning framework for each of the sub-attributes of dependability. These reasoning frameworks will be implemented as external plugins for ArchE (Architecture Expert), a platform for reasoning frameworks [3].

- A dependability reasoning framework that combines quantitative and qualitative results from the reasoning frameworks for the sub-attributes of dependability.

To verify the result of the research, a user study will be conducted. Two groups of students will be invited to design a software architecture with dependability as a required quality attribute. One group will be asked to use the dependability reasoning framework in the process and the other group will be left without using it. Both quantitative and qualitative results of the user study will be presented as evidence of the effectiveness and acceptability of the reasoning frameworks.

Chapter 2

Background

In this chapter, the concept of quality attributes and how dependability is different from other quality attributes will be explained. Secondly, the concepts of software architecture and reasoning frameworks will be introduced. Lastly, the tool ArchE (Architecture Expert Design Assistant) and how qualitative reasoning can be used to improve dependability will be presented.

2.1 Quality Attributes

Quality attributes are non-functional properties of a software system. There are many quality attributes that are of interest in a software system including availability, modifiability, performance, security, testability, and usability.

Table 2.1 provides definitions of some quality attributes. Availability and reliability are closely related but different in one crucial aspect. Availability can be high even if the reliability is low as long as the time-to-repair is short. In simple terms, the mean-time-to-repair (MTTR) is important for availability and the mean-time-to-failure (MTTF) is important for reliability.

Previous studies on quality attributes have been hazy because they were not operational. For example, it is meaningless to discuss the modifiability of a system because every system is modifiable with respect to one set of changes and not modifiable with respect to another. [6] It was also difficult to categorize in which quality a certain aspect would belong, an aspect such as "system slowdown" could be related to performance issues or usability issues.

The solution to this problem is quality attribute scenarios, described in section 2.3. Quality

Quality Attribute	Definition
Availability	The degree to which a system or component is operational and accessible when required for use [43].
Modifiability	The degree to which a system or component facilitates the incorporation of changes, once the nature of the desired change has been determined [43].
Performance	A category of quality measures that address how well a system functions.
Security	The ability of a system to manage, protect, and distribute sensitive information.
Testability	The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met [43].
Usability	The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component [43].

Table 2.1: Definition of Some Quality Attributes

attribute scenarios give us a way to characterize quality attributes in terms of quality-attribute-specific requirements [55].

Dependability is a bit different from the quality-attributes in Table 2.1 and is defined differently in different domains. The definition given earlier states that dependability is defined as that property of a computer system such that reliance can justifiably be placed on the service it delivers [61] [94]. Most would agree with that definition but it is too general for detailed study or for decision making. Most of the definitions of dependability in the literature use some subset of the quality attributes in figure 2.1.

We will use the following definition for dependability: reliability, availability, safety, maintainability, integrity and confidentiality. The definition for each are the following [5]:

- Availability: readiness for usage
- Reliability: continuity of service
- Safety: non-occurrence of catastrophic consequences on the environment
- Confidentiality: non-occurrence of unauthorized disclosure of information
- Integrity: non-occurrence of improper alterations of information
- Maintainability: aptitude to undergo repairs and evolution

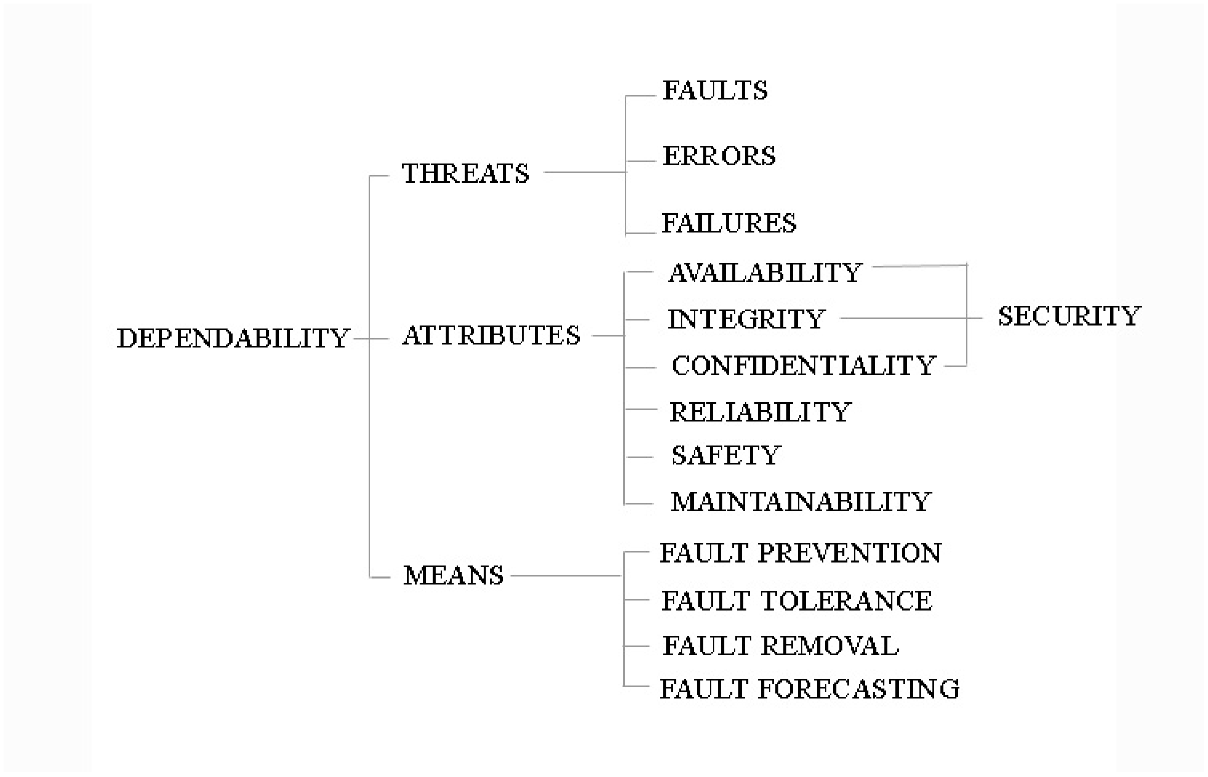


Figure 2.1: The dependability tree

Table 2.2: Quality Priorities by market

	Availability	Reliability	Safety	Security	Maintainability
E-commerce	1	2	3T	3T	4
General Embedded	1	2	3	4	5
Aerospace Embedded	3	2	1	4	5
E-government	3	2	4	1	5
Telecommunications	1	2	3T	3T	4
Data Storage	2	1	4	3	5
Middleware	1	3	4	2	5

Availability, integrity and confidentiality are collectively referred to as security. The importance of each attribute to overall dependability depends on the domain of the system.

To fully illustrate the issues, the literature on dependable systems was examined. Two points are made with this data. First, the range of qualities which most often comprise dependability was explored. Second, the difference in priority among those composed qualities was examined. Table 2.2 shows five of the most frequently used non-functional qualities across the horizontal axis[1]. The vertical axis lists a set of system types. For each system type the literature concerning that type of system was examined and the frequency with which each of the qualities is mentioned was counted. The frequency of mention was used as an indication of priority [71]. No effort was made to be exhaustive. The data is purely illustrative of the point that definitions of dependability vary across domains. Rather than try to reach a consensus on a universal definition of dependability the assumption is that the definition will remain domain specific.

Certain quality-attributes are not isolated and impact other quality-attributes. For example, nearly every improvement in other quality-attributes negatively impacts performance. An interesting aspect of dependability is that this trade-off among quality-attributes occurs within the quality attributes that compose dependability. Here are the interactions:

- Availability vs. Confidentiality

  Design decisions that increase the availability of the system can decrease confidentiality by introducing prolonged exposure of data to system users [89].

- Availability vs. Integrity

  Design decisions that increase the availability of the system can decrease integrity by exposing data for longer periods of time to the possibility of malicious changes [89].

[1]Integrity and confidentiality were combined as security for convenience.

- Safety vs. Confidentiality

  Design decisions that increase safety can come in direct conflict with confidentiality because safety decisions often require distribution of knowledge (e.g. replication prevents loss of data at the cost of a higher probability of theft and unauthorized modification) [82].

- Safety vs. Integrity

  Design decisions that increase safety can come in direct conflict with integrity because safety decisions often require distribution of knowledge [82].

2.2 Software Architecture

Software architecture is defined as "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" [6]. An essential quality of software architecture is that it shows the relationship between architectural elements. There are two parts associated with an element: the public part which describes how the element interacts with the external world and the private part which describes the internal implementation. Any description regarding two or more elements in a software architecture is an architectural description.

Also, there can be more than one type of structure associated with an architecture. For example different kinds of elements can be described (implementation unit and process unit), more than one kind of interaction among elements can be shown (subdivision and synchronization) and different contexts (development time and runtime) [6].

A view is a representation of such a coherent set of structures. Here are some views that can be part of an architecture description.

- Allocation deployment view

  This shows the process allocation of a system. Some processes can be executed in one processor or machine and some processes can be executed in multiple processors or machines.

- Module decomposition view

  This shows how the system is decomposed into modules. The relationship between two modules can be association or composition.

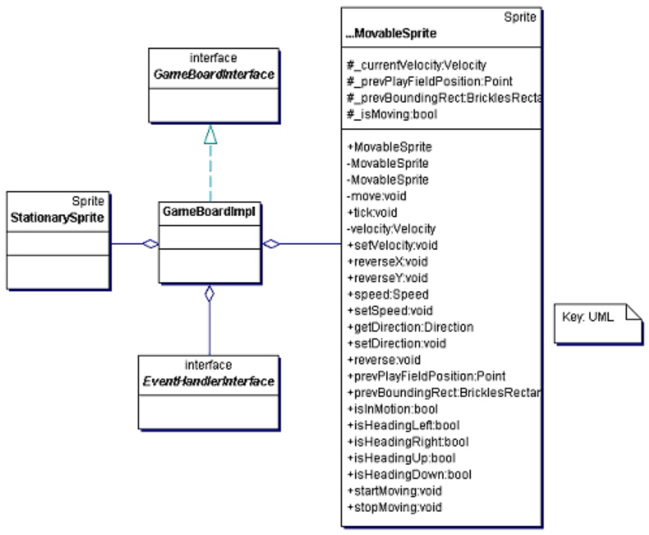


Figure 2.2: GameBoard Module Decomposition

- Module generalization view

  This shows how modules can be generalized into categories using inheritance.

- Component and connector view

  This shows the operational behavior or the flow of computation of the system.

Figure 2.2 shows a module decomposition view for the AGM (Arcade Game Maker) pedagogical product line [67]. It shows how a typical GameBoard in AGM is composed of movable and stationary sprites with the addition of eventhandlers.

These kinds of software architecture descriptions (or views) create a basis for reasoning about quality-attribute requirements. Any requirement involving the cooperation of multiple elements is architectural and by carefully balancing the conflicting constraints of each element in a system, the quality requirements of the system can be optimized, but not always achieved. Quality-attributes of a system cannot be achieved in isolation and are the combined result of architectural decisions and non-architectural decisions such as algorithms and coding practices.

Before a software architecture can be defined, the qualities that are most important to the system must be elicited. To elicit those quality-attributes, the Architecture Business Cycle (ABC) is used. This is the phase where the requirements of the system are elicited from the stakeholders. The requirements can be classified as functional and non-functional. Functional requirements specify

Software Architecture	**Affected Quality Attributes**
Layers	Testability, Performance
Pipes and Filters	Reliability, Performance, Usability
BlackBoard	Maintainability, Availability, Testability
Broker	Modifiability, Availability, Testability
Model-View-Controller	Modifiability, Performance
Presentation-Abstraction-Control	Modifiability, Performance
Microkernel	Modifiability, Maintainability, Reliability
Reflection	Modifiability, Performance

Table 2.3: Architectures and its affected Quality Attributes

what a system should be able to do. For example, mail client software is required to send and receive email. Non-functional requirements do not describe what a system should do but describes how it should do it. For example, a mail client software is required to respond to the users commands in 50 ms (performance) and it should protect the user from malicious email attachments (safety).

Since, the earliest architectural decisions are hardest to change later on in the project, it is very important for the stakeholders to prioritize the required qualities. One technique for this is a Quality Attribute Workshop (QAW) [4]. The result of this workshop is a ranked list of quality attributes that was gathered from the stakeholders.

Once the requirements are collected, a basic architecture is defined. It will address all the requirements, both functional and non-functional, from the stakeholders. Usually, a structural architectural pattern is chosen initially. Architectural patterns such as pipe and filter, blackboard, broker and Model-View-Controller describe the highest-level structure of an application. These patterns are used to achieve system requirements from the earliest stages of development. Suppose a client-server architectural pattern was chosen as the best architecture for a project. With that choice some quality attributes have been influenced. Client-server architecture improves the reliability of the system because a crash on either the server or client does not affect the other. But it introduces points of vulnerability and reduces security. Table 2.3 , illustrates the various qualities that are affected by each architectural pattern [15].

To ensure that the resulting product satisfies these qualities, appropriate design decisions must be made in the architecture. This method of satisfying a quality-attribute-response measure by manipulating some aspect of a quality attribute model through architectural design decisions is also called applying a architectural tactic [2]. Tactics may influence one or more quality attributes of a system. For example, redundancy is a tactic that is chosen to improve availability. Adding an intermediary between two modules that are highly coupled is a tactic that improves modifiability [18]. But tactics are not arbitrarily chosen, they are coupled with specific operational scenarios.

Scenarios with their associated attributes drive which tactics are selected. A scenario is always associated with a quality attribute; an example of a performance scenario is when a mechanical door needs to be open within 2 seconds. Performance tactics will be applied to satisfy that scenario and any tactic that reduces performance will be discarded. This process of identifying the driving quality attributes, refining use scenarios and applying tactics is also referred to as Attribute Driven Design (ADD).

The steps in ADD are the following.

- Choose a module to decompose.
- Identify the architectural drivers from the functional and non-functional requirements and build an architectural pattern with tactics
- Further decompose modules.

But a problem with ADD is that the software architect has to be familiar with all the quality-attributes identified during the QAW. It is difficult for a software architect to consider all the quality-attribute scenarios and check that the chosen architecture satisfies the quality attributes of the system. This is especially hard for composite quality-attributes such as dependability. Reasoning frameworks can be used to facilitate ADD by reducing the technical burden on the software architect.

2.3 Reasoning Frameworks

The software architecture of a system influences the satisfaction of the quality attributes requirements such as performance and modifiability. To determine whether an architecture satisfies the quality requirements, analytic theories are used. For example, Rate Monotonic Analysis (RMA) can be used to generate a task model that corresponds to an architectural description and from that we can estimate the latencies of the components. RMA can be used in a performance reasoning framework.

Reasoning Frameworks are built for the following reasons: [7]

- Predict behavior before the system is built

- Understand behavior after it is built
- Make design decisions while the system is being built and when it evolves

Analytic theories use a variety of representations to describe the aspects of a system about which they reason. Analytic constraints, the assumptions made before the theory is used, warn about the limitations of the analytic theory. The architecture is mapped to a model in a process termed interpretation, called interpretation, in order to represent the architecture in a way that can be analyzed or reasoned about. The procedure of analyzing or reasoning from the model is called the evaluation procedure.

In summary there are six elements in a reasoning framework [7]

- Problem description: the set of quality measures that can be calculated.
- Analytic theory: the foundations on which analyses are based. The disciplines include queuing theory, rate monotonic scheduling theory, finite state automata, and temporal logic.
- Analytic constraints: assumptions for using the theory.
- Model representation: a model of the architecture that is relevant to the analytic theory and acceptable for the evaluation procedure.
- Interpretation: a procedure that generates the model from the architectural descriptions.
- Evaluation procedure: algorithm or formulae that calculate estimates of the specific measures of a quality attribute from a model representation.

Each reasoning framework addresses a problem regarding a specific quality attribute. The problem description identifies the quality attribute for which the reasoning framework is used. As figure 2.3 indicates, the reasoning framework can answer two things.

- Does the architecture satisfy the desired quality measures (quality requirements)?
- What quality measures can be calculated?

A general scenario describes the problems that can be solved by a reasoning framework [6]. It can be used to describe the quality attribute requirement for one specific quality attribute. A general scenario has six parts.

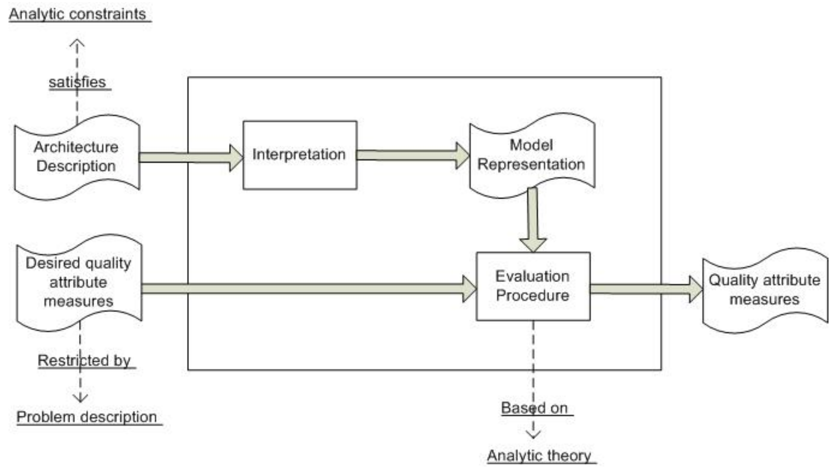


Figure 2.3: Reasoning Framework Diagram

- stimulus: the condition that arrives at the system.
- source of the stimulus: the entity that generated the stimulus.
- environment: the conditions in which the stimulus arrives.
- artifact: the entity affected by the stimulus.
- response: the activity by the artifact after the stimulus arrives.
- response measure: a quality-attribute specific constraint that must be satisfied by the response.

Concrete scenarios are derived from a general scenario. An example of a concrete scenario (for performance) is the following: If an obstacle (person or object) is detected by the garage door during descent, the door must halt within 0.1 seconds [2].

- Analytic theory

  The underlying principle for a reasoning framework is the use of an analytic theory to reason about a quality-attribute [7]. The model is represented to fit the theory and the evaluation procedure is the application of the theory to derive estimates of quality attribute measures.

- Analytic Constraints

  Analytic constraints limit the design space of the reasoning framework and thereby ensure the solvability and tractability of the problem. A co-refinement process can be used to incrementally relax the analytic constraints and improve the analytic theory or evaluation procedure until the analytic constraints allow the framework to be applied to an appropriate class of problems. An example of an analytic constraint for a performance reasoning framework is: “all event arrivals must be periodic.”

- Model Representation

  Models are abstractions which capture the information of the system that is relevant to the analysis being performed. The primary purpose of the model is as an input to an automated evaluation procedure. The model includes all the information that the evaluation procedure needs to calculate quality attribute measures. This information must be derived from the reasoning frameworks’ architecture description or concrete scenarios.

- Evaluation Procedure

  It is a deterministic procedure for calculating dependent parameters (quality attribute measures calculated using the reasoning framework) from the independent parameters (characteristics of the architecture that appear in the model representation)

2.4 ArchE Architecture Design Assistant

ArchE is a software tool that assists an architect in making architectural decisions. A model for ArchE is tax preparation software. The software may have complete knowledge about the tax code and have the ability to check for consistency, but the information on which the tax return is based and the correctness of the return depends on the information provided by the user [2].

There are three concepts associated with ArchE

- quality attribute scenarios. This concept was introduced in section 2.3
- reasoning frameworks. This concept was explained in section 2.3
- responsibilities. A responsibility is a property of a module in the module viewtype [92]. It describes the role of a module in a system. In ArchE, functional requirements are represented as a graph where the nodes represent responsibilities and the edges represent the relationships between the nodes. ArchE’s quality attribute scenarios show the quality requirements of the system. The reasoning framework converts the scenarios into a quality attribute-specific model. The models represent the design that can satisfy the requirements specified for that quality

attribute. The reasoning framework takes the model that satisfies all the quality attribute requirements and converts it into an architectural representation of the design.

Responsibilities can be decomposed into smaller responsibilities in order to make the granularity of the responsibility smaller and thus make it more specific and concrete as needed. Some types of decompositions are shown below.

Decomposition of responsibilities

- Responsibility A precedes B. When a responsibility A must be executed before B.
- Responsibility A intersects B. When the sub responsibilities of A have an intersection with those of B.

Responsibilities play a central role in reasoning frameworks and their specific roles are shown below.

Roles of responsibilities

- Applying a tactic introduces new responsibilities that must be treated by every reasoning framework relevant to any scenario. E.g. The tactic “apply scheduling strategy” results in the responsibility “schedule units of concurrency” which the modifiability reasoning framework must assign to a module [3].
- The relationship among responsibilities can guide the reasoning framework. If a responsibility is contained in another, this information can be used by the modifiability reasoning framework towards decomposing it into modules.
- The properties of responsibilities include the information that is necessary to create a design [3].

Responsibilities can come from scenarios, requirements not embodied in scenarios, design specifications, or the application of a tactic.

Here is an example of how responsibilities are derived from scenarios and the steps that are taken by ArchE to satisfy the scenarios. Consider two scenarios in the context of the same reasoning framework [3]:

- A garage door must detect an obstacle and halt within 0.1 seconds.

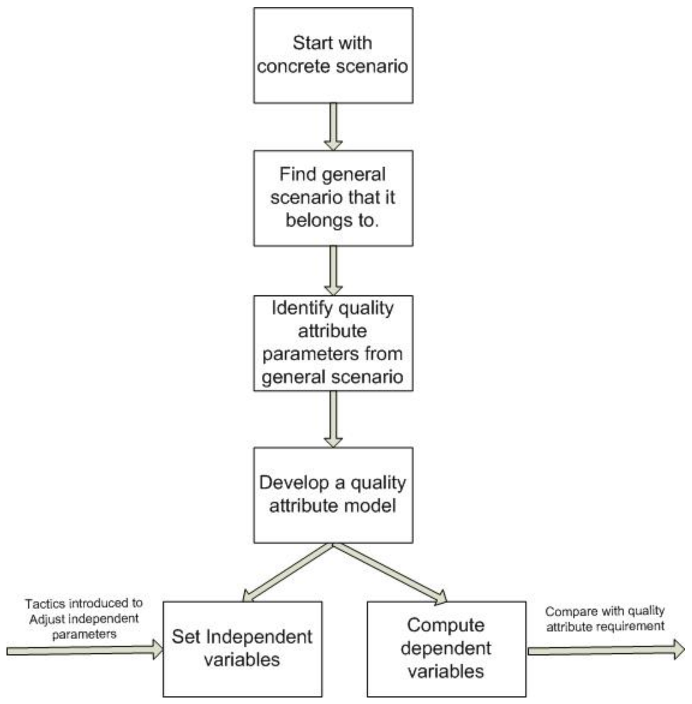

Start with concrete scenario
Find general scenario that it belongs to.
Identify quality attribute parameters from general scenario
Develop a quality attribute model
Tactics introduced to Adjust independent parameters
Set Independent variables
Compute dependent variables
Compare with quality attribute requirement

Figure 2.4: ArchE Process Diagram

- The garage door must respond to a user request to lower the door within 0.3 seconds

These are deadline scenarios which can be analyzed with a fixed-priority-scheduling reasoning framework and a cyclic-executive reasoning framework. ArchE will take the following steps to satisfy the scenarios [3].

- Detect responsibilities and their relationships
- Determine the reasoning framework.
- Determine the initial hypothesis for values of free parameters.
- Determine and apply tactics.

Step 1: Detect responsibilities and their relationships: By converting scenarios into a standard six-part form, responsibilities are derived. For example, “manage the garage door” responsibility yields these five responsibilities [3].

- Detect obstacle.

- Halt door.
- Detect user request to lower the door.
- Lower door.
- Do everything else involved in managing the garage door.

Step 2: Determine the reasoning framework: An appropriate reasoning framework must be chosen to analyze the scenario.

Step 3: Determine the initial hypothesis for values of free parameters: The designer assigns the execution time for each responsibility which are to be tested as hypothesis.

- Detect obstacle 10 ms
- Halt Door 15 ms
- Detect user request to lower the door 25 ms
- Lower door 50 ms
- Do everything else 300 ms

Step 4: Determine and apply tactics: The hypothetical results of applying a tactic are presented in this step. Tactics are introduced to adjust the independent parameters that are used to compute the dependent parameters which are then compared with the quality attribute requirements specified by the concrete scenarios.

2.5 Qualitative Reasoning

Qualitative reasoning is a technique that is used to reason when exact values, or some type of ordinal scale, are not available or not needed. A pot of water above a sufficiently hot stove will eventually come to a boil and evaporate. We can reason about such a scenario because we know from experience that, without exception, a pot above a lit stove always boils if the temperature is high enough. We could write the energy equations and prove this for specific values but qualitative reasoning allows us to address the general case at an appropriate level. Qualitative reasoning can be used to model such tacit knowledge that people use to reason about the world [24]. Tacit (or implicit) knowledge is also used in the design of software architectures. Architectural design decisions or tactics are an example of tacit knowledge. For a sub-attribute of dependability such as security, a reasoning framework based on qualitative theory might be more useful than a one based on a quantitative theory.

Qualitative reasoning [51] provides a means of making decisions involving attributes that can not be expressed quantitatively. Qualitative techniques do assume some type of ordinal scale. The reasoning rules use two fundamental characteristics: a current position on an ordinal scale and an indication of whether the attribute is changing its value and if so in which direction along the scale. For example, the security attribute of a piece of software might be rated on an ordinal scale as “very” secure and that recent architectural changes are making the software “more” secure. Qualitative reasoning supports building models that represent these relationships between qualitative values. These models support inferences about how the values change over time and how they cause other values to change.

Qualitative reasoning techniques allow the behavior of assemblies of concepts to be inferred from the behavior of individual pieces of the assembly. The behavior is described using qualitative rather than quantitative data either because the level of interest is to provide a conceptual description or because quantitative data is difficult or impossible to obtain.

Qualitative attributes take on values that lie on a qualitative scale. These values are defined as belonging to quantity spaces, i.e. data types. A quantity space is an alternating sequence of points and intervals. A point represents some significant milestone in the quantity space, the trigger to change from one steady state to another. An interval value groups together a set of quantitative values for which no change occurs at the qualitative level.

An attribute takes on an initial value that is a point in the quantity space. Any change to the system will result in the attributes value changing from the point into one of the adjacent intervals. Reaching a milestone value or leaving a milestone value represents time passing. Once in the interval, the attribute takes on different values in that interval but changes within the interval are not recognized by the qualitative reasoning system. At some point a milestone is reached and it can be inferred that the attribute has taken on the value of the next point.

For example, the point at which water changes from a liquid to a solid and the point at which it changes from liquid to a gas would be point values in a scale describing the temperature of water. Changing the temperature of water from 45 to 46 degrees is of no interest in a qualitative description about the behavior of water because the water is liquid at 45 degrees and still a liquid at 46 degrees. In other words, the change from 45 degrees to 46 degrees does not represent a change from one interval to another interval which would correspond to a change from one qualitative state to another. This representation of the scales allow the description of each scale to focus on only those aspects of interest.

Qualitative reasoning supports describing the effects of causation and time. Causation is modeled by influences and proportionalities. Influences model processes that cause changes within a model and proportionalities propagate the effects of a process. Applying a heat source to water causes its temperature to increase. The exact amount of heat is not known but it is a non-zero value that is high enough to cause the temperature of the water to rise. This is described as a positive influence of the heat on water. In some special cases only an increase in the amount of heat changes the temperature of the water. This can be modeled as a proportionality between the change in heat and the change in water temperature.

These techniques have been applied in the formation of a theory of qualitative physics [51]. This is a conceptualization of physics that captures both the structure and behavior of a system. This theory has been used to describe the behavior of many physical systems, such as a set of storage tanks, in which the behavior of the system is described in terms of the behaviors of individual connections among the constituent entities. Other domains, such as population dynamics have been described and the behavior of the systems has been simulated.

Chapter 3

Related Work

Previous research on software dependability has been largely about mechanisms to avoid or accept faults in software. Many mechanisms to improve software safety, availability and reliability can be found in the literature of real-time systems [90, 59, 62, 85].

To improve reliability, techniques such as n-version programming, recovery blocks, formal verification and static analysis have been proposed. Watchdog timers have been used to improve availability and fault tree analysis have to used to derive run-time assertions that can be used to improve safety [62].

There have been efforts to predict the quality attributes of software architectures. Some have concentrated on specific quality attributes such as availability [85], reliability [78] and security [21]. ArchE also provides quantitative quality attribute reasoning frameworks for modifiability and performance as examples [18]. There is a lot of knowledge about the quality-attributes that make up dependability but the knowledge has not been consolidated into a reasoning framework for dependability.

There has been much debate over the correctness of dependability prediction because the models have made inaccurate assumptions about the failure occurrence and their distribution [62]. But some research [37] indicates that it is not necessary to have exact calculations to identify software architecture with better dependability and therefore the approach to use reasoning frameworks to improve dependability can be a promising approach.

The inclusion of confidentiality and integrity into the definition of dependability makes it important to consider qualitative models in order to create a complete reasoning framework for dependability. But to our knowledge, no work as been done to qualitatively reason about quality-attributes. Given the disparate nature of the related work of dependability, a more detailed presentation of the related works will be given in the introductions of each chapter.

Chapter 4

Quantitative Reasoning Frameworks

The goal of this work is to create a dependability reasoning framework to help increase the dependability of software systems. This is because more and more critical requirements are placed on software systems and therefore there is a growing need to reason about dependability during architectural design.

The knowledge for a reasoning framework in dependability has not been organized very effectively. There is also a problem of blending the sub-attributes of dependability into one reasoning framework. To build a reasoning framework for dependability, there must first be a reasoning framework for each sub-attribute of dependability. This means that a reasoning framework for dependability will be a combination of reasoning frameworks for reliability, availability, confidentiality, integrity and maintainability. Attributes such as reliability, availability and maintainability can be based on analytic theories but confidentiality and integrity will be based on qualitative theories. In this chapter I will present a reasonably complete framework for reliability and sketch how a qualitative framework for security might be developed.

The example software architecture used to derive the quality-attribute scenarios is called CTAS (Clemson Transit Assistant System). It is a telematics product designed to run on handheld devices [68]. Telematics software helps the user retrieve itinerary information from various information providers. CTAS is an itinerary planning system that plans routes and modes of transportation

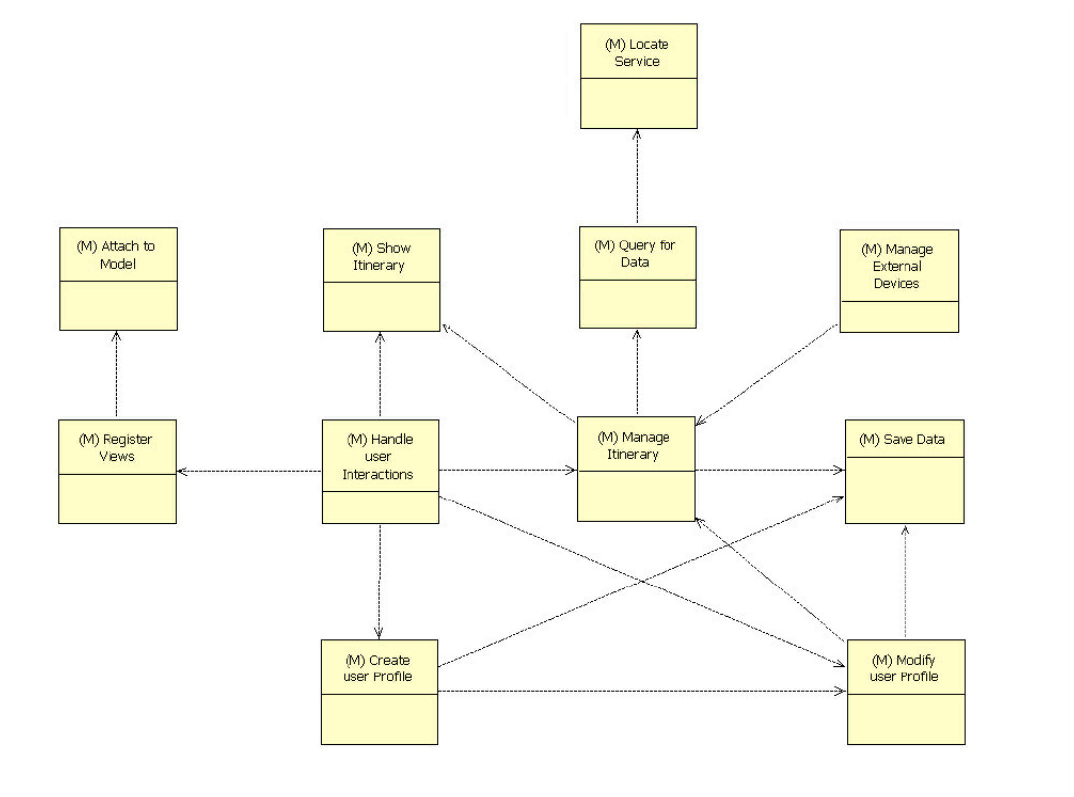


Figure 4.1: CTAS Architecture (Dependency View of Modules)

from one place to another. CTAS can be executed on various platforms including hand-held devices. The hand-held devices add GPS and use wireless connections to check schedules, make reservations, and check the status of various modes of transportation. Developers can develop plug-ins that integrate appropriate technologies to provide various services through CTAS. An architectural view of CTAS is shown in figure 4.1.

For attributes of dependability such as availability, reliability and maintainability there are well known analytic models that try to predict its values from the software architectural level. Reasoning frameworks based on quantitative theories will be referred to as quantitative reasoning frameworks.

A complete reasoning framework for dependability requires reasoning frameworks for availability, reliability and maintainability. In this chapter, the necessary theory that is required for a reasoning framework for reliability is illustrated. A similar approach is required for availability and maintainability.

4.1 Reliability

Reliability is the measure of the probability of failure-free operation for a specified time. It is often represented in terms of failures per hour (failure intensity) or by the probability of failure [73]. More specifically, reliability can be separated into two different types: a *perceived reliability* and an *actual reliability*.

The perceived reliability is the reliability perceived by the user. The system can have parts where it has numerous errors but if the user avoids those parts and uses the system where it touches only parts with few or no errors, then the perceived reliability will be very high. The actual reliability is a theoretical value related only to what defects are present in the system and not how it is used. The user may use the system in ways that avoid the errors in it but the system will be judged to have low actual reliability because it contains many errors regardless of the use profile.

It is usually infeasible to find the actual reliability of the system because the number of test cases to find all the errors in a system is usually unbounded. But finding the perceived reliability of a system, which depends largely on the operational profile, is possible. The operation profile is the quantitative characteristic of how the software will be used [83].

4.1.1 Analytic Theory

The analytic theory behind the reliability reasoning framework is the work on software architecture based reliability modeling [31]. A software architecture based reliability modeling that closely mirrors the scenario based reliability analysis is the model introduced by [30]. In that model, the architecture is expressed as a Discrete Time Markov Chain (DTMC) where the states represent the components of the software system. State-based reliability analysis models can include a correct output state or a failure state. Such models are called composite models but a hierarchical model, which solves first the architectural model and then superimposes the failure behavior on the solution, is better suited for the scenario based approach to reliability analysis [31].

The reliability of a component in the model by *Gokhale et al.* is expressed with the equation below.

$$R_i = e^{-\int_0^{V_i t_i} \lambda_i(t)dt} \approx e^{-a_i c_i(t_i)} \tag{4.1}$$

The time-dependent failure intensity $\lambda_i(t)$, the number of times a component is passed

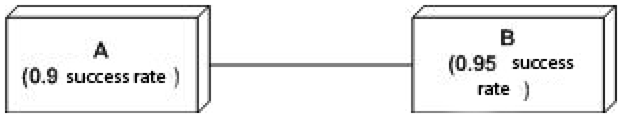


Figure 4.2: A Serial Connection

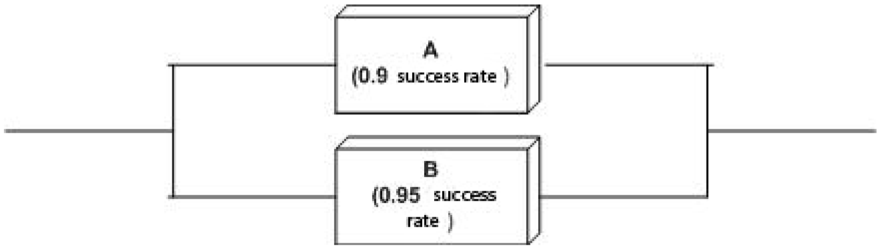


Figure 4.3: A Parallel Connection

through, V_i, and the time spent in component t_i determines the reliability of a component. To simplify the calculation, the number of faults in the component a_i and the expected behavior of the code coverage resulting from executing the application with successive test cases $c_i(t_i)$ is used to calculate the reliability of the component [28]. The term $a_i c_i(t_i)$ is known as the mean value function of the component i which shows the average cumulative failures associated with time point t_i [73].

Based on the estimated reliabilities of each component, the reliability of the overall system can be generalized to:

$$R = \prod_{i=1}^{n} R_i \tag{4.2}$$

But when calculating the system reliability of an execution path through a software architecture, serial and parallel connections need to be considered separately. This is because the inputs to the computation will accumulate through an execution path. If a failure in one part causes the entire sequence of events to fail, then the connection is considered to be serial.

The overall reliability for the serial connection in figure 4.2 will be equal to $0.9 \times 0.95 = 0.855$

In a parallel connection such as the one shown in figure 4.3, there is a failure when both paths fail.

The overall reliability for this parallel connection will be equal to $1((1-0.9)+(1-0.95)) = 0.85$ But since software reliability is usually stated in terms of the executed time, it is convenient to

Source of Stimulus	**User**
Stimulus	User request
Environment	Normal operation
Artifact	Storage, computation algorithm
Response	Return itinerary to user
Response Measure	Return itinerary to user with a reliability of 0.95

Table 4.1: A Reliability Scenario

convert the reliability of each software component into a failure intensity which is represented by λ [73]

$$\lambda = -\frac{1}{t}\ln R \text{ (t is the execution time)}$$

Operational profiles can be expressed with the utilization rate. Utilization expresses in percentages how much of a component is in use. When utilization is taken into account, the reliability is the following:

$$\lambda = \rho\lambda \text{ (ρ is average utilization)}$$

4.1.2 Model Representation

To reason about the reliability of a software architecture, a concrete reliability scenario must be stated. The premise of the reliability reasoning framework is that the perceived reliability of the system depends on how reliable the system is for each scenario in a system. An example reliability scenario for CTAS (Clemson Transit Assistant System) is the following: When a user requests a new itinerary 95% of the time it will be delivered correctly. The details of this scenario are shown in Table 4.1:

A reliability scenario closely mirrors an execution path through a software system. To model an execution path, use cases or activity diagrams can be used. There is a natural correspondence between use cases and responsibilities, but depending on the required granularity, activity diagrams can be used to derive the responsibilities. Figure 4.4 illustrates the use cases and figure 4.5 illustrates an activity diagram of CTAS. Each node in a use case diagram or an activity diagram can become a responsibility. Each responsibility must have a known reliability in order to compute the system reliability. In addition to the reliability, the utilization rate for the scenario should be given as a parameter. The utilization rate influences the reliability of the overall system depending on the operational profile.

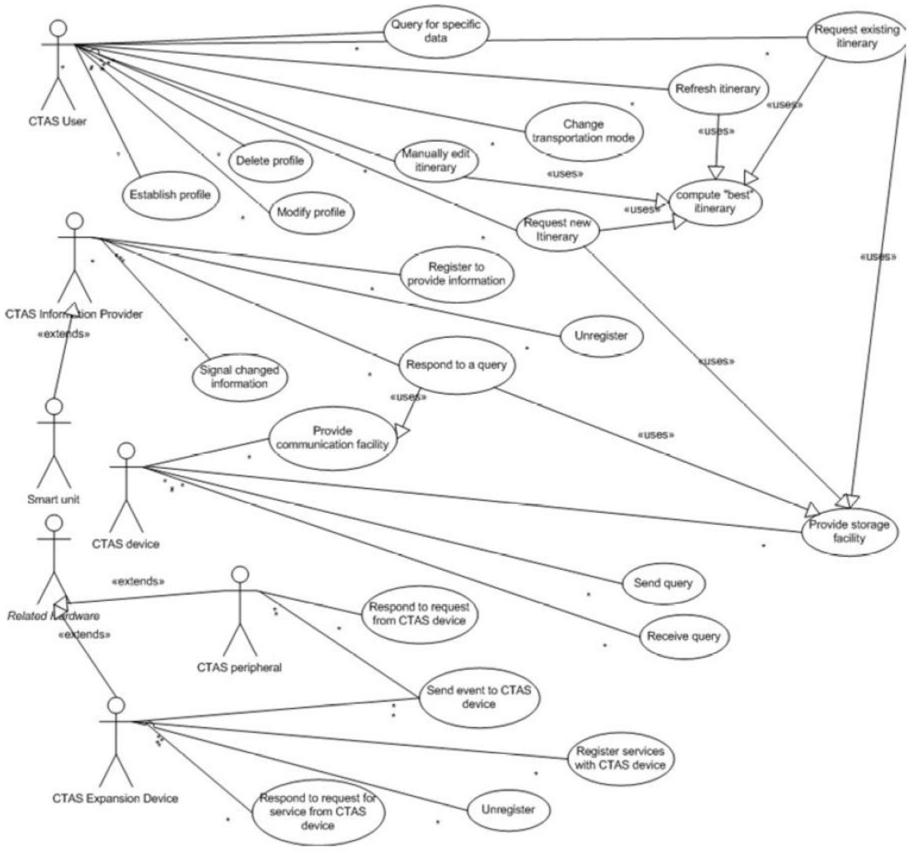


Figure 4.4: Use Case Diagram for CTAS (Clemson Trasit Assistance System)

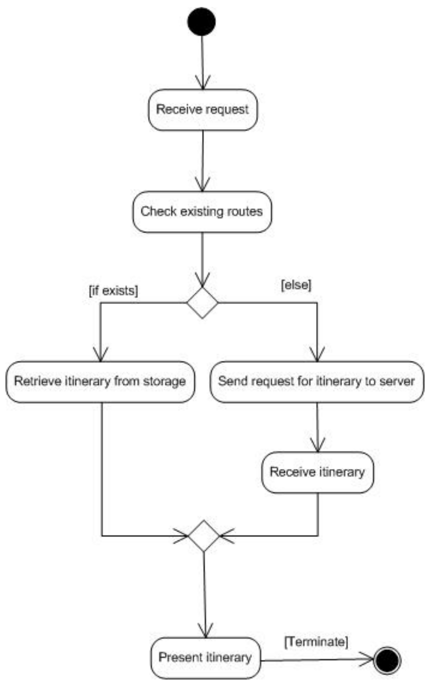


Figure 4.5: A CTAS Activity Diagram

4.1.3 Interpretation

The reliability modeling method by Gokhale et al. can be applied to calculate the reliability of a scenario if the components in the model are generalized into responsibilities. Instead of components having a reliability measure, the responsibilities will have a reliability measure. The relationships among the responsibilities closely mirrors the DTMC of the Gokhale model. There are three types of relationships in a quality scenario.

- contains: indicates that one or more children responsibilities are contained within a responsibility.
- dependency: indicates that a responsibility is dependent on another responsibility.
- sequence: indicates that a first responsibility must be executed prior to the second responsibility.

Depending on the types of the relationships that exist among the responsibilities of a scenario, the calculation of the overall reliability of the scenario will change. How the calculations change for each relationship is listed below.

- contains: the reliability of the child node determines the reliability of the parent. For example, if there are two children nodes (assuming a parallel relationship) with reliabilities of 0.95 and 0.93, the reliability of the parent node is $\frac{(0.95+0.93)}{2} = 0.94$

- dependency: when there is a dependency between two nodes, the overall reliability of the two nodes is the product of the reliabilities of the two nodes.

- sequence: when two nodes form a sequence, the overall reliability of the two nodes is the product of the reliabilities of the two nodes.

The parameters required by the reliability reasoning framework for the functional responsibilities are the following:

- Use Rate - Each scenario will be given a percentage of usage to express the operation profile. The general system reliability will be calculated based on the operation profile as expressed with these values.

- Reliability Rate - The architect should provide an estimate of the reliability expressed as a probability for each responsibility. These values can be calculated by estimating failure rates and coverage levels for testing.

An important point about these parameters is the fact that they represent estimates. Since the reasoning framework for reliability is used for the prediction of the reliability of a system prior to building it, the parameters can only be estimates derived by the architect.

4.1.4 Evaluation Procedure

To better illustrate an example of the use of the reliability reasoning framework, consider the following reliability scenario from CTAS: **The system must be able to show an itinerary with a reliability of 0.97**

- stimulus: show itinerary

- source of the stimulus: the user

- environment: normal conditions

- artifact: system

Responsibilities	**Reliability**
Handle user interactions	0.96
Manage itinerary	0.95
Query for data	0.92
Locate service	0.90
Save data	0.99
Show itinerary	0.99

Table 4.2: Responsibilities and Reliability

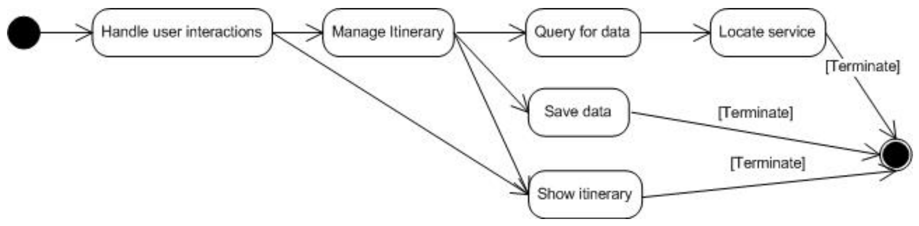


Figure 4.6: Relationships in a Scenario

- response: itinerary is shown
- response measure: a reliability of 0.97

In the next step, the functions that are used by the scenario must be specified. A function is a description of one of the features that the system being designed should satisfy [3].

The functions are then mapped to a responsibility. A responsibility is an activity that the system being designed should perform or information that it should remember [3]. In this example, all the functions are mapped to the the same responsibility. The responsibilities and the failure rate for each responsibility is shown in table 4.3.

The next step is to have a mapping from a scenario to one or more responsibilities. For example, in a scenario where a developer is able to add a new variable to the user profile in five days of effort, there are two responsibilities that are mapped to it: “modify user profile” and “create user profile”.

The relationships between the responsibilities are then specified. This is where the execution path of the scenario is shown. Figure 4.6 indicates the relation between the responsibilities of the execution path. The arrows indicate a dependency between the responsibilities.

When the user enters all the relevant information, such as the use rate and the reliability rate for each responsibility, the reasoning framework can determine if the scenario can be satisfied or not. To calculate the reliability of the scenario, the reliability of the responsibilities that are mapped to the scenario must be calculated. All the reliabilities in figure 4.6 varies depending on the internal structure of each responsibility and the relationships that exists among the responsibilities is used to derive the total reliability of the scenario. For each relationship the following rule applies when the reliabilities are calculated.

- contains: the reliabilities of the children responsibilities are ignored in lieu of the parent responsibility.
- dependency: the overall reliability of a dependent relationship is a multiplication of the two involved components. (serial connection rule)
- sequence: it is the same as a dependency relationship.

Figure 4.6 shows the relationships of responsibility "Handle User Interaction" that is mapped to the scenario. All the arrows indicate a dependency and there is a total of four paths through the scenario. The paths and the reliability per path are shown below.

- Path 1: Handle user interactions (0.96) $--->$ Show itinerary (0.99) $==>$ Reliability ($0.96 \times 0.99 = 0.95$)
- Path 2: Handle user interactions (0.96) $--->$ Manage itinerary (0.95) $--->$ Show itinerary (0.99) $==>$ Reliability ($0.96 \times 0.95 \times 0.99 = 0.90$)
- Path 3: Handle user interactions (0.96) $--->$ Manage itinerary (0.95) $--->$ Save Data (0.99) $==>$ Reliability ($0.96 \times 0.95 \times 0.99 = 0.90$)
- Path 4: Handle user interactions (0.96) $--->$ Manage itinerary (0.95) $--->$ Query for data (0.92) $--->$ Locate service (0.90) $==>$ Reliability ($0.96 \times 0.95 \times 0.92 \times 0.90 = 0.76$)

The correct operation of the scenario depends on all the paths and therefore to derive the reliability of the scenario the reliability of all the paths must be multiplied. Therefore, the total reliability is $0.95 \times 0.90 \times 0.90 \times 0.76 = 0.58$

Calculating the reliability of the overall system requires the calculation of the operating profile for a user. If the user uses the "show itinerary" scenario 30% of the time and the rest of the system is used 70% of the time with the reliability of 0.95, the general system reliability is

$$R = (0.58 \times 0.30) + (0.95 \times 0.70) = 0.839$$

The calculated reliability for the scenario is 0.58 and the goal was 0.97, which means that the reliability criteria for the scenario has not been met. An implementation of the reliability reasoning framework should produce the same result.

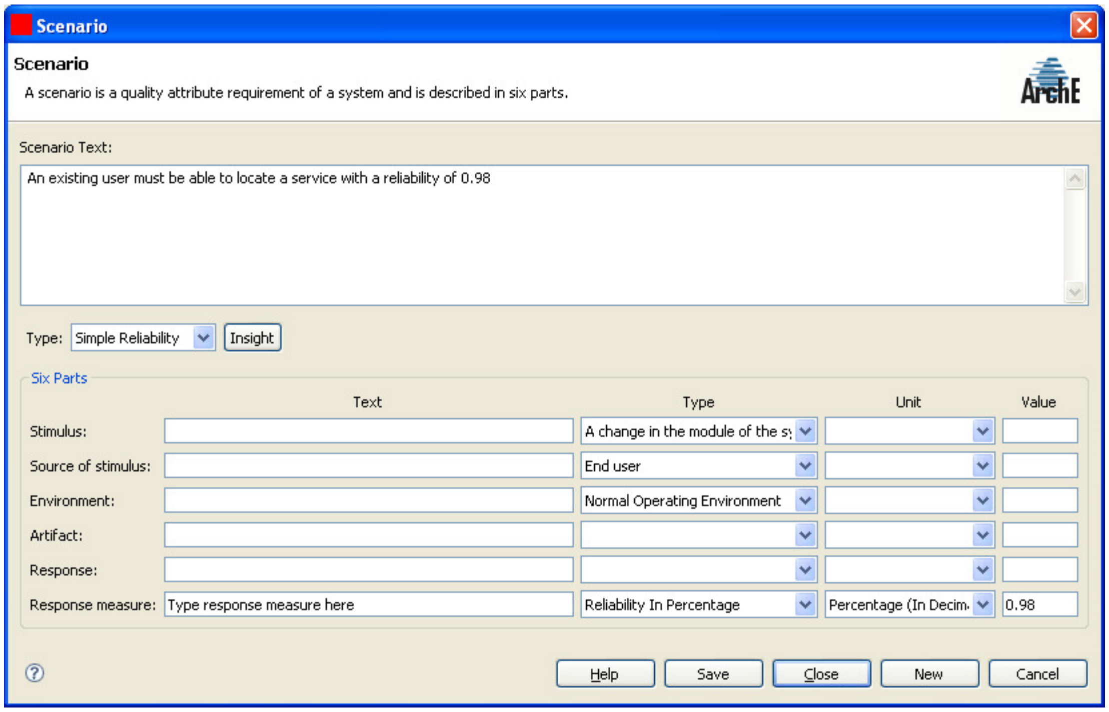
In summary, here are six elements in the reliability reasoning framework.

- Problem description: the estimation of reliability for a reliability scenario and the overall reliability based the operational profile.
- Analytic theory: software architecture based reliability analysis.
- Analytic constraints: the states of the modeled software architecture are the components of the system.
- Model representation: the software architecture is modeled as a Markov chain where the there is a clear start and termination.
- Interpretation: the reliability scenario's relationship among its responsibilities shows a Markov chain that can be used as a model.
- Evaluation procedure: the reliability of each component in the model is calculated with the number of failures and the coverage level of testing as input parameters. The reliability of the modeled scenario is evaluated as a Markov chain.

4.1.5 Implementation

In this section, a simple preliminary version of the reliability reasoning framework is presented. It is implemented as a plugin for ArchE (architecture assistant) and runs as an Eclipse application. The preliminary version of the reliability reasoning framework does not cover all possible cases of connections among components but it shows the basic concepts that can be used to build the complete implementation.

This example illustrates how a quantitative reasoning framework can be implemented as a plugin for ArchE. The same approach will be taken to build plug-ins for the remaining quality attributes of dependability. The goal is to deliver a working implementation of a reasoning framework for dependability that combines the results from the quantitative and qualitative attributes of dependability.


Scenario

Scenario

A scenario is a quality attribute requirement of a system and is described in six parts.

Scenario Text:

An existing user must be able to locate a service with a reliability of 0.98

Type: Simple Reliability

Six Parts

	Text	Type	Unit	Value
Stimulus:		A change in the module of the sy		
Source of stimulus:		End user		
Environment:		Normal Operating Environment		
Artifact:				
Response:				
Response measure:	Type response measure here	Reliability In Percentage	Percentage (In Decim	0.98

Figure 4.7: Reliability Scenario

The first step in applying the simple reliability reasoning framework is to define concrete scenarios for the reasoning framework and enter them in the scenario editor as shown in figure 4.7. As an example, consider the scenario “An existing user must be able to locate a service with a reliability of 0.98”. This scenario is derived from the dependency view of CTAS from [18].

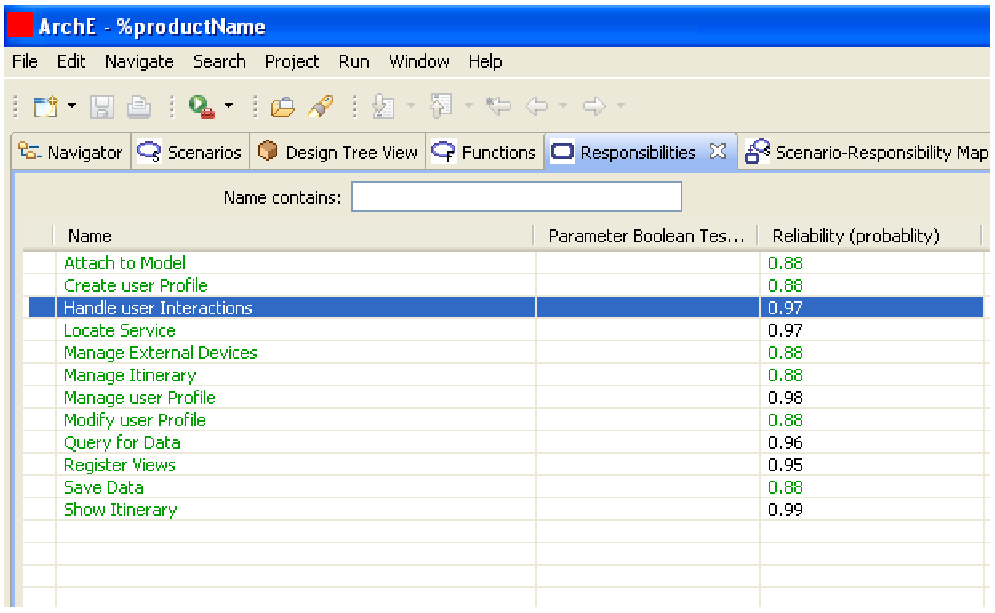
The most important field in the simple reliability scenario is the response measure value. This is the quality attribute value the system under design must achieve to satisfy the scenario. For this example, we will set the desired reliability to be (0.98).

The second step is to specify the responsibilities and their reliabilities as shown in figure 4.8. ArchE uses a responsibility-driven approach in which a responsibility of the system is synonymous with a component in the architecture. The reliability for each responsibility is expressed as a probability of failure free operation. These values can be derived by the software architect using the equation from the Gokhale model.

The third step is to map each scenario to one or more responsibilities. In the reliability reasoning framework, each scenario is mapped to one or more responsibilities that represent the initial system components that handle the scenario. The mapping is shown in figure 4.9. Other

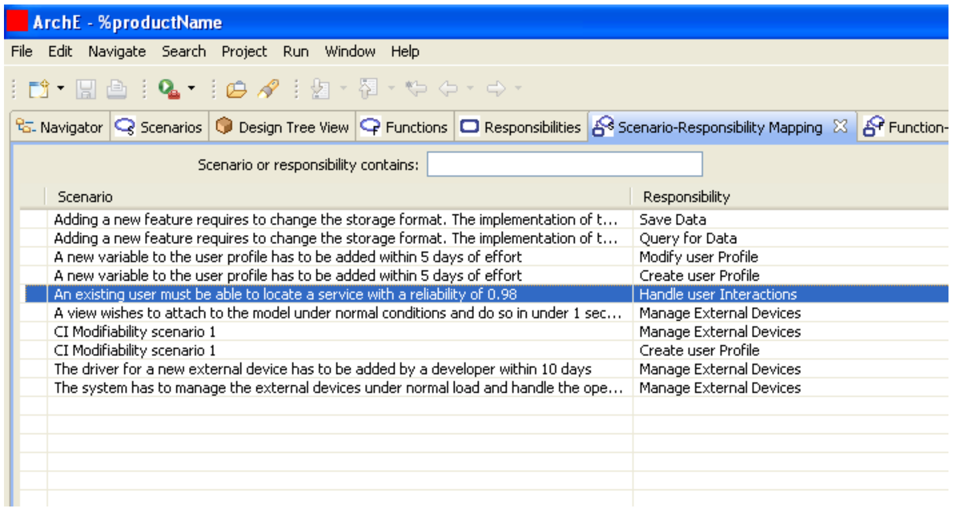
Responsibility	Reliability Rate
Handle User Interactions	0.97
Manage User Profile	0.98
Query For Data	0.96
Locate Service	0.97

Table 4.3: Reliability Rate for the Responsibilities in Scenario



Name	Parameter Boolean Tes...	Reliability (probablity)
Attach to Model		0.88
Create user Profile		0.88
Handle user Interactions		0.97
Locate Service		0.97
Manage External Devices		0.88
Manage Itinerary		0.88
Manage user Profile		0.98
Modify user Profile		0.88
Query for Data		0.96
Register Views		0.95
Save Data		0.88
Show Itinerary		0.99

Figure 4.8: Responsibilities and their reliabilities


ArchE - %productName
File Edit Navigate Search Project Run Window Help
Navigator Scenarios Design Tree View Functions Responsibilities Scenario-Responsibility Mapping Function-
Scenario or responsibility contains:

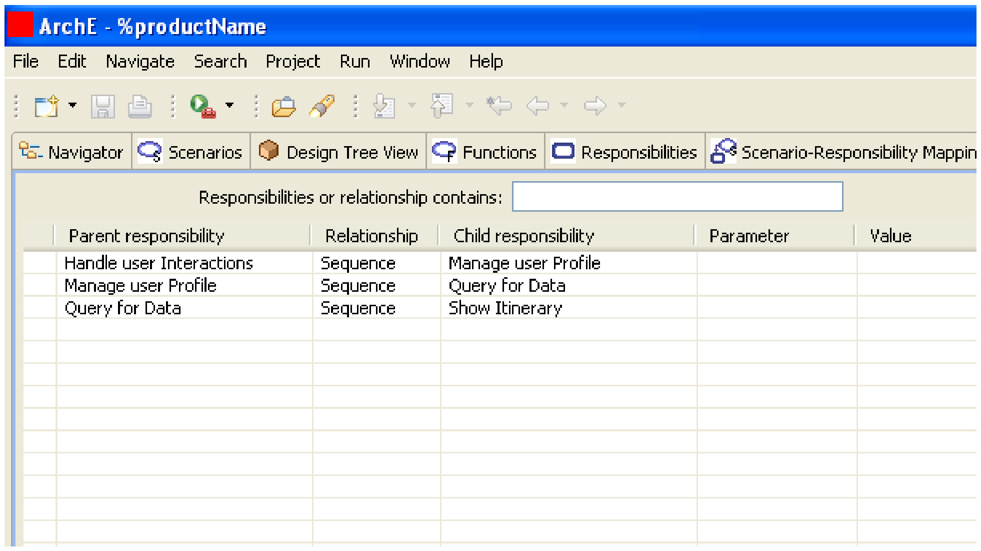
Scenario	Responsibility
Adding a new feature requires to change the storage format. The implementation of t...	Save Data
Adding a new feature requires to change the storage format. The implementation of t...	Query for Data
A new variable to the user profile has to be added within 5 days of effort	Modify user Profile
A new variable to the user profile has to be added within 5 days of effort	Create user Profile
An existing user must be able to locate a service with a reliability of 0.98	Handle user Interactions
A view wishes to attach to the model under normal conditions and do so in under 1 sec...	Manage External Devices
CI Modifiability scenario 1	Manage External Devices
CI Modifiability scenario 1	Create user Profile
The driver for a new external device has to be added by a developer within 10 days	Manage External Devices
The system has to manage the external devices under normal load and handle the ope...	Manage External Devices

Figure 4.9: Mapping from Scenario to Responsibility

responsibilities that are not directly related to a quality attribute scenario will also be identified during this process.

The fourth step is to define the relationships among the responsibilities of the scenario. The responsibilities that are part of the scenario under consideration, and their reliabilities, are presented in table 4.3. The relationship among the responsibilities in this scenario is called "sequence" and it is used to express a connection between two responsibilities in which the output of one responsibility is the input to the second responsibility. The responsibility "Handle User Interactions" is mapped to the scenario as the entry point to the architecture. The "Handle User Interactions" responsibility is the first node of a serial connection made up of three other responsibilities shown in figure 4.11 and it is entered in ArchE as shown in figure 4.10.

Finally, the result of analyzing the scenario is shown in figure 4.12. The red ball indicates that the scenario has not been satisfied based on the reliability of the responsibilities in the sequence.


ArchE - %productName
File Edit Navigate Search Project Run Window Help
Navigator
Scenarios
Design Tree View
Functions
Responsibilities
Scenario-Responsibility Mappin

Responsibilities or relationship contains:

Parent responsibility	Relationship	Child responsibility	Parameter	Value
Handle user Interactions	Sequence	Manage user Profile		
Manage user Profile	Sequence	Query for Data		
Query for Data	Sequence	Show Itinerary		

Figure 4.10: The Relationships among the responsibilities

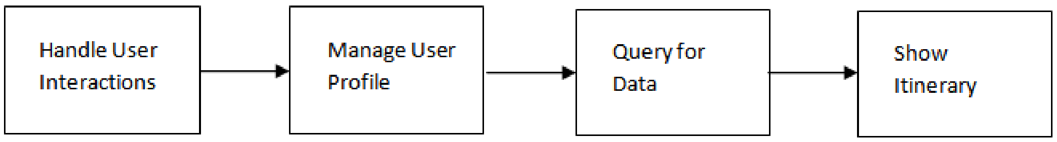
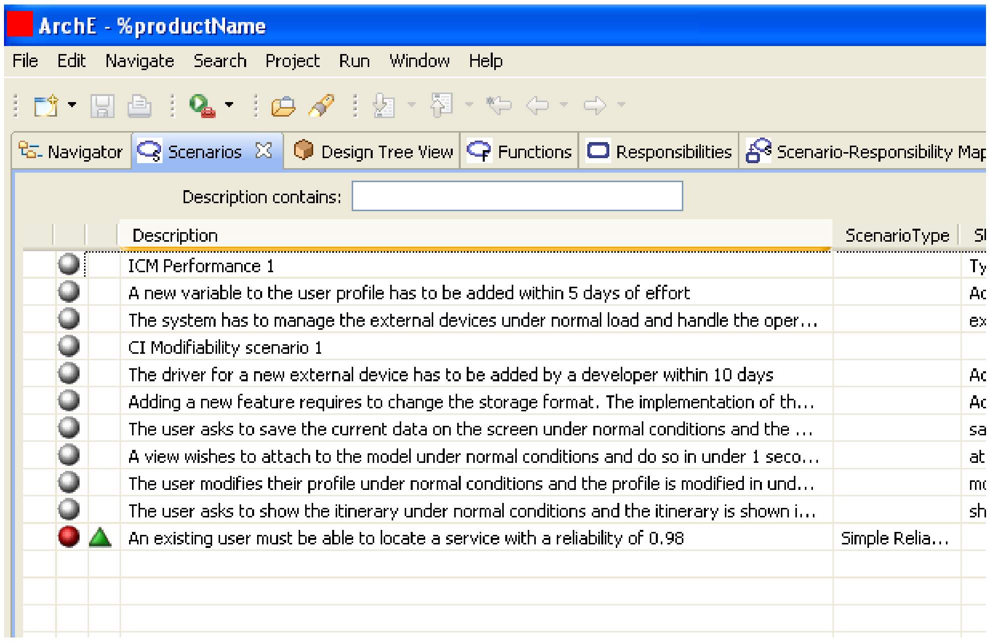


Figure 4.11: Sequence of responsibilities



		Description	ScenarioType	S
●		ICM Performance 1		Ty
●		A new variable to the user profile has to be added within 5 days of effort		Ac
●		The system has to manage the external devices under normal load and handle the oper…		ex
●		CI Modifiability scenario 1		
●		The driver for a new external device has to be added by a developer within 10 days		Ac
●		Adding a new feature requires to change the storage format. The implementation of th…		Ac
●		The user asks to save the current data on the screen under normal conditions and the …		sa
●		A view wishes to attach to the model under normal conditions and do so in under 1 seco…		at
●		The user modifies their profile under normal conditions and the profile is modified in und…		mo
●		The user asks to show the itinerary under normal conditions and the itinerary is shown i…		sh
●	▲	An existing user must be able to locate a service with a reliability of 0.98	Simple Relia…	

Figure 4.12: Analysis result of the Scenario

4.2 Availability

Availability is defined as a value that expresses the degree to which a system is "operational and accessible" when required for use [52]. It is an operational quality attribute closely related to reliability. Much work has been done on analyzing availability at the architectural level [6].

Availability is closely related to reliability but a system need not be highly reliable in order for the system to be highly available. The availability of a system can be high even if the reliability of the system is low if the system can be recovered and restored quickly. The reasoning framework for availability is very similar to the reasoning framework for reliability (mainly that it would be quantitative). In this section, the elements of a availability reasoning framework are presented.

4.2.1 Problem Description

Availability is an operational quality attribute and it is closely related to reliability by building on top of it the notion of recovery [84] and similarily the elements of a general availability scenario are the following:

- Stimulus: an user tries to operate a system.
- Source of the stimulus: a user.
- Environment: at runtime.
- Artifact: the software system.
- Response: indication of whether the system is able to satisfy the user's request.
- Response measure: the probability value that indicates the downtime of the sytem.

Since the only difference between availability and reliability is the notion of recovery, it is important to calculate the MTBF (Mean time between failure) and MTTR (Mean time to repair).

4.2.2 Analytic Theory

An order to calculate an availability value for a scenario, the availability of each responsibility must first be identified. In order to do this, we must first assign a MTBF and MTTR for each responsibility. For some responsibilities, the concept of MTBF and MTTR will not apply. But for any responsibility that can fail and then be recovered, the steady-state availability (availability over a sufficently long time) is calculated using the following equation [84].

$$\alpha = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

When the steady-state availability value is determined for all responsibilities of a scenario, the calculation for availability is similar to the calculation for reliability as shown in section 4.1.1.

4.2.3 Analytic Constraints

At a conceptual level, it is important to calculate assign MTBF (mean time between failures) and MTTR (mean time to repair) for each responsibility but there is going to be an inherent inaccuray in estimating MTBF and MTTR as these estimates are based on a system that has not been implemented yet. There is also going to be inaccuracy in terms of accounting for how much actual use will take place. This is important as the amount of use can significantly influence availability.

4.2.4 Model Representation

The model representation for the availability reasoning framework uses the same representation as used for availability. This is because both quality attributes are run-time attributes where the connections represent control flows.

4.2.5 Interpretation

An availability scenario is mapped to a set of responsibilities in the architecture just like a reliability scenario. The same types of relationships between responsibilities found in reliability scenarios are also found in availability scenarios. Essentially the only difference between the interpretation process between availability and reliability is that in availability the MTTR value of each responsibility needs to be estimated.

The failure rate λ is calculated from the reliability value as mentioned in section 4.1 and equation 4.3 shows how to calculate the MTBF from it.

$$\text{MTBF} = \frac{1}{\lambda} \tag{4.3}$$

Responsibilities	MTBF	MTTR	MTBF in % MTTR in % Availability
Handle user interactions	0.96	0.001	0.99
Manage itinerary	0.95	0.001	0.99
Query for data	0.92	0.01	0.98
Locate service	0.90	0.01	0.98
Save data	0.99	0.001	0.99
Show itinerary	0.99	0.001	0.99

Table 4.4: Responsibilities and Availability

The MTTR value is estimated from the time it takes to recover from a failure. MTTR can be the value it takes to reboot the system from a a failure caused by a responsibility or it can be the time it takes to replace the responsibility.

4.2.6 Evaluation Procedure

The evaluation procedure for availability mirrors the evaluation procedure for reliability. The only difference is that the availability value is used instead of the reliability value.

To better illustrate an example of the use of the availability reasoning framework, consider the following availability scenario from CTAS: **The system must be able to show an itinerary with a availability of 0.9**

- stimulus: show itinerary
- source of the stimulus: the user
- environment: normal conditions
- artifact: system
- response: itinerary is shown
- response measure: a availability of 0.9

In the next step, the functions that are used by the scenario must be specified. A function is a description of one of the features that the system being designed should satisfy [3].

The responsibilities and the availability value for each responsibility is shown in table 4.4 (The availability estimates are with respect to a 24 hour time frame). The execution path of the scenario is shown in figure 4.6 and it indicates the same relation between the responsibilities of the execution path as seen previously when analyzing the reliability scenario. The arrows indicate a dependency between the responsibilities.

Figure 4.6 shows the relationships of responsibility “Handle User Interaction” that is mapped to the scenario. All the arrows indicate a dependency and there is a total of four paths through the scenario. The paths and the availability per path are shown below.

- Path 1: Handle user interactions (0.99) $--\ >$ Show itinerary (0.99) $==>$ Availability ($0.99 \times 0.99 = 0.98$)
- Path 2: Handle user interactions (0.99) $--\ >$ Manage itinerary (0.99) $--\ >$ Show itinerary (0.99) $==>$ Availability ($0.99 \times 0.99 \times 0.99 = 0.97$)
- Path 3: Handle user interactions (0.99) $--\ >$ Manage itinerary (0.99) $--\ >$ Save Data (0.99) $==>$ Availability ($0.99 \times 0.99 \times 0.99 = 0.97$)
- Path 4: Handle user interactions (0.99) $--\ >$ Manage itinerary (0.99) $--\ >$ Query for data (0.98) $--\ >$ Locate service (0.98) $==>$ Availability ($0.99 \times 0.99 \times 0.98 \times 0.98 = 0.94$)

The correct operation of the scenario depends on all the paths and therefore to derive the availability of the scenario the availability of all the paths must be multiplied. Therefore, the total availability is $0.98 \times 0.97 \times 0.97 \times 0.94 = 0.86$

Calculating the availability of the overall system requires the calculation of the operating profile for a user. If the user uses the “show itinerary” scenario 30% of the time and the rest of the system is used 70% of the time with the availability of 0.95, the general system availability is

$$A = (0.86 \times 0.30) + (0.95 \times 0.70) = 0.923$$

The calculated availability for the scenario is 0.923 and the goal was 0.9, which means that the availability criteria for the scenario has been met. An implementation of the availability reasoning framework should produce the same result.

4.2.7 Discussion of Technique

The availability reasoning framework is a quantitative reasoning framework where specific numeric values are calculated. One limitation of the availability reasoning framework is that the MTBF and MTTR can be influenced by the pattern of usage or the amount of usage. Because of this, there can be a considerable difference between perceived and actual availability. Also, the availability value will be different depending on the time frame under consideration. These two factors have to be taken account in the intrepretation of the availability analysis.

4.2.8 Related Work

There have been efforts to implement an availability reasoning framework for embedded systems such as in [85]. Previous work on the availability reasoning framework was focused on how the prediction of availability based on technologies such as the watch dog timer. In our reasoning framework, we focus on the architectural level where the analysis is more abstract.

4.3 Maintainability

Maintainability is defined as the capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in the environment and in the requirements and functional specifications. Maintainability is related to modifiability but the main difference between maintainability and modifiability is that maintainability includes the correction of errors [9]. Currently, there exists a reasoning framework for modifiability [20]. In this section, a reasoning frameowork for maintainability is presented.

4.3.1 Problem Description

The maintainability reasoninng framework calculates the man hours that are required to perform a maintenance task based on a software architecture description. The general maintainability scenario describes a situation where a part of the system is maintained. There are four different types of maintenance activities. Here are the definitions for the different types of maintenance activities [87],[16].

- Preventive maintenance: to prevent latent faults from becoming apparent faults.
- Perfective maintenance: to perfect the system in terms of its performance, processing efficiency, or maintainability.
- Adaptive maintenance: to adapt the system to changes in its data environment or processing environment.
- Corrective maintenance: to correct processing, performance, or implementation failures of the system.

The maintainability reasoning framework takes these various maintenance tasks and calculates the man-hours required to perform them based on the structure of the architecture and values

4.3.2 Analytic Theory

The analytic theory for the maintenance reasoning framework is based on change impact analysis [1]. Change impact analysis is used to analyze the impact of a change on the responsibilities of the system and calculate how much effort is required to perform that change. In the context of calculating a maintenance task, three variables are required to calculate the required effort.

The first variable is size of the responsibility, this can be calculated in terms of source lines of code (sloc). The size is usually derived by using an effort analogy or using a model such as COCOMO II [41]. The second variable is the percentage of change that is required for a responsibility in a task. For example, a 50% change in the “handle user interaction” responsibility might be required in a certain maintenance task. Finally, the third variable is the productivity of a maintenance task. The productivity for different maintenance task various according to the task and organization. But in general the productivity will be shown as x sloc per hour.

4.3.3 Anaytic Constraints

The problem with change impact analysis is that the values calculated for the responsibilities are prone to miscalculation errors. For examples, the size of the responsibilities have to be based on historial data and the experience of the architect. As such, calculating a range of values instead of using an average value can be more useful.

4.3.4 Model Representation

The model representation for the maintainability reasoning framework is the dependency graph. Dependencies between two responsibilities indicate a structural relationship between responsibilities. It can represent dependencies such as data flow and functional dependencies.

4.3.5 Interpretation

A maintainability scenario is mapped to a set of responsibilities in the architecture. This is done by tracing the scenario through the architecture and determining which responsibilities are part of the sceanrio. A responsibility is seldom isolated from other responsibiliites in a scenario as there is a certain amount of coupling from one responsibility to another. Identifying all the responsibilities involved in a scenario will enable the reasoning framework to calculate the effort required to perform a specific maintenance task.

4.3.6 Evaluation Procedure

In order to calculate the effort required on a maintenance task, the multiplication of the size of the responsiblity (sloc) and the percentage of change in a responsibility is taken and then divided by the productivity of the specific maintenance task.

$$\text{effort in man-hours} = \frac{\text{size}\times\text{percentage of change}}{\text{productivity}}$$

This is done for each responsibility of the maintenance scenario and then the values are added.

4.3.7 Example Application of Technique

In CTAS, the following maintenance scenarios exists as an example. Various maintenance scenarios have different productivity rates and the architect is responsible for assigning the productivity rate in order to calculate the man-hours required to complete a maintenance task.

- Perfective Maintenance: Change CTAS so that the user can switch service providers.
- Adaptive Maintenance: Some information regarding the user may be changed. For example, we might want to add additional information about the user such as the user's gps location.
- Corrective Maintenance: Interaction with outside services are malfunctioning because of the changes made by the service providers. These interactions with the external service providers can change frequently and corrections have to be made accordingly.
- Preventive Maintenance: Data added to CTAS has to be sorted/truncated/ compressed/optimized to ensure optimal operation of CTAS.

In the scenario, "Change CTAS so that the user can switch service providers", the scenario is mapped to the responsibilities as showned in table 4.5. The following equation calculated the total man-hour required for the maintenance scenario:

$$\frac{3000\times 0.3}{0.5} + \frac{4000\times 0.5}{0.5} + \frac{3500\times 0.5}{0.5} + \frac{2000\times 0.5}{0.5} = 11300 \text{ hours}$$

The calculation shows that it will take 11300 hours to complete the maintenance sceanrio.

Responsibility	**Size**	**Change Percentage**	**Produtivity**
Query for specific data	3000 SLOC	30%	0.5 sloc/hour
Register to provide information	4000 SLOC	50%	0.5 sloc/hour
Unregister	3500 SLOC	50%	0.5 sloc/hour
Signal changed Information	2000 SLOC	50%	0.5 sloc/hour

Table 4.5: Information of a Maintenance Scenario

4.3.8 Discussion of Technique

The maintainability reasoning framework is a quantitative reasoning framework where specific numeric values are calculated. Modifiability is also analyzed using Change Impact Theory [11] and the theory is used to implement a reasoning framework for modifiability [20]. One limitation of the modifiability reasoning framework is that there is no accounting for the difference in the productivity of perfective and corrective maintenance scenarios. In the maintainability reasoning framework, we have taken in account the differences by subdividing the maintenance scenarios into specific categories and allowing the architect to decide the productivity for the task.

4.3.9 Related Work

No explicit and numbered specifications can be provided for quality attributes like maintainability. In [8] the authors provide a method for predicting maintainability of software during architecture design. The method is formulated based on extensive experience in software architecture design and detailed design and exemplified using the design of software architecture for a haemo dialysis machine. The authors present a scenario profile for performing impact analysis and calculate maintainability based on it. Although there are many advantages and disadvantages of using this method, it is very practical and suitable for design processes that iterate frequently with evaluation in every iteration.

In order to establish whether the software architecture design process is complete, several techniques are required to analyze and measure the quality attributes of the system. In [13] the authors present a technique to assess maintainability of software architecture. Similar to [8] maintainability is calculated based on scenario profile and impact analysis and is optimized based on the maintenance effort of the profile. The method evaluates the required maintenance effort assuming all the changed requirements can be implemented by adding new components to the system. It is well exemplified by an industrial case from telecom field. The optimal average effort per scenario was determined which was comparable to the average effort of the architecture in the example.

Chapter 5

Qualitative Reasoning Frameworks

It is the hypothesis of this research that qualitative reasoning framework can be developed for quality attributes of dependability such as safety, confidentiality and integrity. For a complete reasoning framework for dependability, a qualitative reasoning framework is required for safety, confidentiality and integrity.

In this chapter, the reasoning frameworks for confidentiality, integrity and safety will be presented. The example software architecture used to derive the quality-attribute scenarios will still be CTAS (Clemson Transit Assistant System).

5.1 Confidentiality & Integrity

Confidentiality is defined as the absence of unauthorized disclosure of information. It is an operational quality attribute very much like reliability and safety [12]. In the confidentiality requirements research community, there has been work in which the proper elicitation and specification of confidentiality requirements have been explored [35], [17], [76]. There has also been work on assessing the confidentiality of software components [58], [14] in the context of security which is a quality attribute that encompasses confidentiality. Current methods to analyze confidentiality rely on existing threats.

At the architectural level, there is work on analyzing the confidentiality risk that is emergent from the software architecture [88], [40] (confidentiality is analyzed as security, which is a broader concept, in this work). This work takes a "blocks and arrows" view of the architecture and analyzes the risk on the system based on known attacks, attack patterns and vulnerabilities. The risks posed by the architecture are mitigated by changing the architecture to counter the threat. At least 50% of confidentiality defects are architectural in nature [88] and therefore considering the software architecture to improve confidentiality is essential.

The concept of risk is introduced to analyze confidentiality. Risk is determined by a simple equation that involves the attack potency, target exposure and the potential impact of the attack [39]. The attack potency describes the potential of an attack to create damage. High potency attacks cause noticeable damage to the system and low potency attacks cause damage that is not very noticeable. Applied to confidentiality, high potency attacks would result in a total (or near total) disclosure of data and a low potency attack would correspond to a small (or insubstantial) disclosure of data.

Exposure measures the ease of carrying out an attack. For example, low exposure can be said to occur when there is an authentication method that prevents dictionary attacks but still leaves more complicated types of attacks open (such as distributed denial of service attacks). High exposure can occur when an attack can be carried out with something as simple as a malformed URL. Attack potency and target exposure can be combined and referred to as the damage potential [88].

Another variable is introduced to calculate risk and it is called impact. Impact measures the consequence of the damage done by the attack. When an empty database is attacked, the attack has no impact because nothing of value was damaged. But when a database of personal information (that includes sensitive information such as a social security number) is attacked, the attack potentially has high impact. These three variables are combined to calculate risk as shown in equation 5.1.

$$\text{Attack Potency} \times \text{Target Exposure} \times \text{Impact} = \text{Actual Risk} \tag{5.1}$$

But at the architectural level, it is difficult to determine the attack potency or the impact of an attack. The attack potency can vary according to factors such as implementation and it can vary according to what we would define as a high or low attack potency. The impact is also a variable that is difficult to define. The impact can vary according to what is considered to be important data. Since these two variables cannot be readily determined at the architectural level, it will not be considered in the reasoning framework. But the exposure can be expressed in terms of the authorization level granted to the user depending on the sensitivity of the data under consideration. Data sensitivity is a classification of data using a model such as the Bell-LaPadula model. The Bell-LaPadula model classifies data into levels of decreasing sensitivity: confidential, top secret, secret and unclassified [10]. The sensitivity of data indicates how important a piece of data is to an organization.

In this section, an approach to analyze the confidentiality of the product derived from the architecture is presented in the standard form for describing a reasoning framework. The confidentiality reasoning framework is a qualitative reasoning framework based on analyzing variables such as the sensitivity of the data handled by the responsibilities and the authorization level of the user.

5.1.1 Problem Description

The main difference between confidentiality and integrity is that confidentiality requires data to be read only by the authorized users and integrity requires data to be modified only by the authorized users [53]. Confidentiality is a quality attribute that affects the operation of the system. Thus, confidentiality scenarios will be operational scenarios where the stimulus is from a user of the system trying to access (read) data that is stored in the system. The user could also store the data and then access it. Such scenarios models the pattern that a user accesses data and based on the pattern the reasoning framework can determine if the architecture supports the scenario based on the authorization level of the user.

Instead of dividing the user of the system as an agent or an unauthorized agent [17], the user will be given an authorization level based on the types of data the user should be allowed to read. For example, the access levels can be divided into an ordinal scale such as lowest, low, medium, high, highest authorization levels. These confidentiality levels are very much like the classical discretionary policies, such as confidential, top secret, secret and unclassified [81] but a nominal scale can be used instead of an ordinal one if necessary.

An example of a confidentiality scenario is the following: Given that the authorization level of a user is x, can the user complete a task y? By giving an authorization level to every type of user, agents and unauthorized agents can be modeled. Every confidentiality scenario is related to accessing some kind of data.

The elements of a general confidentiality scenario are the following:

- Stimulus: an agent/unauthorized agent tries to access data from the system with a certain authorization level.
- Source of the stimulus: an agent/unauthorized agent.
- Environment: at runtime.
- Artifact: the information system.
- Response: indicate which responsibility prevents the scenario from occurring.
- Response measure: a boolean true or false. True indicates that the user can access the data. False indicates that the confidentiality scenario cannot be satisfied.

In summary, the confidentiality reasoning framework takes a confidentiality scenario and analyzes it based on authorization level granted to the user and the sensitivity of the data that the user is authorized to access.

5.1.2 Analytic Theory

The analytic theory for confidentiality is a qualitative method of examining the sensitivity of the data handled by the responsibilities and the authorization level granted to the user as long as it is along the path indicated in the scenario. The sensitivity of the data indicates the level of authorization that is required to be able to read the data. For example, social security numbers can be considered as highly sensitive data. Therefore, to satisfice any scenario that handles social security numbers, the user who does not have the appropriate level of authorization should not have access to it.

Showing that a scenario where a user with only a certain level of authorization has access to the social security numbers can be an acceptable way of showing that the social security numbers are sufficiently protected in the scenario. But depending on the organization and the stakeholders, the required authorization may be different. The appropriate level of authorization required for data with different levels of sensitivity will be referred to as the satisficing authorization level matrix for the organization or stakeholder. Table 5.1 presents the satisficing authorization level matrix for the CTAS stakeholders.

Using this matrix, architects can assign a maximum data sensitivity level of the data handled in each responsibility and derive the appropriate level of authorization level that is required to access

Data Sensitivity	**Satisficing Authorization Level**
Very Highly Sensitive	Very High
Highly Sensitive	High
Medium Sensitive	Medium
Low Sensitive	Low
Very Low Sensitive	Very Low
Unclassified	None

Table 5.1: Satisficing Authorization Level for CTAS

the data in the responsibility. (The maximum data sensitivity level is considered because a user has access to all data that requires less authorization than the user's current authorization level.) Once the maximum data sensitivity levels for each responsibility is assigned, the reasoning framework can determine if the user can access the data in certain responsibilities based on the authorization level. A detailed explanation of the reasoning process is given in the evaluation section.

5.1.3 Analytic Constraints

The analytic constraint of the confidentiality reasoning framework is the lack of consideration for unknown risks. For each scenario, the reasoning framework returns a result that shows whether the architecture is satisficing the scenario or not based on the user's authorization level and the sensitivity of the data handled in the responsibilities. But confidentiality can be influenced by other system-specific variables, such as the operating environment, that can be exploited to gain confidential data. It is difficult to consider those variables at the architectural level and we assume those factors to be considered later during the development of the product.

5.1.4 Model Representation

Every confidentiality scenario has informational flow and therefore an architectural view that shows the informational flow is essential to its analysis. The architectural view associated with the flow of information is called the informational viewpoint [79]. The informational viewpoint can be used to answer questions about how the system will store, manipulate, manage, and distribute information [79]. The informational viewpoint can be modeled with the standard responsibilities and its connections. But the responsibilities and connections associated with the informational view point are not the only elements that should be considered. The informational viewpoint should be the source for deriving the responsibilities but all the responsibilities associated with a confidentiality scenario should be included in the model for completeness.

The responsibilities derived from the informational viewpoint are responsibilities that are directly involved in the flow of data in a scenario. But the model for a confidentiality scenario must also include responsibilities that are indirectly involved in the scenario. For example, a responsibility to authenticate a user is not directly involved in the flow of information but it is still part of the operational scenario that stores the user name and password. In other words, there are responsibilities that are the essential part of the confidentiality scenario and there are responsibilities that support the essential ones.

For each responsibility identified for a confidentiality scenario, there is a maximum sensitivity level of the data that is handled by the responsibility. For example, a responsibility that handles credit-card numbers has a maximum data sensitivity level of "very high" and this indicates that this responsibility has enough controls to handle data such as phone numbers which has a lower data sensitivity level of "medium." The maximum data sensitivity level of each responsibility indicates the authorization level that is required to operate that responsibility. For example, a responsibility that has a maximum data sensitivity level of "medium" requires the user to have at least a medium authorization level to operate it.

The connections between the responsibilities in a confidentiality scenario indicates either a control flow or an informational flow. A control flow from one responsibility to another is a unidirectional connection where a parent responsibility indicates a responsibility that needs to be operated before the child responsibility. An informational flow is a connection that indicates data transferring from the parent responsibility to the child responsibility. These connections form a graph of responsibilities that represent the run-time informational/control flow of a confidentiality scenario where a user is trying to access some kind of data given an authorization level. Control flow captures all the participating responsibilities of a scenario and the informational flow shows the responsibilities that either process or store data.

5.1.5 Interpretation

The confidentiality scenario does not indicate that a user is an agent or an unauthorized agent based on the authorization level. An unauthorized agent can be modeled as a user who has a low authorization level (or no authorization level) trying to access a part of the architecture that requires high authorization. The reasoning framework returns a true or false based only on the authorization level of the user. A true indicates that the confidentiality scenario is satisficed and a false indicates that there is a potential for a breach in confidentiality.

The responsibilities of the confidentiality reasoning framework model any elements that read, write or modify data as part of satisficing the scenario. The connections among the responsibilities represents control flow from one responsibility to another. From the confidentiality scenario the responsibilities that are part of it must be derived, their maximum data sensitivity levels must be set and the connections among the responsibilities must be identified.

5.1.6 Evaluation Procedure

There are two types of confidentiality problems that can occur. In the first case, a user can have unauthorized access to responsibilities that require a higher authorization level than the current authorization level the user is assigned. In the second case, a user with an appropriate level of authorization may not be able to access responsibilities that are part of the scenario. These two cases must be avoided if the scenario is to be satisficed by the reasoning framework.

For the first case, to check that there are no unauthorized accesses, all authorization levels lower than the current level assigned to the user will be run through the scenario. This will make sure that users with an authorization lower than the current level will be not be able to access the scenario under consideration. For the second case, to check that the authorized access is possible, the current authorization level will be used to go through the responsibilities to determine that the user has access to all the responsibilities.

The main steps for analyzing the scenario is as follows:

1. Using the Satisficing Authorization Level Matrix for the organization, determine the specific authorization level require for a specific piece of data that is the target of the scenario.

2. Check if unauthorized access is possible by taking the authorization levels lower than that of the user and comparing it with the required authorization level for each responsibility mapped to the scenario. The scenario should fail at some point since some responsibilities cannot be accessed with a lower authorization level.

3. Check if authorized access is possible by taking the authorization level of the user and comparing it with the required authorization level for each responsibility mapped to the scenario. The

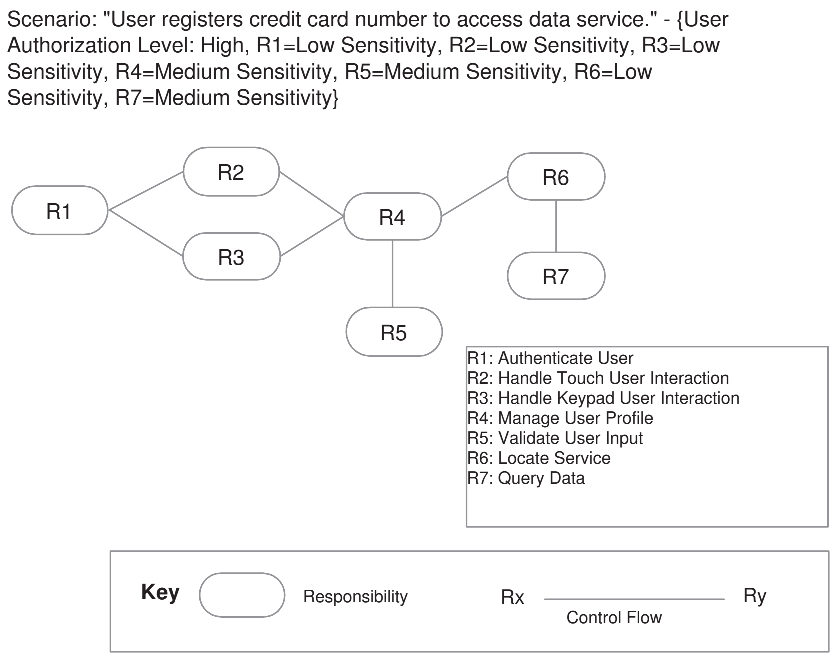

Scenario: "User registers credit card number to access data service." - {User Authorization Level: High, R1=Low Sensitivity, R2=Low Sensitivity, R3=Low Sensitivity, R4=Medium Sensitivity, R5=Medium Sensitivity, R6=Low Sensitivity, R7=Medium Sensitivity}
R1
R2
R3
R4
R5
R6
R7
R1: Authenticate User
R2: Handle Touch User Interaction
R3: Handle Keypad User Interaction
R4: Manage User Profile
R5: Validate User Input
R6: Locate Service
R7: Query Data
Key
Responsibility
Rx
Ry
Control Flow

Figure 5.1: Confidentiality Scenario

scenario should succeed since all responsibilities can be accessed with the user's authorization level.

The above steps can be used to a determine if confidentiality scenario can be satisficed for not.

Here is an example of analyzing a confidentiality scenario. A CTAS confidentiality scenario is the following: User registers credit card number to access a data service provided by CTAS. The responsibility graph of the scenario is shown in Figure 5.5. In this scenario, the user tries to access a data service provided by CTAS by registering a credit card number. The authorization level granted to the user is “High” which indicates that the user has authorization to operate any responsibility which has a maximum sensitivity level of “High.”

The first step of the analysis is to determine if the scenario is actually possible given the authorization level of the “High.” When all the sensitivity levels of the responsibilities in the scenario is considered, we can see that every responsibility has a maximum sensitivity level of “Medium” or lower. This indicates that the user will be able to register a credit card number and have access to a data service.

The second step of the analysis is to determine if users with lower authorization levels have unauthorized access to the scenario. When a user with a “Medium” authorization level operates the scenario, we can see that no responsibility in the scenario prevents the user from accessing it since the maximum sensitivity level of every responsibility is “Medium” or lower. From this we can see that the scenario is operable even when a “Medium” authorization level is granted to the user. This means that a user with a “Medium” authorization level is able to register a credit card number and access a data service. This violates the “High” authorization level required for the user by the scenario. The result of running this scenario in the confidentiality reasoning framework would indicate that the scenario cannot be satisficed by the current architecture.

5.2 Safety

Safety is an essential attribute of many dependable systems. Safety is defined as the freedom from unacceptable risk of an event that results in loss of life, injury, or property damage [63]. What is unacceptable varies from one domain to another. Software cannot cause direct harm but software embedded in automobiles, aircraft, robots, and even elevators has the potential for causing accidents, which result in injury or death. These accidents are the result of hazards, events that can potentially cause death or injury [63].

Many safety issues arise from the context in which the software is used but some issues arise due to a failure to achieve sufficient levels of certain quality attributes. For example, a failure to meet a performance goal may cause a synchronization problem, which results in loss of control at a critical moment. Even qualities concerning the static structure of the architecture, such as maintainability, can affect the safety of the system by reducing the rate at which defects are removed or increasing the rate at which defects are injected during maintenance.

Architects design an architecture to meet safety requirements by introducing certain structural elements and relationships. Certain architectural designs reduce the likelihood of a hazardous event from occurring and mitigate the impact of those accidents if they do occur [74]. The functionality and the external factors of the software are known to cause most safety problems. In order to address those problems more effectively, safety has to be considered starting with the architecture level.

At the earliest stages of safety analysis, system requirements, and the context in which the system will be deployed, are analyzed to identify hazards [64]. Certainly some accidents result from hardware failures over which the software has no control, but many accidents occur because of software failures. A safety analysis of the software examines the functional and non-functional software requirements. These requirements are analyzed to associate individual requirements with specific hazards. Our work on safety design is based on the idea that some safety requirements are closely related to other quality requirements.

The architect needs a method by which a proposed modification of the architecture can be compared to the original architecture to determine which will result in safer systems. This can be done by examining the effect of the modification on the safety hazards. To replace the older architecture, the modified architecture should either reduce the risk of certain events from occuring or it should result in those accidents that do occur having less impact.

Accidents are caused by failures (hazards that have been realized) [63] and there is an inherent nature to each failure [93], which in some cases can be related to qualities of the software involved in the failure. A failure of omission can indicate an availability problem, a failure to produce a correct value can indicate a reliability problem, and timing issues are indicative of performance problems [93]. Even failures that result from usability problems can lead to safety problems [60].

The "safety" quality will be defined in terms of the qualities identified as potential safety hazards during the analyses. We hypothesize that safety can be analyzed using a set of reasoning frameworks depending on the nature of the hazards and then the aggregate result from the reasoning frameworks can be used to characterize the level of safety in the architecture. This is because the failures that cause many safety problems can each be classified as a failure to satisfy a specific quality requirement. Any quality attribute that can increase the probability of a hazard resulting in an accident should be part of the safety analysis. For example, the safety of a software-based fire alarm system is largely dependent on how reliably the fire is detected and processed by the system. This is an example where a failure to satisfy a reliability requirement becomes a safety problem.

In this section, a safety reasoning framework which leverages this observation while codifying some of the existing techniques and approaches is presented. These are first outlined as actions in a traditional safety analysis and then the canonical outline for describing reasoning frameworks to describe our framework. These work together to provide the safety analysis process shown in Figure 5.2.

5.2.1 Initial Safety Analyses

The safety reasoning framework builds on the information collected in a traditional system safety analysis which includes Functional Hazard Analysis and Fault Tree Analysis. These analyses are subjective and directly affected by the expertise of the personnel conducting the analyses.

Functional Hazard Analysis (FHA) identifies hazards by examining the potential effects of functional failures [91]. Those that can result in injury or death are safety hazards and are the input to the Fault Tree Analysis. The potential effect of a functional failure will also help determine the root cause of the failure. The information from the FHA is captured in the form shown in Table 5.2.9 which shows the functional hazard analysis for our example system.

Fault Tree Analysis (FTA) identifies the root causes of undesired events including those

Quality Attribute	Hazard-affecting events
Reliability	Incorrect output is generated
Availability	A system element that was supposed to be in service is not ready for use when needed
Confidentiality	Information of highly sensitive nature is visible to unauthorized persons
Integrity	Highly sensitive data is modified by an unauthorized person
Maintainability	A critical fault is repaired incorrectly

Table 5.2: Some Quality Attributes Used to Analyze Safety

events that result in injury or death. For our purposes only those hazards that are identified as safety critical during the Functional Hazard Analysis are subjected to FTA. FTA uses the graphical symbology shown in Figure 5.4 that identifies different types of events and linkages.

The Fault Trees for the safety critical hazards related to a specific product are an input into our reasoning framework. In our reasoning framework we will be looking for basic events, those circles at the bottom of the diagram in Figure 5.4.

5.2.2 Problem Description

Accidents occur as a result of failures during system execution, but may be caused by either dynamic or static faults in the architecture. Table 5.2 illustrates some of the failures that can affect the probability of a hazard resulting in an accident. This is not a comprehensive list. Almost any quality attribute can, in the right circumstances, affect safety. The architect is responsible for judging which quality attributes are a safety concern for the system that is under consideration. Although this may seem subjective, there are examples, such as, the Architecture Trade-off Analysis Method (ATAM), which show a similiar approach to identifying qualities.

The input to the safety reasoning framework is the set of safety hazards output from the initial safety analysis described in section 5.2.1. A safety scenario is constructed for each of the identified safety hazards. A system safety requirement sets an expected tolerance for each hazard and the requirement is captured in a safety scenario.

The general scenario for the safety reasoning framework, which defines the form of a safety scenario, contains the information described in Table 5.3.

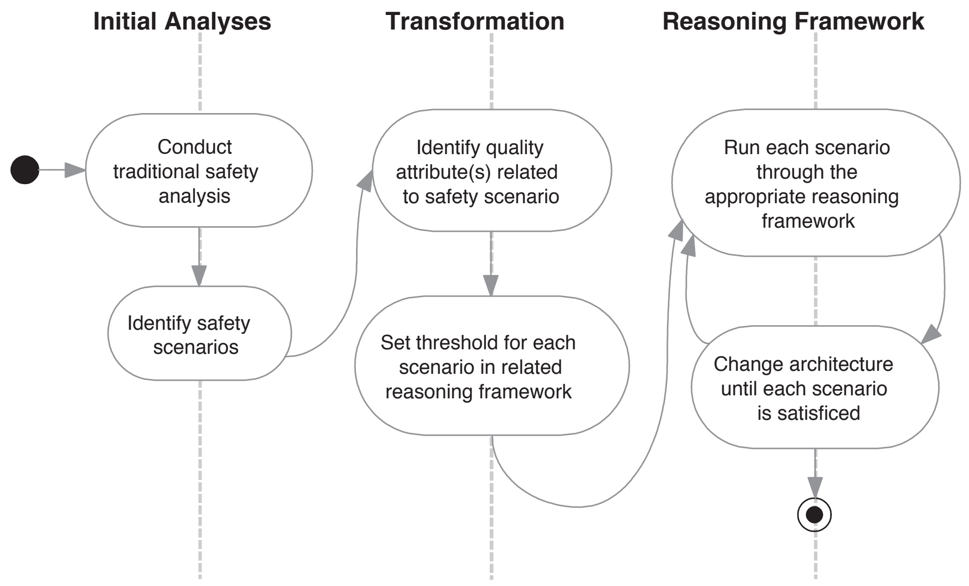

Initial Analyses
Transformation
Reasoning Framework
Conduct traditional safety analysis
Identify safety scenarios
Identify quality attribute(s) related to safety scenario
Set threshold for each scenario in related reasoning framework
Run each scenario through the appropriate reasoning framework
Change architecture until each scenario is satisficed

Figure 5.2: Safety Analysis

5.2.3 Analytic Theory

The analytic theory for the safety reasoning framework involves a set of transformations among the analytic theories related to the identifed qualities. As stated previously, words in the description of a safety scenario, such as fault, missed deadline, etc, are used by the architect to map into quality attributes such as availability, reliability, or some other quality. The safety scenario is transformed into scenarios in the appropriate reasoning framework(s), as shown in Figure 5.3. This transformation from a safety scenario to a scenario for another quality attribute is done by semantically matching keywords in the scenario. For example, if a word related to scheduling is found in the safety scenario, then it is matched to a performance scenario. This transformation allows the safety reasoning framework to utilize the results of the target reasoning frameworks.

The transformation is further defined by identifying the information required for the general scenario of each target reasoning framework. The architect acquires any information needed for the quality scenario that is not already available in the safety framework. For example, the performance reasoning framework needs a maximum latency value. If performance is sufficiently important that it can threaten safety, it is reasonable to assume the architect will be able to estimate the required latency.

The target reasoning framework is applied and the scenario is evaluated. As is usually the

Stimulus:	A stimulus that addresses a quality requirement that can increase the likelihood of an accident from a hazard.
Source of the stimulus:	Any actor of the system
Environment:	The stimulus is introduced during the execution of the system.
Artifact:	System and potentially some real-world accident that results in injury or death
Response:	The system fails and the user is harmed in some way.
Response measure:	What is the impact of the failure?

Table 5.3: General Scenario for Safety

case, the reasoning framework computes an estimate for the quality using the current architecture and determines whether that value satisfices (We use the term "satisfice" since the outcome of the analysis does not tell us the optimal solution to satisfy the safety requirements but instead it tells us that the scenarios have reached their minimum allowed thresholds.) the derived scenario by estimating whether the quality threshold would be reached. That judgment is returned to the safety reasoning framework as input to the determination of whether the safety scenario is satisficed.

When multiple quality attributes are identified for a safety scenario then the results from all of these attributes are combined to determine whether the safety scenario has been satisficed. For example, in Figure 5.3, a safety scenario is composed of two other scenarios which are related to usability and confidentiality. Those two other scenarios would both have to be satisficed in order to satisfice the corresponding safety scenario.

Once the individual safety scenarios have been evaluated, the architect checks the set of safety scenarios to see which have been satisficed. The priority is based on the cost of an accident and the probability of the hazard causing an accident. The architect changes the architecture in response to the values returned by the reasoning framework. The evaluation process is repeated until, in the expert judgement of the architect, a sufficient number of the scenarios are satisficed.

5.2.4 Analytic Constraints

The main limitations on the safety reasoning framework are the focus on qualities and the dependence on the architecture representation. There are failures that are due to functional problems that are not the result of a failure to satisfy a quality requirement [63]. The reasoning framework does not address these problems since they are largely outside the software. For each safety scenario, the quality attribute estimate computed by the reasoning framework depends on the network of responsibilities that represent the system. The completeness, correctness, and consistency of this network determines the quality of the network.

5.2.5 Model Representation

The model representation for the safety reasoning framework is a network of reasoning frameworks. The mapping that transforms the individual safety scenarios into scenario(s) in the appropriate reasoning framework(s) constructs the model. It is the set of arrows shown in Figure 5.3 that originate from the safety scenario. Constructing the mapping requires the architect to identify the qualities represented in the safety scenario and then to provide the additional information that is needed for the target reasoning framework that is not used by the safety reasoning framework.

The resulting model is a network of reasoning frameworks that are joined together by transformations and returned results. This is made possible by considering each reasoning framework to be a view on a single architecture. In tools such as ArchE the architecture is a network of responsibilities. Different reasoning frameworks add their own linkages between responsibilities, but the set of responsibilities remains the same.

5.2.6 Interpretation

The hazards obtained by FHA are subjected to FTA to identify their respective causes and the quality attributes behind them. The hazards are prioritized based on criticality as described in section 5.2.1. These hazards are associated with their respective quality attributes (causes). The safety scenarios associated with each hazard are mapped to the scenarios of the quality attributes related to the corresponding hazards. Figure 5.3 gives a conceptual view of the mapping. In this example, the safety scenario is mapped to usability and confidentiality scenarios.

5.2.7 Evaluation Procedure

The quality attribute scenarios that are derived from the safety scenarios are each evaluated in the appropriate reasoning framework. The result of that evaluation is locally judged to have satisficed its requirement. The data available to the safety reasoning framework are boolean,

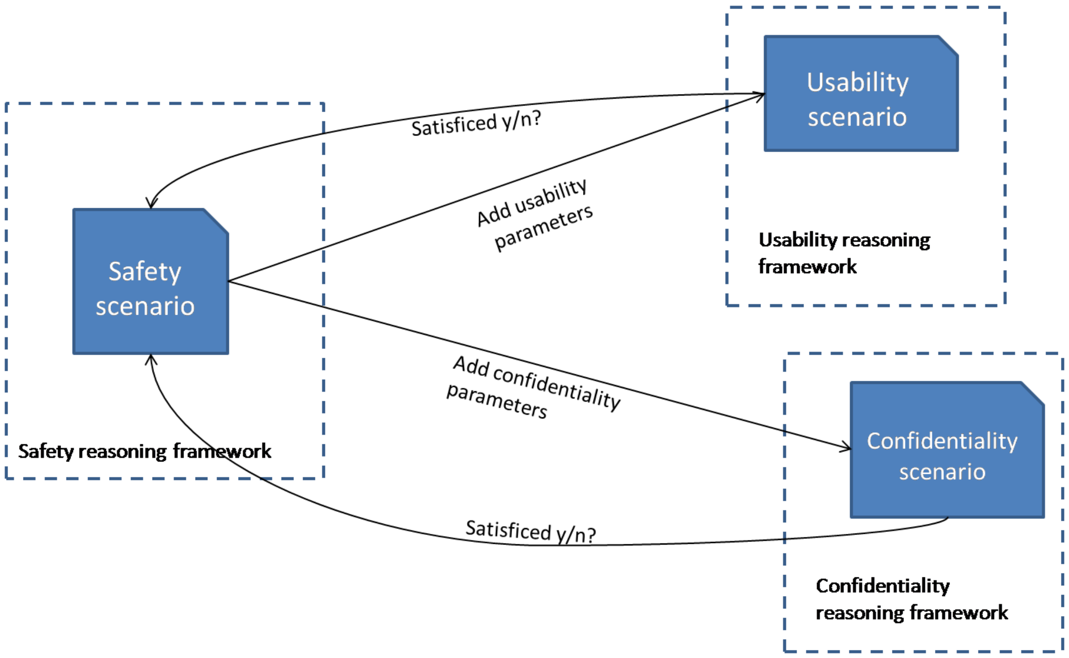

Usability scenario
Satisficed y/n?
Add usability parameters
Safety scenario
Usability reasoning framework
Add confidentiality parameters
Confidentiality scenario
Safety reasoning framework
Satisficed y/n?
Confidentiality reasoning framework

Figure 5.3: Transformations between theories

satisficed/not satisficed, judgements for each scenario.

We exploit the risk-based nature of hazards to rank the hazards identified in the initial safety analysis. This approach uses the probability that the hazard will occur, the probability that the hazard will lead to an accident if it does occur, and the cost if there is an accident to determine the ranking. The ranking is an ordered list of satisficed and not satisficed scenarios. There is no intention to derive a single score from the set of safety scenarios.

The ranking of the scenarios gives the architect an order in which to attack the scenarios that have not satisficed their requirements. Suggestions regarding how to improve the safety aspect of the architecture can be made to the architect using tools that draw on the reasoning frameworks for the derived scenarios.

One aspect of the qualitative nature of the safety reasoning framework shows up when we think of which transformations to apply. Certain tactics will improve some scenarios and diminish others (based on general trade-offs) [46] and by applying these tactics based on that general observation (a qualitative method) we may be able to satisfice all the safety scenarios (or not).

Function:	User Data Management
Failure Condition:	Disclosure of private data to unauthorized user
Phase:	run-time
Effect:	Disclosure of data is undetectable to the system
Class:	Medium Criticality
Verification:	User is harmed by the abuse of the disclosed private data

Table 5.4: Results of a Function Hazard Analysis on CTAS

5.2.8 Summary

The presented technique fits well within the typical iterative software architecture design process. The architect makes changes to the network as a result of the feedback from all of the reasoning frameworks. The individual reasoning frameworks re-evaluate the status of each scenario and report back an update on whether each scenario is satisficed by the new architecture.

5.2.9 Example Application of Technique

We will illustrate our technique using the Clemson Travel Assistance System (CTAS) [70].

As previously described in section 4, the quality attribute requirements of CTAS include performance, security, functionality, reliability, usability, safety, efficiency, maintainability and portability [70]. With these drivers in mind, the architecture representation of CTAS, shown in Figure 5.5 will be used to apply our technique of safety analysis and transformation. Due to space limitations we show only the most pertinent results.

5.2.9.1 Perform Initial Safety Analyses

The architect was assisted by domain experts to define the context in which CTAS operates and to identify potential failures. Table 5.2.9 shows a portion of the results of the FHA on CTAS.

The FHA revealed hazards associated with the itinerary management component, the user data management component, and the user interface component that could lead to safety problems. The next step in our analysis was to use each hazard as the starting node of a fault tree and analyze the root cause of the hazard. Figure 5.4 illustrates the fault tree for the hazards associated with the user data management component.

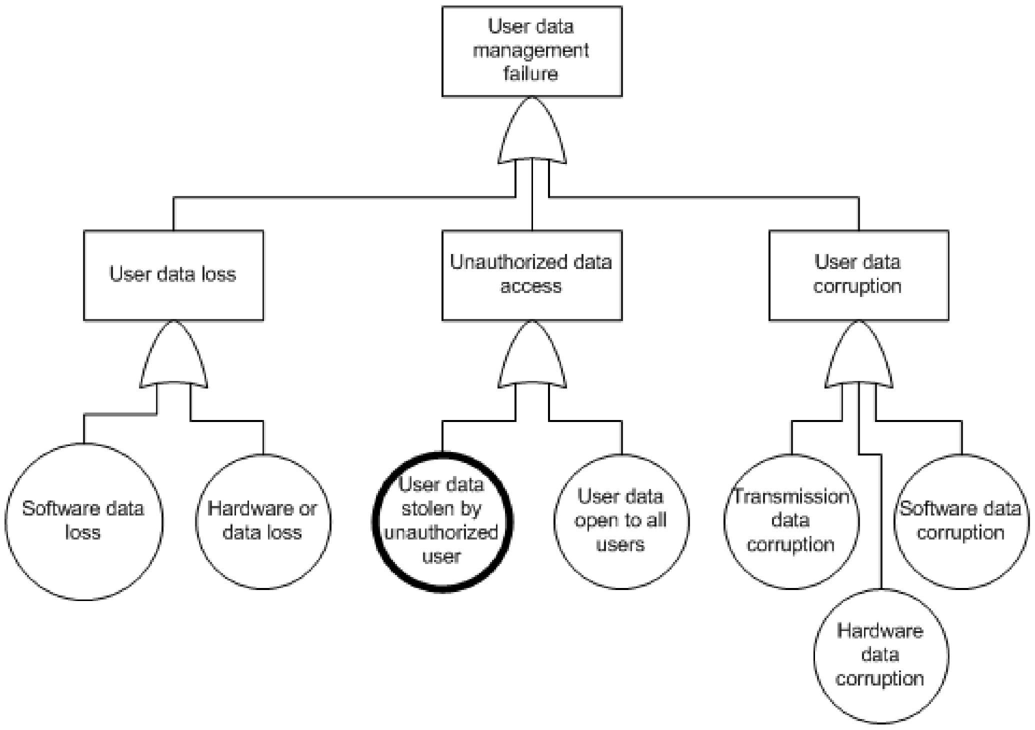


Figure 5.4: Fault Tree for Failure in User Data Management

In the FTA diagram, the bold circle indicates a safety fault that is related to a quality attribute of CTAS. The other circles indicate safety faults that are related to the functionality of CTAS.

5.2.9.2 Translate into Safety Scenarios

The faults from the FTA that affect certain quality attributes were turned into safety scenarios. Using the general scenario for safety we constructed the concrete scenarios. We will only show the safety scenario mapped to the confidentiality scenario [44]. In this scenario, CTAS stores sensitive information such as address, telephone, social security number of the user, which could be used to harm the user. An unauthorized user accesses the user's personal data. The data is used to locate the user's home and the user is attacked at their home. The concrete scenario, in the standard form of the general scenario for the safety reasoning framework, is illustrated in Table 5.5.

Stimulus:	Access of confidential data
Source of the Stimulus:	Unauthorized user
Environment:	Normal mode
Artifact:	Personal Data of user
Response:	End user's personal data is accessed
Response Measure:	The loss of privacy the user experiences due to the unauthorized access

Table 5.5: Safety Scenario related to potential confidentiality failure

Stimulus:	Attempt to read CTAS user's social security number
Source of the Stimulus:	Unauthorized person
Environment:	End user's hand-held CTAS device
Artifact:	CTAS database
Response:	The social security number is read
Response Measure:	The amount of physical harm that comes to the user whose SS number was read

Table 5.6: Confidentiality scenario after mapping from the safety scenario

5.2.9.3 Apply the Constituent Reasoning Frameworks

Each of the safety scenarios was mapped to the specific reasoning frameworks corresponding to the derived qualities, as illustrated in Figure 5.3. In other words, each scenario was transformed into the form prescribed by the general scenario for that framework. We show the process only for the confidentiality scenario. The general safety scenario was mapped to the general confidentiality scenario. The confidentiality scenario needed the architect to identify the data that was to be safeguarded, which actors in the use case model were the unauthorized users, and the amount of exposure that was permissable. The safety scenario in Table 5.5 was mapped to the confidentiality scenario in Table 5.6. The mapping was done to a confidentiality scenario because of the keyword "unauthorized" found in the safety scenario. The architect provided the *Stimulus*, *Response* and *Response Measure* goal for the new scenario.

Table 5.6 shows the concrete confidentiality scenario that resulted from mapping the safety scenario to the confidentiality reasoning framework. The actual *Response Measure* when evaluated by the confidentiality reasoning framework is "failed" for the current CTAS architecture since some sensitive data was accessed by an unauthorized person. The result that the scenario is not satisficed is returned to the safety reasoning framework as the result of the confidentiality reasoning framework.

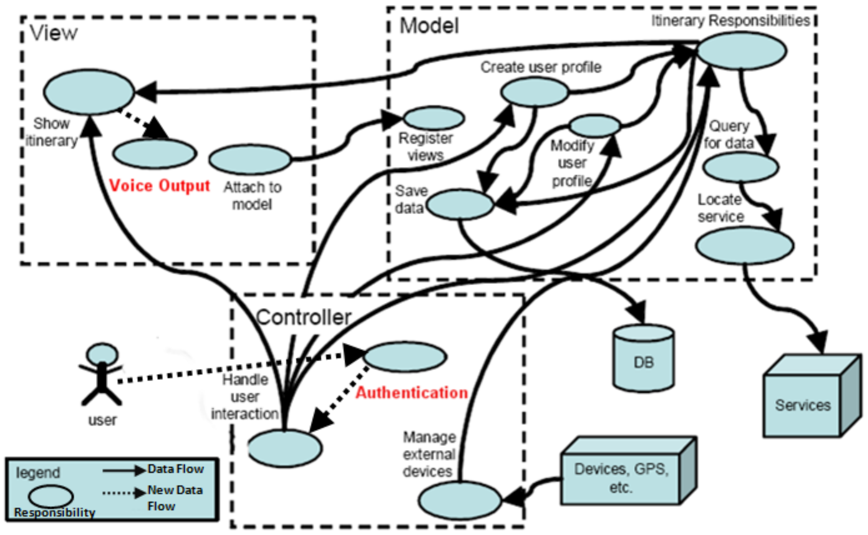


Figure 5.5: CTAS Software Architecture

5.2.9.4 Apply the Safety Reasoning Framework

The architect determined to what extent the safety scenarios have been satisficed by the current architecture. When all of the safety scenarios have been evaluated in the constituent reasoning frameworks the architect had a set of true/false results. In our analysis, the confidentiality and the usability scenarios were not satisficed by the current architecture so the corresponding safety scenarios were not satisficed. The architecture needed to be changed to satisfice the two scenarios under consideration.

The architect used the criticality among the scenarios, established early in the initial safety analyses, to determine the order in which transformations are going to be applied to the architecture. In our example, the architect has previously determined that the confidentiality hazard takes precedence over the usability hazard. The architect followed suggestions generated by the confidentiality and usability frameworks and made modifications to the architecture.

The orginal architecture, as a set of responsibilities in the form of an architectural cartoon, is shown in Figure 5.5. The dashed lines indicate responsibilities that have been added as transformations to the architecture.

In order to improve the confidentiality scenario, an *authentication* responsibility was added. These transformations in the architecture are not conflicting; however, the realization of the au-

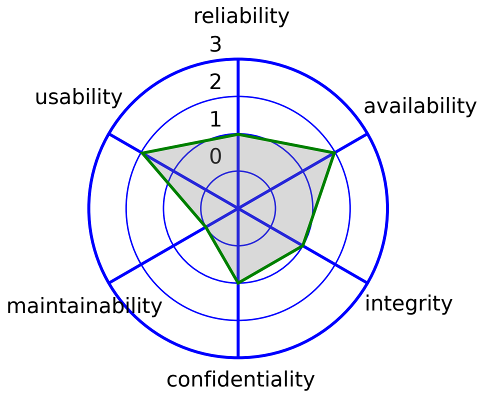


Figure 5.6: Star Plot of Safety Analysis

thentication responsibility did reduce the usability of the system. A *voice output* responsibility was added to improve the usability scenario. The revised architecture now satisfices the confidentiality and usability scenarios which implies that the two safety requirements we are investigating are both satisficed.

By satisficing the two safety scenarios, a determination based on the safety goals of CTAS can be made. And from the analysis we have determined that the safety scenarios related to usability and confidentiality have been satisficed.

The result of the analysis are expressed using a star plot was shown in figure 5.6. The various levels on the star plot indicate the degree to which a quality attribute is satisficed (0 - Unsatisficed, 1 - Minimum level satisficed, 2 - Good level satisficed, 3 - Max level satisficed). The value on each axis is a weighted average calculated from the the scenarios found in the analysis of that quality attribute. We omit the confidence intervals for each analysis because of space limitations.

The star plot is also used to determine to what extent an architecture has become *safer* after certain transformations have been applied. The typical approach is for the architect to use the star plot to guide modifications to the architecture.

5.2.10 Related Work

Much work has been done to develop methods for satisfying software safety requirements. At the architectural level, techniques such as Failure Propagation and Transformation Notation (FPTN) [23] and Component Fault Trees (CFTs) [54] are used to evaluate whether a software architecture can meet the safety requirements. These approaches are based on modular failure propagation models that model the system in terms of components that create, transform and consume failures [34]. These techniques allow the architect to determine the likelihood that a hazard would occur based on the software architecture. Our technique complements exsiting techniques by taking a broader perspective of safety that includes quality attributes.

5.3 Conclusion

We have demonstrated the viability of implementing a quality attribute reasoning framework by decomposing it into other reasoning frameworks. The results were published in [50], [46] and [71]. The mappings used in this work transform a safety scenario with a required level of safety into scenario(s) in the target framework(s) with corresponding levels of those attributes necessary to satisfy the original safety requirement. The results in the target reasoning frameworks provide the data for the safety reasoning framework to use to guide the architect toward beneficial modifications of the architecture.

Our work on reasoning frameworks has led us to identify relationships among frameworks. Initially we found that one way to address the varying definitions of an attribute is to view that attribute as a composite of other attributes. For example, an attribute such as dependability is really an aggregation of other attributes including reliability, availability, confidentiality, integrity, maintainability and safety. Now our work on safety has led to the realization that safety, as a quality, is a facade that has dependencies on other attributes for analytic theories that provide estimates of attribute values. Our future work will explore these relationships more deeply and possibly identify other relationships as well.

Chapter 6

Assembling the Reasoning Frameworks

Modern software architecture design techniques emphasize the quality attributes of the anticipated products. In order to satisfy the non-functional requirements that define acceptable levels for the quality attributes the architect applies specific tactics. Each tactic is intended to enhance a particular quality but also affects other qualities as well. Tools such as ArchE [20], intended to support the architect in this activity, focus on one quality at a time. The result is an architecture that optimizes one attribute but is seldom optimal for other qualities.

The goal of the architect is to design one software architecture that satisfies all of the quality attribute goals of the system. The reality is that the architect can balance a couple of attributes at the same time but not the number usually associated with an industrial-strength program. One apporach is to address one attribute at a time until a set of architectures are produced, each optimizing one of the qualities. The architect must then combine these multiple architectures. The contribution of this chapter is to describe an investigation into techniques for combining multiple attribute-specific architectures into a single product architecture.

In Section 6.1 we provide the background necessary to understand the work. In Section 6.2 we describe attribute-specific architectures. In Section 6.3.1 and Section 6.3.2 we give an algorithm for combining multiple attribute architectures. In Section 6.4 we apply the technique to an example. Finally we summarize the results so far and identify future work.

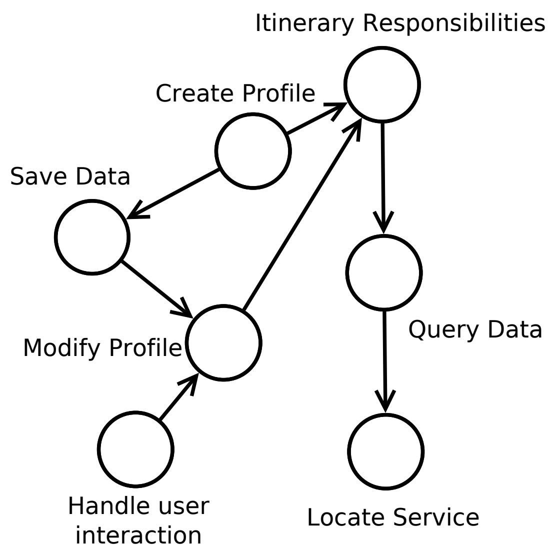


Figure 6.1: Dependency Graph of CTAS

6.1 Background

Connectors correspond to dependencies among code modules. A dependency may be a static relationship such as inheritance or a dynamic relationship such as control flow or data flow [66]. In software architectural languages such as AADL (Architecture Analysis and Design Language) [22], keywords such as data, event, event data, data access, bus access and port group are used to specify the different types of connectors found in the architecture [22]. A more detailed analysis of the various types of connectors is found in [72].

Connectors are also found in the architectures that are derived to analyze various quality attribute requirements of a system. Figure 6.1, shows a fragment of a dependency graph found in the Clemson Travel Assistance System (CTAS) architecture[69]. The circles in the figure represent responsibilities and the lines represent connectors. In this example, the connectors represent dependencies and the dependency connectors are used to analyze modifiability scenarios of the architecture. Depending on the quality attribute that is being analyzed, the connectors can represent control flow, data flow or a more general dependency.

Figure 6.2, represents five of the qualities that constitute dependability[1]. Each circle is a responsibility and the lines between circles are connectors. We will use these cartoons as a means

[1]We omit safety for simplicity because it is a combination of multiple attributes itself.

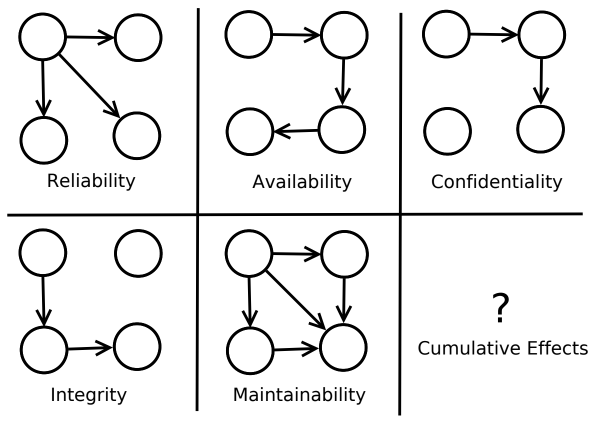


Figure 6.2: Attribute Architectures

of explaining our technique for combining architectures.

In the following sections we present the process of taking two or more attribute architectures and combining them into one product architecture.

6.2 Attribute Architectures

An attribute architecture consists of a finite set of responsibilities and its connections that determine the value of a particular quality attribute. The set of responsibilities of an attribute architecture is represented as:

$$R = \{R_1, R_2, \ldots, R_n\} \tag{6.1}$$

and the connections among them are represented as:

$$C = \{C_{12}, C_{21}, \ldots, C_{nm}\} \tag{6.2}$$

A connection C_{12} is a connection starting from responsibility R_1 and ending at R_2. Connections are created by an architect, or explicitly omitted, in an effort to optimize a specific quality attribute.

We use a common set of responsibilities across all of the attribute architectures but each attribute is characterized by distinct types and arrangements of connections to form a specific architecture. “Control flow” connections are used to form an attribute architecture that is being optimized for reliability because reliability is computed by tracing the program flow and combining the reliabilities of the various segments. The architecture can be expressed as (the r represents reliability and c represents a control flow):

$$C_r = \{C^c_{12}, C^c_{21}, \ldots, C^c_{nm}\} \tag{6.3}$$

An attribute architecture consists of two elements: a set of responsibilities (which is a subset of the entire responsibilities found in an architecture) and a set of quality attribute connections.

$$A_r = \{R_r, C_r\} \text{ where } R_r \subseteq R \text{ and } C_r \subseteq C \tag{6.4}$$

The aggregation of attribute architectures results in a product architecture that can be expressed with R and C as in the following equation:

$$A = \{R, C\} \tag{6.5}$$

But deriving A becomes a composition problem when more than one quality attribute architecture needs to be “combined” to form a product architecture.

In order to derive A, a composition of the attribute architectures must be computed. Suppose we are trying to combine a reliability attribute A_r and an availability attribute architecture A_a into A.

$$A = A_r \circ A_a = \{R_r \cup R_a, C_r \circ C_a\} \tag{6.6}$$

Deriving $R_r \cup R_a$ is a straight-forward union of responsibilities but deriving $C_r \circ C_a$ requires the composition of the connectors found in the attribute architectures. In the next section we explore the issue of combining the various connectors of different attribute architectures.

Quality Attribute	**Analytic Theory**	**Connector Type**
Reliability	Reliability rate based calculation [47]	Control flow (C^c)
Availability	Availability rate based calculation [47]	Control flow (C^c)
Confidentiality	Authorization level based calculation [44]	Data flow (C^d)
Integrity	Authorization level based calculation [44]	Data flow (C^d)
Maintainability	Change impact based calculation [20]	Dependency (C^l)

Table 6.1: Connector Types

6.3 Combining Attribute Architectures

In this section, we present the basic theory on combining the attribute architectures and then present a realization of the theory based on AADL syntax.

6.3.1 Connection Types and Compositions

In order to combine the six sub-attributes architectures of dependability into one representative architecture, the connectors of each attribute architectures and their characteristics must be analyzed. Table 6.1, shows the connector type for each attribute architecture and shows how it relates to the analytic theory of the respective quality attribute.

The analytic theories for reliability and availability involve calculations using the control flow C^c between responsibilities. A control flow expresses the flow of execution from one responsibility to another. As such, the connector is expressed as C^c and it is used in the reliability and availability attribute architectures.

The analytic theories for confidentiality and integrity use the data flows between responsibilities. A data flow expresses the passage of data from one responsibility to another and the connection is expressed as C^d and it is used in the confidentiality and integrity attribute architectures.

The analytic theory for maintainability uses the dependency between responsibilities. The dependencies show that a responsibility relies on some other responsiblity for its operation. But as shown in [80], there are different types of connectors in a dependency connector, namely: abstraction, binding, realization, substitution and usage. With the exception of *usage*, which shows a dynamic (run time) relationship among responsibilities, the other types of dependency show a static (compile time) structural relationship. A dependency is expressed as C^l and it is used in the maintainability attribute architectures.

Connectors are combined when attribute architectures are combined to derive a product

$\circ$	C^c	C^d	C^l	*null*
C^c	C^c	C^{cd}	$*$**See Table 6.3**	C^c
C^d	C^{cd}	C^d	$*$**See Table 6.3**	C^d
C^l	$*$**See Table 6.3**	$*$**See Table 6.3**	C^l	C^l
null	C^c	C^d	C^l	*null*

Table 6.2: Combination of Connectors

architecture. Connectors such as *control flow*, *data flow* and *dependency* that are found in separate attribute architectures are useful when analyzing quality attributes but only provide an incomplete view of the product architecture. The attribute architectures need to be combined for a complete product architecture.

One of three things can happen when we try to combine connectors. The two connectors can be of the same type and they are combined to a single connector of that type, two connectors of different but compatible types are combined and transformed into a new type of connector, or the two connectors cannot be combined and instead are left in place as a set of connectors. In the next section, we explore the specific details on how to combine the connectors found in the attribute architectures.

6.3.2 Composition Operator Defined

A composition operator is used to combine two connectors. As such, the composition operator needs to be defined for each possible pair of connector types. The composition operator assumes that the connections are unidirectional and indicate a relationship in the same direction. But the composition operator can also be used to combine bidirectional connections with other bidirectional connections.

The three types of connectors found in the attribute architectures for quality attributes in Table 6.1 can be combined as shown in Table 6.2. Combining C^c and C^d with C^l is presented in detail in table 6.3 as a dependency connector can represent more than one type of dependency.

We observe the following from Table 6.2 when we try to combine the connectors:

- When a control flow connector and a data flow connector are combined, a new connection called the control-data flow C^{cd} is created. For example, this would correspond to a method call with parameters at the implementation level.

$\circ$	C^c	C^d
C^l **(abstraction)**	$\times$	$\times$
C^l **(binding)**	$\times$	$\times$
C^l **(realization)**	$\times$	$\times$
C^l **(substitution)**	$\times$	$\times$
C^l **(usage)**	C^l	C^l

Table 6.3: Dependency Connector Combination Specifics

- The dependency connection C^l, in some cases, subsumes the control flow connector and the data flow connector when they are combined. This is because control flow and data flow can be considered as a type of dependency. The specific implementation level meaning of the dependency combinations are shown in Table 6.3.
- Quality attribute values as calculated from the attribute architectures must be recalculated if new connections of the same type are introduced from another attribute architecture as part of the combination. This is further explained in section 6.3.3.
- The composition operator is commutative as the order in which the connectors are combined do not influence the outcome of the combination. Combining more than two connectors involves pair-wise combinations that result in a same connector regardless of the order in which the combinations were applied.

Table 6.3 shows that some types of dependency connectors cannot be combined with some other connectors (as indicated by $\times$). Instead of a combination, such pairs are left in place and show a dynamic interaction dependency [86]. The list below elaborates on the pairs found in Table 6.3:

- C^l (abstraction) $\circ\, C^c = \times$

  This can be seen as a relationship where the parent responsibility is acting as a delegate for the child responsibility. The child responsibility is an instance of the parent responsibility and the control flow shows a delegation of control to the parent.
- C^l (abstraction) $\circ\, C^d = \times$

  This is a case where the child responsibility has a role of a *subject* and the parent responsibility has a role of an *observer*. This relationship can be found in the observer pattern.
- C^l (binding) $\circ\, C^c = \times$

  This can be found in the template method pattern. The child responsibility is bound to the parent responsibility as it implements the details as specified in the parent template. Also, the child responsibility transfers control to the parent responsibility as it contains the template method.

- C^l (binding) $\circ\, C^d = \times$

  The same as dependency binding with control flow but instead of control, data is sent to the parent responsibility.

- C^l (realization) $\circ\, C^c = \times$

  This relationship closely resembles the factory method pattern. The parent responsibility defines an interface (a specification) for the child responsibility to implement and there is a control flow to the parent to access the methods found in the parent.

- C^l (realization) $\circ\, C^d = \times$

  The same as dependency realization with control flow but instead of control, data is sent to the parent responsibility.

- C^l (substitution) $\circ\, C^c = \times$

  This relationship closely resembles the protection proxy pattern where a child responsibility is substituted for the parent responsibility as a proxy. The control flow represents the proxy communicating with the parent responsibility.

- C^l (substitution) $\circ\, C^d = \times$

  Similar to dependency substitution and control flow but with data.

- C^l (usage) $\circ\, C^c = C^l$

  The dependency connector subsumes the control flow connector. This is because a dependency of type usage includes a control flow.

- C^l (usage) $\circ\, C^d = C^l$

  The dependency connector subsumes the data flow connector. Again, this is because a dependency of type usage includes a data flow.

The attribute architectures can be combined using the observations we have presented in this section. By combining the connectors found in each individual attribute architecture into one representative architecture, we derive an architecture that is complete with combined connectors. As

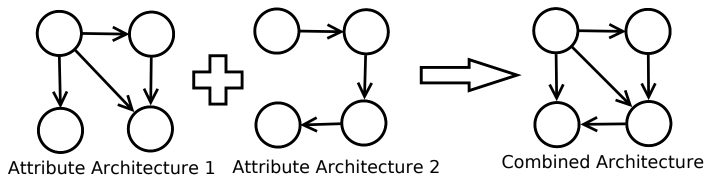


Figure 6.3: Combining Attribute Architectures

shown earlier, some connectors cannot be combined and must be left in place. In the next section, we explore how the quality attributes under consideration can change in the process of combining the attribute architectures.

6.3.3 Effects of Combining Attribute Architectures

Quality attribute values calculated from attribute architectures can change when new connections of the same type are introduced during the combination. This puts a constraint on combining attribute architectures because there are quality attribute goals that a software system has to satisfy.

Combination of attribute architectures can introduce new connections to one another as illustrated in Figure 6.3. Also, disparate connections from separate architectures can also be combined as shown in Section 6.3.2. The combination of disparate connectors can influence the quality attributes as well.

When introducing new connections where there was no connection (a null connection) in attribute architectures, there is a possibility that the quality attributes as calculated from the attribute architectures can change. Table 6.4 shows how replacing a null connection from one attribute architecture with another type of connector can influence the quality attributes under consideration.

The rows in the table represent attribute architectures that are introducing a new connection and the columns represent attribute architectures that are receiving the new connection. (The initials of the quality attributes are used to save space, A - availability, R - reliability, C - confidentiality, I - integrity, M - maintainability). A check mark indicates that the newly introduced connections do impact the quality attribute values of the receiving attribute architectures. A cross indicates that there is no impact.

	A	R	C	I	M
A	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
R	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
C	$\times$	$\times$	$\surd$	$\surd$	$\surd$
I	$\times$	$\times$	$\surd$	$\surd$	$\surd$
M	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Table 6.4: Effects of Combination on Quality Attributes

Table 6.4 shows that a newly added control flow connection can possibly affect all other quality attributes values under consideration. Control flow connections can create “entry points” to confidentiality and integrity attribute architectures via new connections thereby influencing confidentiality and integrity, and making recalculation necessary. Maintainability is also influenced by newly introduced control flow connections as a control flow is a type of dependency.

Newly introduced data flow connections do not influence availability and reliability, since data flow is not used in their computations. But maintainability is influenced by newly introduced data flow connections as a data flow is a type of dependency.

A newly introduced dependency connection can influence availability and reliability if the dependency type represents a control flow. It is difficult to determine if a dependency connection represents a control flow unless it is annotated. In cases where a dependency connection is not annotated, the architect would be required to determine the type of connection by examining the responsibilities involved in the connection. This is also true for dependency connections that represents data flow.

Quality attribute requirements put constraints on combining attribute architectures when null connections are converted to actual connections. But combined connections do not introduce any changes to the quality attribute under consideration as the configuration of the attribute architecture does not change when a connection is combined.

6.3.4 Summary

The combination of attribute architectures results in a changed configuration of the architecture that often requires the recalculation of the quality attribute values under consideration. As such, the combination of attribute architectures is an iterative process that requires the architect to change the attribute architectures if a combination results in a negative effect on other attribute

	Static	*Dynamic*
Implicit	Property	Mode
Explicit	Dependency (implementation, extends, binding)	Control Flow (Event, Event-Data) Data Flow (Data, Event-Data, Data-Access, Bus-Access)

Table 6.5: Relationship Types in AADL

architectures. In the next section, we present an example of how attribute architectures represented using AADL are combined.

6.4 Example with AADL

There are six types of connectors in AADL namely: data, event, event-data, data-access, bus-access and port-group connectors. Previously we have defined that there are three types of relationships in the quality attributes under consideration, namely: control flow, data flow, and dependency.

We can map the three types of relationships found in the attribute architectures to the connectors found in AADL. AADL connections are relationships that are found among AADL elements and can be broadly divided into implicit and explicit relationships. Implicit relationships are relationships that are present but not represented by specific visible constructs in the architecture. For example, having a "reliability value" property for an element is not a relationship that is explicitly shown by the architecture but nevertheless is implied by it. Explicit relationships are expressed using specific types of connectors or keywords. Table 6.5 shows a summary of the relationships found in AADL.

Control flow and data flow connectors have their equivalent connectors in AADL and the dependency connection can be expressed in AADL using syntactic constructs such as *implementation, extends,* and *binding.* Table 6.6 illustrates the AADL equivalent connectors for the combined connectors found in the attribute architectures. A port-group connector is used to express any set of connections that cannot be resolved to a single connection. For example, a dependency combined with any other connection results in a port group connection. Previously, in Table 6.3, we showed

$\circ$	C^c	C^d	C^l	$null$
C^c	Event	Event-Data	Port-Group	Event
C^d	Event-Data	Data	Port-Group	Data
C^l	Port-Group	Port-Group	Port	Port
$null$	Event	Data	Port	$null$

Table 6.6: AADL equivalent of Combined Connectors

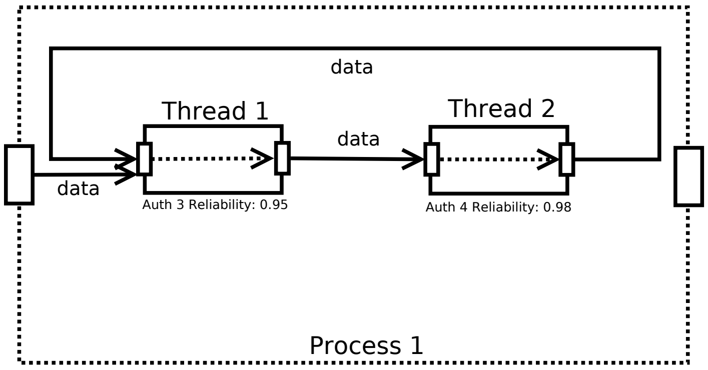


Figure 6.4: Confidentiality Attribute Architecture

that some dependency connections cannot be combined with other connections on a more abstract level.

Figure 6.4 shows a confidentiality attribute architecture with data connections between threads and Figure 6.5 shows a reliability attribute architecture. Each attribute architecture represents the responsibilities (which includes constructs such as threads) and the connections that are used to calculate quality attribute values. By applying an authorization level based analytic theory, we can calculate if any confidentiality requirement violations occur in figure 6.4.

The authorization level based analysis for the confidentiality attribute architecture restricts access to certain responsibilities based on authorization level. Assume a scenario in which the authorization level 4 is required. That would mean that an authorization level of 4 is required for the data flow to completely take place. Confidentiality requirement is violated if the data flow is permitted with a lower authorization level or denied even if the required authorization level is met. From Figure 6.4, we can see that the data flow can take place with an authorization level of 4 and thus satisfies the confidentiality requirement.

The reliability attribute architecture is analyzed using standard reliability calculation methods treating the reliability values as probabilities. For the scenario that involves Figure 6.5, the

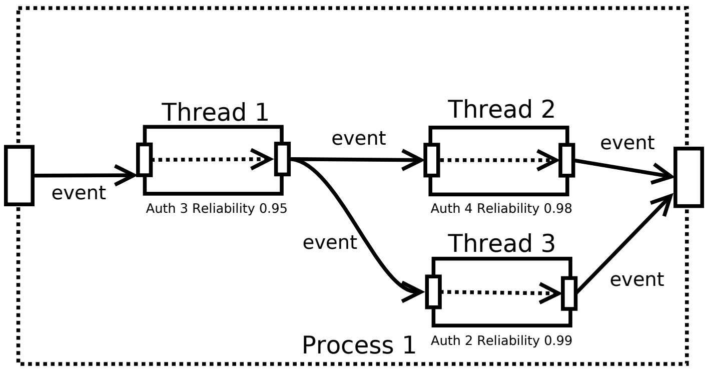


Figure 6.5: Reliability Attribute Architecture

reliability value that is required is 0.93. Equation 6.7 shows us the reliability values we calculate from the information from Figure 6.5 and we can observe that it meets the reliability requirement.

$$R = \frac{(0.95 \times 0.98) + (0.95 \times 0.99)}{2} = 0.936 \tag{6.7}$$

The two attribute architectures can be combined as shown in Figure 6.6. Here are the specific changes made during the combination. (the list omits detailed descriptions due to space concerns)

- When a data flow (data connection) and a control flow (event connection) are found between two elements with the same direction, the two connections are combined into an event-data connection (as per Table 6.6). The combination was performed as it is more convenient to have one connection instead of two.

- The data connection from thread 2 to thread 1 is removed as it decreases the reliability value to less than 0.95. (the data feeds back in a loop) In order to preserve the reliability requirement the explicit connection has been removed and instead the connection is routed to thread 3 where it can exit the process and be operated on (to check for errors) before it comes back into the process.

- The authorization level for thread 3 is raised to 4 as required by the confidentiality scenario under consideration. This can be accomplished by adding additional safeguards to thread 3.

In this section, we have seen how an attribute architecture would be combined in AADL.

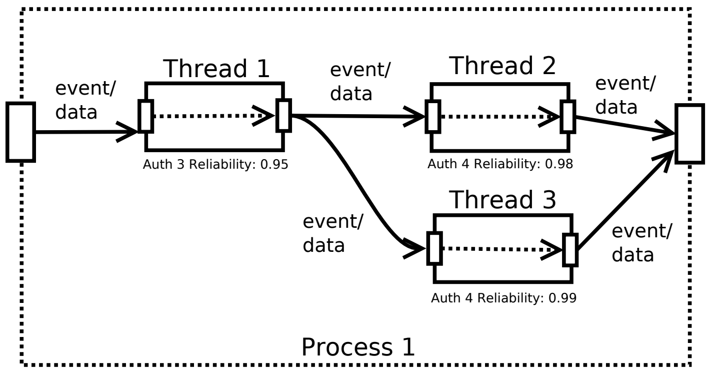


Figure 6.6: Combined Architecture

Because a combination can potentially cause the quality attribute values calculated from the analytic theories to change (as indicated in Table 6.4), combining attribute architecture becomes an iterative process of considering the trade-offs that are involved.

6.5 Summary

We have defined the composition operators that are necessary in order to combine attribute architectures of certain quality attributes. We have also seen how the quality attribute values from attribute architectures can change due to combination with other attribute architectures. An example combination of attrbute architectures was shown using AADL. These results were published in [49] and [48].

Combining the attribute architectures is an iterative process of combining the attribute architectures, recalculating the resulting quality attribute values, and adjusting the combined attribute architectures as needed. In the future work, we plan to extend the composition operators to include more quality attributes and their attribute architectures.

6.6 Assembling the Results from the Reasoning Frameworks

Because of the imprecise nature of the data and the implicit nature of the problem, qualitative reasoning can be used to help the architect reason about certain sub-attributes of dependability and show the architect quickly and efficiently the consequences of design choices. Both analytic and quantitative theories can be useful in designing a reasoning framework for dependability.

To illustrate how to combine the qualitative and quantitative reasoning frameworks for dependability, consider a CTAS webservice provider that plans to add support for heterogeneous middleware to increase its availability. Adding support for heterogeneous middleware would increase the availability of the system but it would reduce the maintainability of the system.

The emphasis on each sub-quality, within dependability, will be different based on the type of web service[2]. CTAS web services have different dependability goals than entertainment web services. A CTAS web service will be used to illustrate our approach. The components of dependability are described in order of priority below.

- Confidentiality:

  The CTAS web service requires that authentication, encryption, and access controls be implemented. If all these features are implemented, the system will be assumed to possess an appropriate level of confidentiality. If one or more features, such as password strength checking, is added beyond this basic set then we can say that the goal for confidentiality has been exceeded. If any one of the requisite features is not present then the goal has not been achieved.

- Integrity:

  The integrity of a CTAS web service can be thought of in terms of the percentage of transactions that are either completed or not executed at all. Our example CTAS web service has a goal that 98% of all transactions will fall into one of these two categories.

- Reliability:

  In our example CTAS web service, the goal is to have less than 10 errors a month for the entire system. An error is defined to be a failure of content delivery. If 8-12 errors occur a month, the reliability goal of the system has been met. The goal will be exceeded if less than 8 errors occur and deficient if more than 12. The system will be severely deficient if more than 16 errors occur a month.

- Availability:

  Our CTAS web service requires that it is available at least 90% of the time. It also requires the TTR (time-to-repair) to be one hour. If the availability is 85% or higher and the TTR is less than 1 hour 10 minutes, the service satisfies the availability goal of the system.

[2]According to [36], the "procedure of looking first within each attribute and then comparing across by some weighting system is superior to that of making global intuitive judgements across attributes regarding each choice in isolation."

Table 6.7: Scales of sub-qualities

Attribute	minimum	maximum	goal
Confidentiality	not clear	not clear	required controls implemented
Integrity	0%	100%	98%
Reliability	0%	100%	98%
Availability	0%	100%	90%

Table 6.8: Sub-qualities mapped to standard scale

Attribute	maximum	exceeds goal	goal	does not meet goal	minimum
Confidentiality	not clear	additional controls in place	all required controls in place	not all controls in place	no controls implemented
Integrity	100%	>98%	98%	<98%	0%
Reliability	100%	>98%	98%	< 98%	0%
Availability	100%	>90%	90%	< 90%	0%

This ranking, in the order listed above, reflects the relative importance placed on each sub-quality for the CTAS web service. Availability and reliability are important but not as important as confidentiality or integrity in the CTAS web service. The ranking does not tell us how much more important confidentiality is than integrity but it is useful when it comes to reasoning qualitatively about the system. For example, we can define rules that state that any tactics that can adversely affect a higher ranking sub-quality should not be implemented. It is not possible to automatically reason about design changes that positively affect integrity while negatively affecting reliability because of the dissimilar scales. Table 6.7 summarizes the scales used by the four dependability sub-qualities.

The mapping of the values from the scenario to the standard scale is summarized in Table 6.8. The state of the architecture design is such that the current quality profile is as shown in Table 6.9. Using the standard scale allows the architect to quickly understand the state of the architecture and to identify areas needing attention.

Consider the application of an architectural tactic to increase the coupling among components in order to increase performance in the CTAS web service. How will this tactic affect the dependability of the system? To reason about this change in architecture, we have to construct two model fragments. On the static model fragment, we would express that as the percentage of components that depend on each other increases (tighter coupling), the confidentiality of the system

Table 6.9: Current quality profile

Attribute	Sub-Attribute	Value
performance		meets goal
dependability		meets goal
	confidentiality	required controls in place
	integrity	98%
	reliability	8 errors per period
	availability	92%
maintainability		does not meet goal
accessibility		exceeds goal

decreases and the performance of the system increases [82]. (Figure 6.7) This is because increasing the coupling among components usually leads to an increase in security problems and performance.

On the process static model fragment, we would express that there is a positive influence that increases the percentage of components that are dependent on one another. When the simulation is run with these two model fragments, it will invariably tell us that the confidentiality will decrease and that the performance will increase. But what if there are some safeguards to offset the security problems that are brought about by tighter coupling among components? Safeguards such as isomorphic graph analysis and message digest comparisons to make sure that the integrity of the components are kept. Depending on whether these checks provide an increase in security that is equal, greater than or less than the amount of security that is taken away by tightly coupled components, the confidentiality and availability of the system will change. (Figure 6.8)

By combining the results from the quantitative and qualitative reasoning frameworks of dependability, a quality profile of the software system can be created to determine if the dependability goals are satisfied. More research needs to be done on how to combine and interpret the results from the reasoning frameworks for the sub-attributes of dependability.

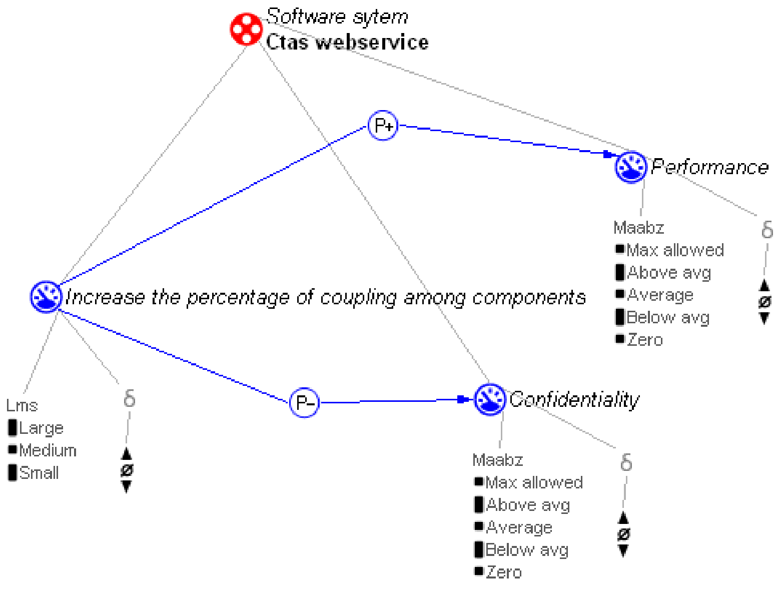


Figure 6.7: Garp 3 Static Model Fragment

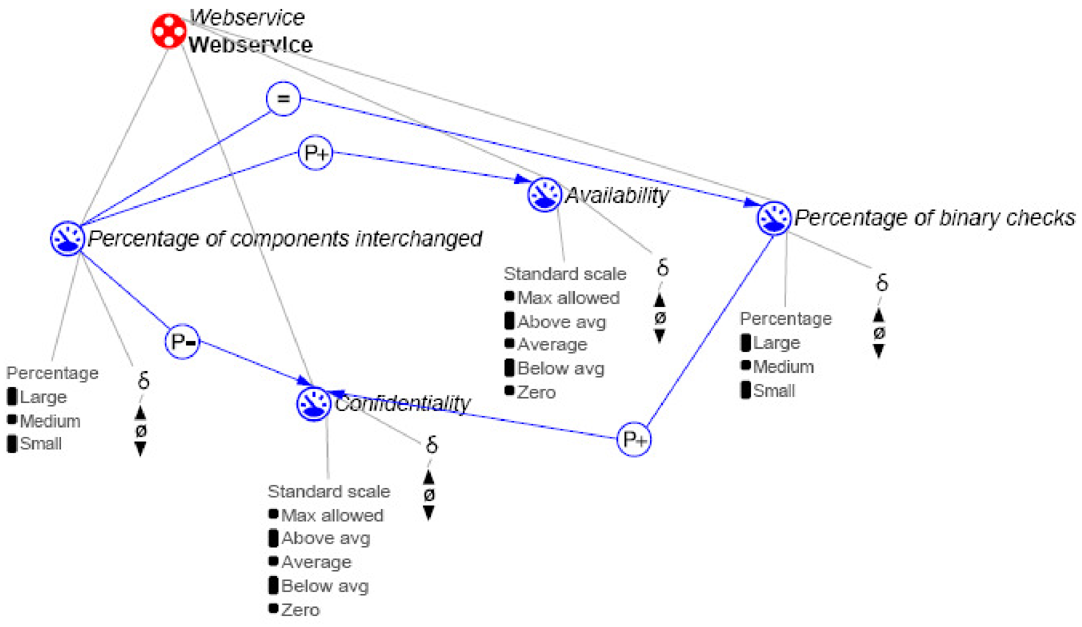


Figure 6.8: Garp 3 Modified Static Model Fragment

Chapter 7

Case Study on Apache

The dependability reasoning framework and the idea of combination of attribute architectures can be applied to open source software products such as the Apache http server. In this chapter, an application of the technique elaborated on the previous chapters is presented using the Apache http server.

7.1 Apache in AADL

The Apache server is an open source http server that is the most popular web servers in the world. The version of Apache current as of this writing is 2.2 and this analysis is based on it. The Apache server consists of the apache portable runtime, the multiprocessing module, the server core, application modules, and the client.

The Apache portable runtime is the API provided by the Apache server. It maps to the underlying operating system and provides platform independent functionality that makes Apache portable. The multi-processing module binds to the ports and dispatches children to handle incoming requests. It plays the most basic functionality of the server. Every Apache server has exactly one multi-processing module and it is optimized for a specific platform.

The server core is responsible for handling the basic functionality of the server such as redirecting requests, limiting resources, configuring settings, registering modules, and calling appropriate modules when necessary. It is also responsible for registering the information for each module. The modules are functionalities that can be plugged into the Apache server. The modules must first register a hook into the server core and, when the hook is called, the handler for that hook is executed (Handlers that change data are referred to as filters and those that generate content are called content handlers). The server core also sends configuration data to each module, which are required to start up the module.

The client sends and receives data to the server and the operating system underlying Apache provides the infrastructure for running Apache.. The Apache software architecture is shown in figures 7.1, 7.2, and 7.3 using AADL. The Apache http server was modelled in AADL with respect to responsibilities. In other words, the elements in the model respect responsibilities that can be derived from the documentation [33][57].

7.2 Parameters for Dependability analysis

The parameters that are generated are not verified but are values that are derived from secondary sources. The values can be different depending on the the assumptions made during the calculations and it should not be taken as an absolute value. Since the intention is to illustrate our technique, the values are not verified in the following sections. The illustration of the technique should not be considered as the only method of applying the dependability analysis technique as there are many ways to show the application of the technique for different software products.

In order to perform a dependability analysis on Apache we must first determine the various parameters that are required by the reasoning frameworks of dependability. The reliability reasoning framework requires that a reliability value is calculated for each responsibility in the architecture. One method of obtaining this value is to rely on the defect density of Apache. In a study [32], Apache 2.1 was found to have a defect density of 0.53 for each thousand lines of code. This translates to a reliability rate of 99.94% if a responsibility consists of a thousand line of code. The reliability rate will decrease if the size of the responsibility is larger.

The availability rate is a bit more difficult to determine for responsbilities. It is because availability analysis required the estimation of the time it takes to fix a failure. But in the analysis of the Apache http server, the time it take to recover from a failure can be estimated by measuring the time it takes to reboot and in some cases where a reboot is not necessary, a realistic estimate

```
system apache_portable_runtime
   features
      request_records: in out event data port;
      server_records: in out event data port;
      conn_records: in out event data port;
      request_pool: in out event data port;
      connection_pool: in out event data port;
   connections
      event data port apache_portable_runtime.request_records ->
                  multi_processing_module.os_interface;
      event data port apache_portable_runtime.server_records ->
                  multi_processing_module.os_interface;
      event data port apache_portable_runtime.conn_records ->
                  multi_processing_module.os_interface;
      event data port apache_portable_runtime.request_pool ->
                  multi_processing_module.os_interface;
      event data port apache_portable_runtime.connection_pool ->
                  multi_processing_module.os_interface;
end apache_portable_runtime

system multi_processing_module
   features
      os_interface: in out event data port;
   connections
      event data port multi_processing_module.os_interface ->
            os.os_interface;
      event data port multi_processing_module.os_interface ->
       server_core.process_request;
end multi_processing_module

system application_modules
   features
      moduleDataOut: out data port;  -- send module info
      configDataAccessIn: in event data port; -- receive config data
      hooksOut: out event data port;    -- register hook handler
      operateHandlerIn: in event data port; -- execute handlers
   connections
      data port application_modules.moduleDataOut ->
                  servercore.moduleDataIn;
      event data port application_modules.hooksOut ->
                  servercore.hooksIn;
end application_modules
```

Figure 7.1: Apache server's APR, MPM and Modules

```
system servercore
   features
      moduleDataIn: in data port;  -- receive module info
      configDataAccessOut: out data port; -- send config data
      hooksIn: in event data port;   -- register hook hanndler
      operateHandlerOut: out event data port; -- execute handlers
      -- sendContentHandler: out data port; -- send content to client
      process_request: in event data port;
      access_resource: out event data port;
   connections
      data servercore.configDataAccessOut ->
            application_modules.configDataAccessIn;
      event data servercore.operateHandlerOut ->
            application_modules.operateHandlerIn;
      event data servercore.access_resource ->
            apache_portable_runtime.request_records;
      event data servercore.access_resource ->
            apache_portable_runtime.server_records;
      event data servercore.access_resource ->
            apache_portable_runtime.conn_records;
      event data servercore.access_resource ->
            apache_portable_runtime.request_pool;
      event data servercore.access_resource ->
            apache_portable_runtime.conn_pool;
end servercore

system client
   features
      socket: in out data port;
   connections
      data port client.socket -> os.socket;
end client

system os
   features
      socket: in out data port;
      os_interface: in out even data port;
   connections
      data port os.socket -> client.socket;
      event data os.os_interface -> multi_processing_module.os_interface;
end os
```

Figure 7.2: Apache's server core, client and OS

```
system implementation Apache.i
   subcomponents
      the_client: system client;
      the_module: system modules;
      the_servercore: system servercore;
      the_database: system database;
   connections
      data port client.sendRequest ->
            servercore.receiveRequest;
      data port servercore.sendContentHandler ->
            client.receiveContent;
      event data port servercore.configDataAccessOut ->
            modules.configDataAccessIn;
      event data port servercore.operateHandlerOut ->
            modules.operateHandlerIn;
      event data port modules.moduleDataOut ->
            servercore.moduleDataIn;
      event data port modules.hooksOut ->
            servercore.hooksIn;
   flows
    register_module: end to end flow application_modules.moduleDataOut ->
            servercore.moduleDataIn ->
            servercore.configDataAccessOut ->
            application_modules.configDataAccessIn ->
            application_modules.hooksOut -> servercore.hooksIn;
    head_request_method: end to end flow client.socket -> os.socket ->
        os.os_interface ->
         servercore.multi_processing_module.os_interface -> servercore.;
    get_request_method: end to end flow post_request_method:
       end to end flow
         client.socket -> os.socket ->
         os.os_interface -> multi_processing_module.os_interface ->
         servercore.process_request ->
            application_module.operateHandlerIn ->
           servercore.process_request -> os.os_interface -> os.socket ->
           client.socket;

end Apache.i
```

Figure 7.3: Apache server's system in AADL

can be used. These values can be measured using OS specific tools. By measuring the MTBF (Mean time between failure) and MTTR (Mean time to repair) using data from such sources, we can calculate the availability rate for each responsibility. Also, some responsibilities might share the same availability rate as in some cases the responsibilities are very similar.

The authorization level is necessary for the confidentiality and integrity reasoning frameworks. Apache runs in two authorization levels: the root level and the user level. The main apache process runs at a root level and its child processes, which run at the user level, are responsible for listening to and answering requests from the client [27]. The separation in authorization level is there so that certain modules are prevented from accessing more sensitive parts of the system. Also, by running Apache in a *chroot* environment, Apache can have another level of authorization. By interpreting these parititioning schemes in a production Apache system, we can assign authorization levels for each responsibility in Apache.

The maintainability reasoning framework relies on the size of the responsibilities to be calculated. The size can be calculated in terms of source lines of code and the boundaries of a responsibility can be a package or a directory in a source code repository. Counting the source lines of code is simple once we know which source files are part of a responsibility.

7.3 Dependability analysis of Apache

To illustrate the dependability of the Apache server, we have translated the Apache architecture into AADL and identified Apache in terms of responsibilities. The next step is to apply the dependability reasoning framework to the use scenarios of Apache. The most important scenarios in using Apache are the methods that Apache implements, namely the HEAD, GET and POST methods and from the administration side, registering a module and fixing a defect in a responsibility are some of the scenarios that can be analyzed using the dependability reasoning framework. We illustrate dependability analysis using the following scenarios.

- Reliability: A client accesses a piece of information served by an Apache content generator (a type of module) (mod_info) with at least 99.5% reliability. (Get method)
- Availability: A client uploads a file to the Apache server, with at least an availability rate of 98%. (Post method).

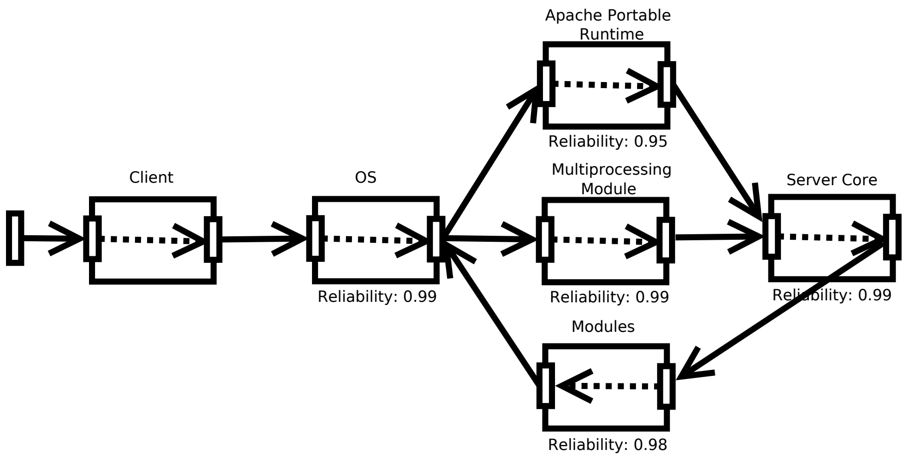


Figure 7.4: Apache Reliability Attribute Architecture

- Confidentiality: A client who is granted an authorization level of 3 tries to access content that is served by the CGI Apache handler (content generator).
- Maintainability: An organization modifies a multiprocessing module in order to adapt to the new hardware configuration of the server. In addition, they need to change mod_cgid in order to account for the change. The expected man-hours for this task is 2000 man-hours.

In analyzing safety, we choose all these scenarios as safety scenario as any one of the scenarios can be considered a safety scenario depending on the organization and execution environment.

The attribute architectures used to analyze each scenario such as figure 7.4 shows an end-to-end flow of the Apache AADL model. The responsibility “Module” in the attribute architectures represent the standard modules in Apache [25]. Also, the client is not part of Apache but is shown in the attribute architecture figures for a clearer representation of the scenarios.

7.3.1 Reliability

In order to analyze the reliabilty scenario in table 7.3, we take an end-to-end flow from the AADL representation of Apache in figure 7.3. The reliability scenario is the following: A client accesses a web page served by the Apache server with at least 99.5% reliability. This scenario corresponds to an end-to-end flow in the Apache AADL model. The end-to-end flow is showned in figure 7.4 and using the annotated reliability value. The reliability of the scenario can be calculated.

In order to calculate the reliability for each responsibility, we can use the reliability equation from section 4.1.1.

$$R_i = e^{-\int_0^{V_i t_i} \lambda_i(t)dt} \approx e^{-a_i c_i(t_i)} \tag{7.1}$$

To simplify the calculation, the number of faults in a responsibility a_i and the expected behavior of the code coverage resulting from executing the application with successive test cases $c_i(t_i)$ is used to calculate the reliability of the responsiblity [28]. The term $a_i c_i(t_i)$ is known as the mean value function of the component i which shows the average cumulative failures associated with time point t_i [73]. In order to calculate these values, we need to first estimate the number of defects in a responsibility.

The defect density per KLOC tells us the amount of defect that Apache contains per thousand lines of code. In a study, [32], it was discovered that Apache has a defect density of 0.53 per KLOC. This value is then multiplied by the size of the responsibility and the percentage of defect that causes a system failure, in order to estimate the number of faults in a responsibility. Calculating the defect density and the number of defects that cause failure is made easier for Apache because Apache is an implemented architecture but in instances where the system has not been implemented yet, there are estimation methods that can be used to predict the defect density [56].

The next step is to predict how many of these faults will occur in some time point t_i. In a study [77], the number of failures per given time period (for an arbitary MTBF of 24 hours) was calculated by computing the number of executions of the responsibility in that period and taking the reciprocal of it. Assuming that the responsibility gets executed a 100 times in 24 hours, the number of expected failures is $1/100 = 0.01$, a reasonable value considering that the responsibility would most likely be executed much more than that.

Table 7.1, shows the calculation for the reliability value for each responsiblity found in figure 7.4. The values are from the Apache 2.2.15 release and we assume that it has a defect density of 0.53 per KLOC. By extrapolating from the Apache bug statistics [26], we can estimate that about 0.04 % of the defects are critical defects that cause a system failure. Also, we assume that 0.01 failures occur in 24 hours as this is a reasonable figure to assume since most scenarios will be executed much more than a hundred times in a 24 hour period. These values were applied in equation 7.1.

Responsibility	**KLOC**	**Defects**	**Critical Defects**	**Reliability**
Operating System Related	0.6	0.32	0.013	0.9998
Apache Portable Runtime	227	120.31	4.81	0.9530
Multiprocessing Module	11.5	6.09	0.24	0.9976
Server Core	15.3	8.11	0.32	0.9968
Modules	78.4	41.55	1.66	0.9835

Table 7.1: Reliability Rate for the Responsibilities in Scenario

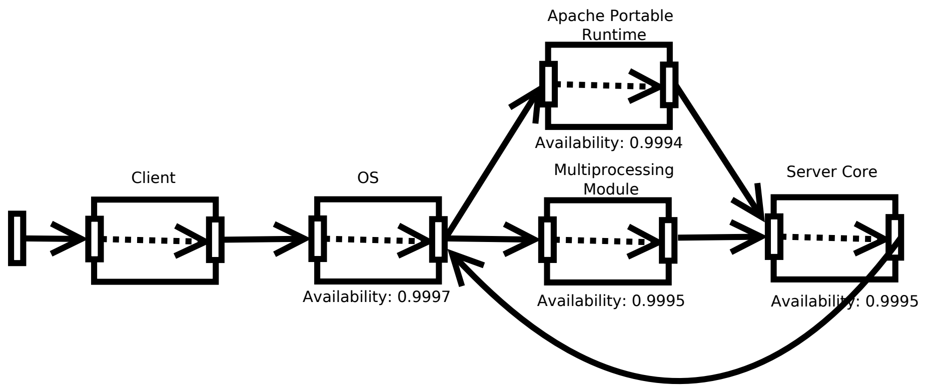


Figure 7.5: Apache Availability Attribute Architecture

The following equation shows how the reliability rate of the scenario can be calculated.

$$R = 0.9998 \times \frac{(0.9976 + 0.9530)}{2} \times 0.9968 \times 0.9835 \times 0.9998 = 0.9557 \tag{7.2}$$

This analysis shows that the scenario is not satisfied by the current Apache architecture. One way to satisfy it is to improve the defect density rate of each responsibility or change the use rate of each responsibility to increased the perceived reliability.

7.3.2 Availability

The following is an availability scenario: A client uploads a file to the Apache server, with at least an availability rate of 98%. (Post method). The attribute architecture used to analyze this scenario is shown in Figure 7.5. Unlike the reliability example which was a get method, the availability scenario is a post method that does not require any module as the default handler processes the Post request and returns the requested file.

As seen in chapter 4.2 the steady-state availability (availability over a sufficently long time)

Responsibility	**Availability**
Operating System Related	0.9997
Apache Portable Runtime	0.9994
Multiprocessing Module	0.9995
Server Core	0.9995
Modules	0.9995

Table 7.2: Availability Rate for the Responsibilities in Scenario

is calculated using the following equation [84].

$$\alpha = \frac{\text{MTBF}}{\text{MTBF + MTTR}} \tag{7.3}$$

The MTBF (Mean time between failure) values for each responsbility were previously derived in the calculation for the reliability of each responsibility. The MTTR (Mean time to repair) for each responsibility can be estimated by measuring the time it takes to recover from a failure in Apache. A failure in Apache can require a reboot of the server itself or it can require a reboot of the entire system. In the case where a reboot of the entire system is required, we can assume a value of 45 seconds (or 0.00052 as a percentage value using the 24 hour time period we are using), as it is the amount of time it takes to reboot a standard Fedora installation [65]. We will use this value as the MTTR for all responsibilities that cause a failure in Apache. This is reasonable estimate because a sole reboot of the Apache server will require a much shorter time period. In some cases, a reboot will not be able to repair a failure. This could be because of a logic error that causes a failure for a certain combination of input values. In such cases, the MTTR can be significantly larger than 45 seconds.

The following equation shows how the availability value of the “Apache Portable Runtime” responsibility is calculated.

$$\alpha = \frac{\text{MTBF}}{\text{MTBF + MTTR}} = \frac{0.9530}{0.9530 + 0.00052} = 0.9997 \tag{7.4}$$

Table 7.2 shows the availability values for each responsibility. We can see from the table that the availability of each responsibility is relatively high compared to the reliability values. This is because the MTTR is very small.

Responsibility	**Authorization Level**	**Privilege**
Operating System Related	2	Access system resource
Apache Portable Runtime	2	Access system resource
Multiprocessing Module	2	Access system resource
Processing Hooks	3	Access restricted resource
Server Core	2	Access restricted resource
Modules	3	Access restricted resource

Table 7.3: Availability Rate for the Responsibilities in Scenario

The following calculation shows how the reliability rate of the scenario is calculated.

$$R = 0.9997 \times \frac{(0.9994 + 0.9995)}{2} \times 0.9995 \times 0.9997 = 0.9984 \qquad (7.5)$$

The calculation shows that the Apache architecture satisfies the availability scenario's requirement of 0.98 availability. This was possible because the MTTR value was very low. Unless external conditions change to increase the MTTR values, the availability scenario will be satisfied.

7.3.3 Confidentiality & Integrity

The following is a confidentiality scenario: A client who is granted an authorization level of 3 tries to access content that is served by the CGI Apache handler (content generator). In this scenario, the client tries to access content served by a CGI handler. Table 7.3 shows the authorization level matrix of Apache. It is based on the sensitivity of the data or operation handled by the responsibility as mentioned in section 5.1. Not shown on the table are the privileges for authorization level 1 and 4 which stand for "No restriction" and "Access to execution environment" respectively. This assignment of authorization level is based on the typical privileges each responsibility in Apache is assigned [57].

In figure 7.6, there is processing hook that acts as a authentication module for the scenario. The user of the scenario is assigned an authorization level 3 and the authentication hook (auth_checker) verifies that the authorization level is 3 in order to authorize the scenario.

By examining the attribute architecture with respect to the authorization level granted to the user of the scenario, we can see that the scenario can be satisficed.

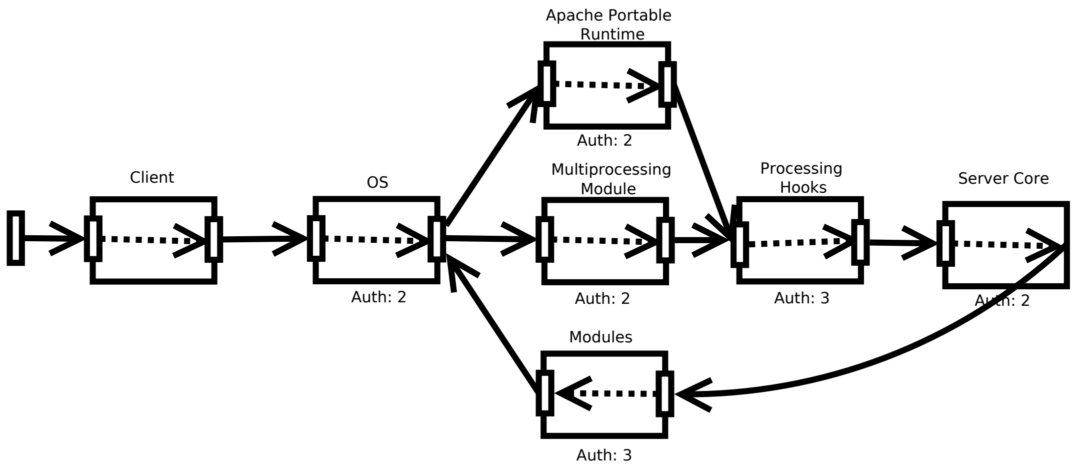


Figure 7.6: Apache Confidentiality Attribute Architecture

Responsibility	**KLOC**	**Percent Affected**	**Productivity**	**Cost**
Operating System Related	0.6	0	0.5	0
Apache Portable Runtime	227	0	0.5	0
Multiprocessing Module	11.5	0.1	0.7	1643
Server Core	15.3	0	0.5	0
Modules	78.4	0.01	0.7	1120

Table 7.4: Values for a Maintainability Scenario

7.3.4 Maintainability

The following is a maintainability scenairo: An organization modifies a multiprocessing module in order to adapt to the new hardware configuration of the server. In addition, they need to change mod_cgid in order to account for the change. The expected man-hours for this task is 2000 man-hours. The MPM is responsible for how threads and processes are created for the server. In our example scenario, a change in the operating environment has made a change in the MPM necessary and it has made a change in mod_cgid necessary. Figure 7.7 shows the maintainability scenario under consideration.

Because of the modular design of Apache, the maintainability scenario under consideration involves changing two responsibilities. If we assume a change percentage as shown in table 7.4 and a productivity of 0.7 sloc/hour based on past estimates [38], the following equation shows the man-hours required to carry out the scenario. From the calculation, we can see that the scenario requires

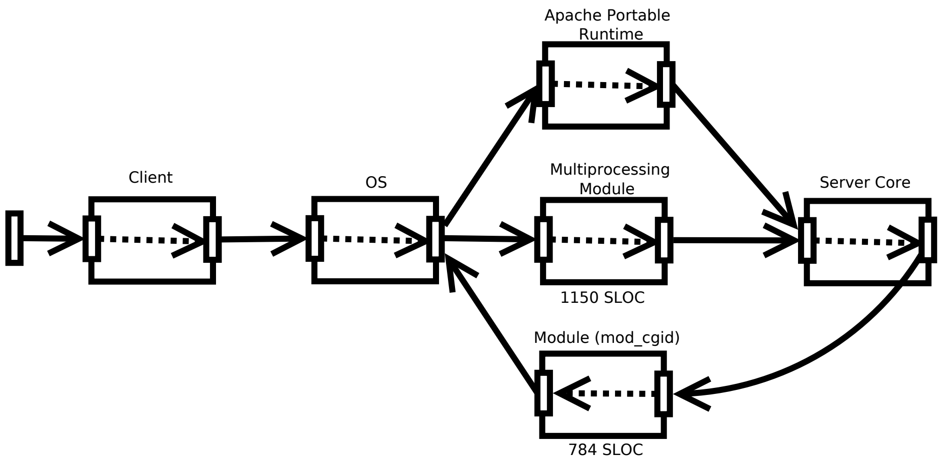


Figure 7.7: Apache Maintainability Attribute Architecture

Quality Attribute	**Hazard-affecting event**
Reliability	Incorrect output delivered from GET method
Availability	GET method is not in service
Confidentiality	Highly sensitive data is accessed by an unauthorized user
Integrity	Highly sensitive data is changed by an unauthorized user
Maintainability	Developers unable to patch a critical defect

Table 7.5: Events affecting Safety

763 man-hours more than what was expected.

$$E = \frac{11.5\text{ KLOC} \times 1000 \times 0.1}{0.7} + \frac{78.4\text{ KLOC} \times 1000 \times 0.01}{0.7} = 2763\text{ man-hours} \tag{7.6}$$

7.3.5 Safety

We consider four scenarios pertaining to Apache. These scenarios were specified and analyzed in section 7.3. From the FTA and FHA analysis of safety, we have identified the hazards associated for the identified scenarios in table 7.5.

The safety goal is to satisfy 80% of the safety scenarios under analysis. From each reasoning framework, we have found out that some scenarios are satisficed and some are not. Table 7.6 shows the result of the analysis for each scenario and the satisficing levels for each. Out of five scenarios, only three are satisfied

Scenario	**Result**	**Minimum level**	**Max level**
Reliability	Not Satisfied	0.995	0.999
Availability	Satisfied	0.98	0.998
Confidentiality	Satisficed	3	3
Integrity	Satisficed	3	3
Maintainability	Not Satisfied	2000	2763

Table 7.6: Safety Scenarios and its Analysis

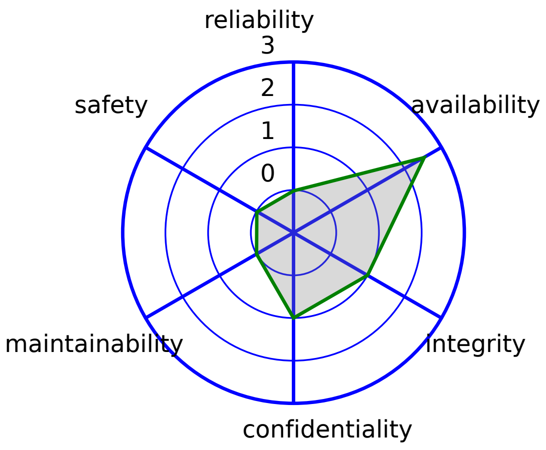


Figure 7.8: Star Plot of Apache Dependability Analysis

Figure 7.8 shows a star plot of the result of the safety analysis. It also shows the overall dependability analysis of Apache. The axis are numbered and mean the following: 0 Unsatisficed, 1 Minimum level satisficed, 2 Good level satisficed, 3 Maximum level satisficed.

As the star plot indicates, the architectural analysis of Apache shows that safety is not being satisficed and the overall dependability is not being satisficed as well.

7.4 Combination of attrbitue architectures in Apache

The combined attribute architecture is shown in figure 7.9. This figure is a graphical form of the AADL representation found in figures 7.1, 7.2 and 7.3. The combination rules described in chapter 6 were the rules that guide the combination of the attribute architectures of Apache.

For instance, some modules are called “filters” and have data flow connections but other modules such as “handlers” have a control/data flow. Although these are both modules, they have different connections and therefore the architectural representation shows a “control/data”

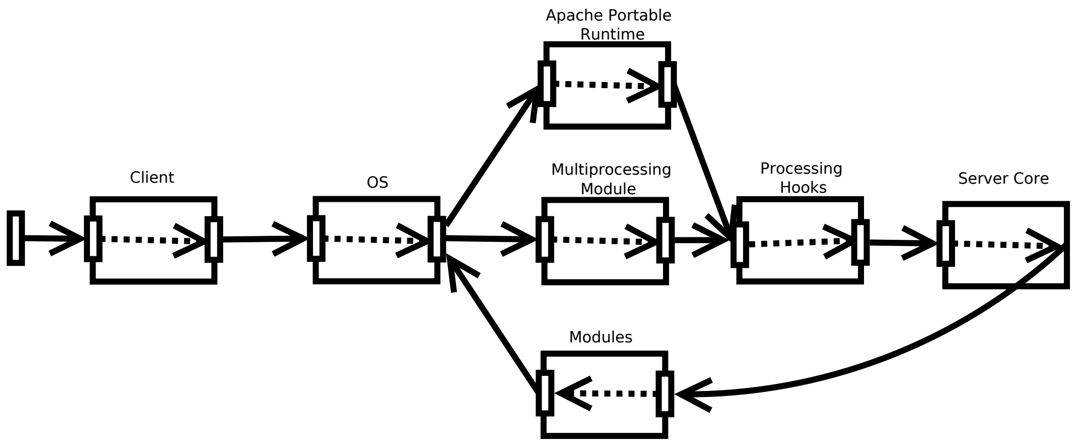


Figure 7.9: Combined Attribute Architecture of Apache

connection.

7.5 Conclusion

The application of the dependability reasoning framework and its techniques on a product such as the Apache http server shows that the framework is applicable to not only architectures that are in the design process but also to system that have already been implemented. This shows that the dependability reasoning framework may be used to justify values used in service level agreements.

Chapter 8

Implementation & Evaluation

In this chapter we explain how the reasoning frameworks were implemented in ArchE and present a user study conducted to evaluate its efficacy. The implementation details cover the structure of the application and the tactics for each sub-attribute of dependability.

8.1 Implementation Issues

The dependability reasoning frameworks were developed from scratch and integrated into ArchE 3.0. They are implemented using the following steps:

- Write a manifesto that defines a scenario type and the rest of the elements that constitute the particular quality attribute.
- Map the elements of the reasoning framework to the ArchE database.
- Subcless *ArchEReasoningFramework* and provide the implementations for the methods that correspond to the interaction commands.
- Codify a number of tactics and user questions for the corresponding quality attribute.

The next sections show the specific implementation details for the dependability reasoning frameworks.

8.1.1 Quantitative Reasoning Frameworks

Quantitative reasoning frameworks are implemented for reliability, availability and maintainability. The first step in implementing the reasoning frameworks is to define the parameters necessary for each analytic theory. Only one parameter is required for the responsibilities in the reliability reasoning framework. The parameter is called *P_Reliabilty* (which represent the reliability of the responsibility) and it is defined in the manifesto. For the availability reasoning framework, two parameters *P_MTBF* and *P_MTTR* which represent the mean time between failures (MTBF) and mean time to repair (MTTR) are defined for each responsibility. For the maintainability reasoning framework, two parameters representing the percentage of change in a responsibility *P_ChangePercentage* and the size of the responsibility *P_Size* are defined.

The next step is to define the relationship classes for each reasoning framework. A relationship class called *ArchEResponsibilitySequenceRelationVO* is defined for the reliability and availbility reasoning frameworks to represent a control flow from one responsibility to another. This is because reliability and availability share the same connector which is the control flow connnector. For the maintainability reasoning framework, the relationship class *ArchEResponsibilityDependencyRelationVO* is defined. It represents the dependency connector between two responsibilities. There are no parameters associated with the *ArchEResponsibilitySequenceRelationVO* relation as any control flow from one responsibility to another represents a perfect transfer of control. Any transfer of control that is less than perfect can be modeled with a responsibility. This is true for the *ArchEResponsibilityDependencyRelationVO* relation as well.

To calculate the system reliability or availability, a separate window is invoked. This window is implemented using an Eclipse Application Window (by extending the ApplicationWindow class) and its class name is *ReliabilityProfileView/AvailabilityProfileView* as this is where the utilization rate of each scenario is entered by the user. The utilization rate can also be entered in the window where the reliability/availability scenario is specified. The value field of the stimulus can be used to enter the utilization rate. When the system reliability/availability calculation window is invoked, the stimulus value field of the scenario is read to populate the utilization rate of the scenario. This may not be obvious to the user who does not know that the stimulus value field is used in that way. The algorithm to calculate the reliability/availability of a scenario is encapsulated in the class *ReliabilityAnalyzer/AvailabilityAnalyzer* and it uses the architectural representations provided by ArchE as input parameters. ArchE provides a list of *ArchEResponsibility* and *ArchERelation* which represent the responsibilities that are mapped to a scenario and the relationships between the responsibilities respectively. These two variables are used directly in calculating the reliability/availability of a scenario without using an intermediate representation. The maintainability reasoning framework does not have a ProfileView and a maintenance task involves one scenario.

Three reliability and availability tactics were implemented, namely: (i) Serialize Responsibilities; (ii) Combine Responsibilities; and (iii) Eliminate Responsibilities. These tactics apply to both reliability and availability and for maintainability, two tactics were implemented, namely: (i) Split Responsibility and (ii) Abstract Common Responsibilities. Table 8.1 details the tactics for each quantitative reasoning framework.

For each tactic there is a class that performs the analysis to find out the suitable changes that can improve the quality attribute under consideration and a class that actually performs the transformation. The first classes are a realization of the *TacticSolver* Java interface and the second classes are a realization of the *RFArchitecturalTransformation* Java interface. For example, the *SerializeResponsibilitiesCommand* class is in charge of actually serializing the candidate responsibilities that can be serialized. The *SerializeResponsibilitiesSolver* class is the solver that searches for the candidate responsibilities that can be serialized in order to improve reliability or availability. The other tactics shown in table 8.1 are implemented similarily. The solvers do not perform an exhaustive search but instead iterate over the dependency graph to find candidates that results in an improvement of quality attribute under consideration.

8.1.2 Qualitative Reasoning Frameworks

Qualitative reasoning frameworks are implemented for confidentiality, integrity and safety. The first step in implementing the reasoning frameworks is to define the parameters necessary for each analytic theory. For both the confidentiality and integrity reasoning frameworks, the parameter *P_AuthLevel* (which represents the authorization level required to operate the responsibility) is defined in the manifesto. A relationship class called *ArchEResponsibilityDataFlowRelationVO* is defined for both confidentiality and integrity as both shared the data flow connector. There are no parameters associated with the *ArchEResponsibilityDataFlowRelationVO* relation.

The safety reasoning framework is an amalgamation of the other reasoning frameworks and the *SafetyProfileView* class takes the results from the other reasoning frameworks, assembles it

Tactic Name	**Rationale**	**Screening Rules**	**Architectural Transformation**
Serialize Responsibilities (Reliability/Availability)	A non-redundant parallel connection is always less reliable than a serial connection of the same responsibilities.	Examine any non-redundant parallel connection that does not have any concurrency issues.	1. Identify parallel connection to serialize. 2. Examine each parallel path and check for concurrency issues between responsibilities. 3. If there are no concurrency issues, serialize the parallel connection by laying each parallel path in a serial connection.
Combine Responsibilities (Reliability/Availability)	If two or more responsibilities can be combined into one responsibility with a higher reliability, combining two or more responsibilities into one responsibility will always increase reliability for the scenario.	The candidate responsibilities for combination can be identified by placing all responsibility in a DSM and applying partitioning algorithms to identify responsibilities that refer to one another cyclically.	1. Place every responsibility of a scenario in a DSM (Dependency Structure Matrix). 2. Apply partitioning algorithms to identify responsibilities that are cyclically dependent. 3. Make sure that the identified responsibilities can be combined into a responsibility with a higher or equal to the average reliability. 4. Combine the identified responsibilities into one responsibility.
Eliminate Responsibilities (Reliability/Availability)	The elimination of a responsibility that has less than perfect reliability will always increase the reliability of the scenario.	Identify the responsibility that has the lowest reliability and eliminate it by using a different responsibility or a different architecture.	1. Identify the responsibility with the lowest reliability in the scenario. 2. Check if the responsibility is essential to the scenario or not. 3. If responsibility is not essential, eliminate the responsibility. 4. If responsibility is essential, replace the responsibility with another responsibility with a higher reliability or reconfigure the architecture to not use the responsibility.
Split Responsibility (Maintainability)	A responsibility that has too much capability is a significant contributor to the total cost. Refining that responsibility into smaller responsibilities can reduce the cost of change. For simplicity, we assume that after splitting, the changes will only affect a portion of the original responsibility.	Order the responsibilities in the dependency graph according to their cost and select the responsibility R that has the greatest cost as the target for splitting.	1. Create two child responsibilities R1 and R2 as refinements of R. 2. Remove originial responsibility R. 3. Specify a dependency relation between R1 and R2. 4. For each responsibility dependent on R, remove the dependency and define two new dependencies with R1 and R2 were appropriate.
Abstract Common Responsibilities (Maintainability)	Two responsibilities are affected by the same change, and both responsibilities have semantic coherence. Moving parts of the two responsibilities into one responsibility will reduce the total cost of the change.	Order the responsibilities in the dependency graph according to their individual cost. Then, pick the two responsibilities A and B that: i) have the greatest cost, and ii) identify two responsibilities are connected and have semantic coherence	1. Ask the user if responsibilities A and B share common functionality. 2. If the answer is positive, split responsibility A into children A and AB, and split responsibility B into children B and AB. 3. Create responsibility dependencies from A to AB and from B to AB. 4. For each responsibility R that had a dependency with A, redirect that dependency to A. 6. For each responsibility R that had a dependency with B, redirect that dependency to B.

Table 8.1: Tactics for Quantitative Reasoning Frameworks

Tactic Name	**Rationale**	**Screening Rules**	**Architectural Transformation**
Implement a secure pipe (Confidentiality/Integrity)	Introducing a connection that requires a certain authorization level can increase confidentiality/integrity.	Identify the responsibility where an unauthorized access takes place and create a respon sibility that represents a connection that connects it to other responsibili ties.	1. Order the responsibilities from the lowest to higher authorization level. 2. Ask user which responsibility R requires an increase in authorization level. 3. Add a new responsibility A which represents a connection to R. 4. Set A with an authorization level higher than R.
Implement a intercepting validator (Confidentiality/Integrity)	Introducing a responsibility that acts as a gate keeper to another responsibility improves confidentiality/integrity.	Identify a responsibility that needs to have its authorization level increased and introduce a responsibility that acts as a gate keeper to it.	1. Order the responsibilities from the lowest to higher authorization level. 2. Ask user which responsibility R requires an increase in authorization level. 3. Add a new responsibility A and route any connection for R through A. 4. Set A with an authorization level higher than R.
Raise authorization level (Confidentiality/Integrity)	Raising the authorization level of one or more responsibility increases confidentiality/integrity.	Identify one or more responsibilities that require an increase in the authorization level to satisfice a confidentiality/integrity scenario.	1. Order the responsibilities from the lowest to higher authorization level. 2. Ask user which responsibility R requires an increase in authorization level. 3. Set A with an authorization level higher than R.
Eliminate Responsibilities (Confidentiality/Integrity)	The elimination of a responsibility that permits access to another responsibility by having a lower authorization level results in higher confidentiality/integrity.	Identify one or more responsibilities that are allowing unauthorized access and eliminate it if possible.	1. Identify the responsibility with the lowest authorization level in the scenario. 2. Check if the responsibility is essential to the scenario or not. 3. If responsibility is not essential, eliminate the responsibility. 4. If responsibility is essential, replace the responsibility with another responsibility with a higher authorization level or reconfigure the architecture to not use the responsibility.

Table 8.2: Tactics for Qualitative Reasoning Frameworks

according to what the user deems as safety scenarios and finally produces a star plot that shows the safety analysis of the architecture under consideration.

As shown in table 8.2, four confidentiality/integrity tactics were implemented, namely: (i) Implement a secure pipe; (ii) Implement a intercepting validator; (iii) Raise authorization level; (iv) Eliminate Responsibilities. Confidentiality and integrity share these tactics because the two attributes are closely related as elaborated in section 5.1. Similar to the quantitative reasoning frameworks, each of these tactics have a class that implements the *TacticSolver* Java interface which searches for the suitable responsibilities for the transformation. The tactics also have a *Command* class that actually carries out the transformations according to the steps desribed in table 8.2. The user's decision is relied on heavily in these tactics and the solvers try to find a satisficing solution instead of performing an exhaustive search.

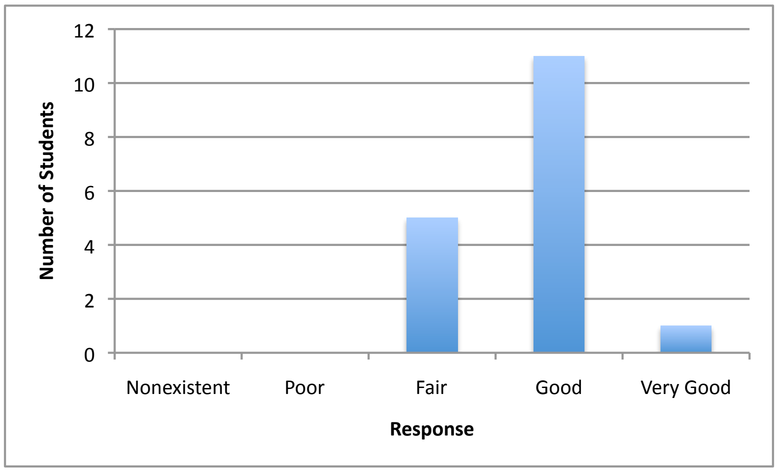


Figure 8.1: Self Rated Skill Level as Software Architect

8.2 Evaluation

A user study was conducted in the spring semester of 2010 in CPSC 875 (Software Architecture) to evaluate the effectiveness of using the dependability reasoning frameworks. A project was given to seventeen graduate students who were enrolled in CPSC 875 and they were asked to design a software architecture with dependability as a priority. A questionnaire was handed to the students to evaluate their experiences using the tools.

8.2.1 Introduction

The students were given a project called the Clemson Medical Chart System [45] which is an electronic system intended to replace paper based medical charts. Using the architectural knowledge that they learned in class, they were to design CMCS with dependability as the most important quality attribute for the system.

Figure 8.1 shows the self-assessed skill level of each student. The majority of the students responded that they are "Good" as software architects. The reason for this could be because the students were given CMCS as the final project for CPSC 875 and felt confident that they were able to design software architectures using the knowledge learned in class.

The students were also exposed to quality attributes during class but they were not familiar with dependability. Background on dependability was given during class and resources where made available [45] for them to learn from. But when asked if actually using the dependability reasoning

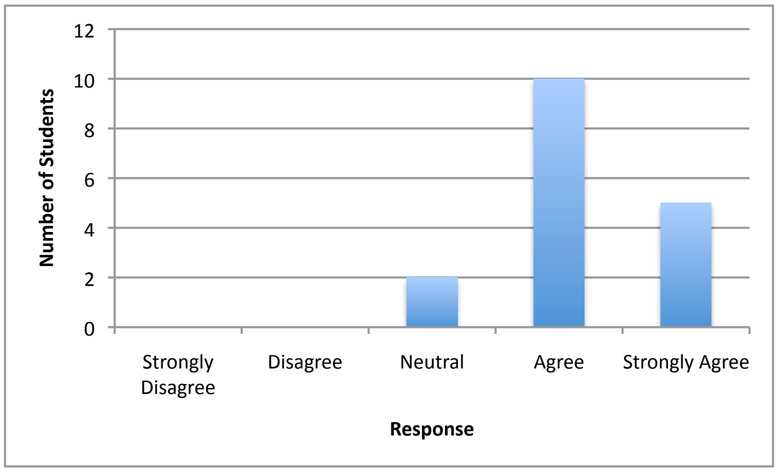


Figure 8.2: Did the framework help in learning about dependability?

framework helped them learn about dependability, they overwhelming responding that it did as shown in figure 8.2. This suggests that using the framework has been a learning experience.

8.2.2 Efficacy of the Dependability Reasoning Framework

CMCS is a system were dependability is critical and the students where asked if the framework helped them design CMCS with dependability as the most important attribute. As figure 8.3 indicates, the majority indicated that the framework has helped them design CMCS with respect to dependability. In the narrative questions on the questionnaire, students were asked which specific dependability reasoning framework was most helpful and which one was the least.

Availability, reliability and confidentiality reasoning frameworks were found to be most helpful but the maintainability reasoning framework was found to be the least helpful. Students indicated that the requirements of CMCS, such as the need for patient privacy, the need for the medical chart to be accurate and be present constantly, has made them use the availability, reliability and confidentiality reasoning frameworks and discover errors in their design. But they indicated that it was difficult to predict what types of maintenance activities would be required in CMCS and therefore made least use of it as all types of maintenance scenarios where deemed equally important.

The students also indicated that the dependability reasoning framework helped them in the architecture definition process and finding errors in their design as shown in figure 8.4. This suggests that the framework was also useful as a aid in the design process.

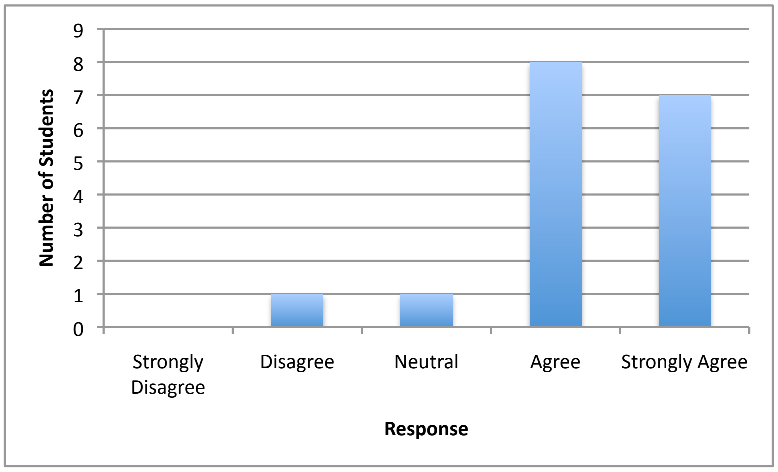


Figure 8.3: Did the framework help design CMCS with respect to dependability?

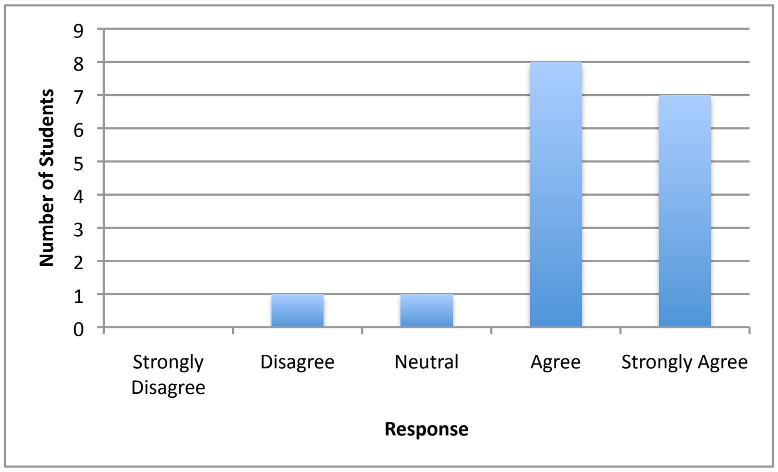


Figure 8.4: Did the framework help in the architecture definition process?

8.2.3 Usability of the Dependability Reasoning Framework

The students indicated that the installation of the tool chain was the most difficult part in using the framework. Various disparate parts such as Jess, Hibernate and Xmlblaster were required in the installation of the framework and many students indicated that the installation was the hardest part in using it. But once the framework was installed, most students indicated that using the framework was simple and straight-forward. Most have indicated that they would use it again in their future design efforts and recommend it to their peers.

8.2.4 Suggestions for the Dependability Reasoning Framework

Some students have indicated that using the framework could have been easier if the dependability reasoning framework accomodated probabilitic values or a range of values instead of a single value. They commented that they had to analyze the scenarios with many different values because the framework does not accomodate fuzzy values. They indicated that the incorporation of fuzzy values to the dependability reasoning framework would make their analysis more easier and complete.

There were suggestions on ArchE as well. Currently the largest complaint of the implementation of the dependability framework on ArchE is the installation process. Also, many have indicated that it would be helpful to export the analysis in a format (such as HTML) that can be shared among team members.

8.3 Conclusion

The implementation of the dependability reasoning framework has been documented in [19] and [46] and a user study was conducted to evaluate the efficacy of the reasoning framework. The user study suggests that the dependability reasoning framework is a useful tool in designing a software architecture with respect to dependability. More improvements in the installation process such as having a single installer for every required dependency and improvements in the usability aspects such as accomodating fuzzy values can be made in the dependability reasoning framework.

Chapter 9

Conclusions and Future Work

In this research a new approach to the design of dependable systems is presented. The focus was on the use of reasoning frameworks to guide the architect when using an attribute-driven approach. Two types of reasoning frameworks were presented: quantitative and qualitative. A new approach for reasoning about qualitative attributes was also presented. These reasoning frameworks were implemented in ArchE and a user study was conducted to evaluate it.

For a complete dependability reasoning framework the quantitative and qualitative sub-attributes of dependability were combined. The goal of this research was to provide a reasoning framework that combines the quantitative and qualitative aspects of dependability and show it as a working implementation in ArchE. These new techniques for reasoning about dependability can be used to build more dependable software systems. But currently, the installation and the usability aspect of the reasoning framework can be improved. Also, more tactics for each sub-attribute of dependability can be implemented in the reasoning framework to help the architect.

In the future, various types of work can be done using the dependability reasoning framework. The dependability reasoning framework can be used to evaluate the dependability of developed and developing software architectures. A promising application of the framework is in justifying the dependability of a software architecture in service level agreements (SLA). It can be used as evidence to prove how dependability is supported or show how it is lacking. For developing software architectures it can be used as a tool to guide the architect in building more dependable architectures. Numerous software architectures require the evaluation of its dependability as the reliance on SOA (service oriented architectures) grows. We also predict that the dependability reasoning framework will require updates in the future to reflect new improvements in the analytic theories that it relies on.

Bibliography

[1] R.S. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press Los Alamitos, CA, USA, 1996.

[2] F. Bachmann, L. Bass, and M. Klein. Deriving architectural tactics: A step toward methodical architectural design (cmu/sei-2003-tr-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003.

[3] F. Bachmann, L. Bass, and M. Klein. Preliminary design of arche: A software architecture design assistant (cmu/sei-2003-tr-021, ada421618). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003.

[4] M.R. Barbacci, R. Ellison, A.J. Lattanze, J.A. Stafford, C.B. Weinstock, and W.G. Wood. Quality Attribute Workshops (QAWs), 2003.

[5] M.R. Barbacci et al. *Quality Attributes.* Carnegie Mellon University, Software Engineering Institute, 1995.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison Wesley, 1998.

[7] L. Bass, J. Ivers, M. Klein, and P. Merson. Reasoning frameworks (cmu/sei-2005-tr-007). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005.

[8] P. Bengtsson and J. Bosch. Architecture level prediction of software maintenance. In *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*, pages 139–147, 1999.

[9] P.O. Bengtsson, N. Lassing, J. Bosch, and H. Van Vliet. Analyzing software architectures for modifiability. 2000.

[10] M.A. Bishop and M. Bishop. *Computer Security: Art and Science.* Addison-Wesley Professional, 2003.

[11] S.A. Bohner. Software change impacts-an evolving perspective. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 263–272, 2002.

[12] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2000.

[13] J. Bosch and P. Bengtsson. Assessing optimal software architecture maintainability. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 168–175, 2001.

[14] G. Brændeland and K. Stølen. Using model-based security analysis in component-oriented system development. *Proceedings of the 2nd ACM workshop on Quality of protection*, pages 11–18, 2006.

[15] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996.

[16] N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, and W.G. Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1), 2001.

[17] R. De Landtsheer and A. van Lamsweerde. Reasoning about confidentiality at requirements engineering time. *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 41–49, 2005.

[18] A. Diaz-Pace. ArchE Reasoning Framework Developers Guide.

[19] A. Diaz-Pace, T. Im, J.D. McGregor, and P. Bianco. Implementing external reasoning frameworks for arche. In *SEI TECHNICAL NOTE (Under Review)*, 2010.

[20] A. Diaz-Pace, H. Kim, L. Bass, P. Bianco, and F. Bachmann. Integrating quality-attribute reasoning frameworks in the ArchE design assistant. *Proceedings QoSA*, pages 14–17, 2008.

[21] R.J. Ellison, A.P. Moore, L. Bass, M. Klein, and F. Bachmann. *Security and Survivability Reasoning Frameworks and Architectural Design Tactics.* Carnegie Mellon University, Software Engineering Institute, 2004.

[22] P.H. Feiler, D.P. Gluch, and J.J. Hudak. The architecture analysis & design language (AADL): An introduction, 2006.

[23] P. Fenelon, JA McDermid, M. Nicolson, and DJ Pumfrey. Towards integrated safety analysis and design. *ACM SIGAPP Applied Computing Review*, 2(1):21–32, 1994.

[24] K.D. Forbus. Qualitative Reasoning. *CRC Handbook of Computer Science and Engineering*, pages 715–733, 1996.

[25] Apache Software Foundation. Apache http server version 2.0. `http://httpd.apache.org/docs/2.0/mod/`, 2010.

[26] Apache Software Foundation. Apache httpd bug statistics. `http://people.apache.org/~erikabele/httpd/bugstats/`, 2010.

[27] Apache Software Foundation. Starting apache. `http://httpd.apache.org/docs/2.0/invoking.html`, 2010.

[28] S. Gokhale, W.E. Wong, K. Trivedi, and JR Horgan. An analytical approach to architecture based software reliability prediction. *Proceedings of IEEE International Computer Performance and Dependability Symposium (IPDS)*, 1998.

[29] H. Goldstein. Who killed the virtual case file? `http://www.spectrum.ieee.org/print/1455`, 2005.

[30] K. Goseva-Popstojanova, K. Trivedi, and A.P. Mathur. How different architecture based software reliability models are related. *Proc. of the Fast Abstracts 11th IEEE Int'l. Symp. on Software Reliability Engineering (ISSRE 2000). San Jose, California*, 2000.

[31] K. Goševa-Popstojanova and K.S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2-3):179–204, 2001.

[32] R. Groenboom. Reflections on the quality of open source software. `http://www.artima.com/weblogs/viewpost.jsp?thread=21730`, 2010.

[33] B. Grone, A. Knopfel, R. Kugel, and O. Schmidt. Fmc - the apache modeling project. `http://www.fmc-modeling.org/projects/apache`, 2010.

[34] L. Grunske. Early quality prediction of component-based systems–A generic framework. *The Journal of Systems & Software*, 80(5):678–686, 2007.

[35] S. Gürses, J.H. Jahnke, C. Obry, A. Onabajo, T. Santen, and M. Price. Eliciting confidentiality requirements in practice. *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 101–116, 2005.

[36] R. Hastie and RM. Dawes. *Rational Choice in an Uncertain World: The Psychology of Judgment and Decision Making*. Sage Publications Inc Thousand Oaks CA., 2001.

[37] J. Heit. Impact of methods and mechanisms for improving software dependability on non-functional requirements.

[38] J.E. Henry and J.P. Cain. A quantitative comparison of perfective and corrective software maintenance. *Journal of Software Maintenance: Research and Practice*, 9(5):281–297, 1997.

[39] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Pearson Higher Education, 2004.

[40] M. Howard, D. LeBlanc, and Safari Tech Books Online. *Writing Secure Code*. Microsoft Press Redmond, Wash, 2003.

[41] D. Huizinga and A. Kolawa. *Automated defect prevention: best practices in software management*. Wiley-IEEE Computer Society Pr, 2007.

[42] W.S. Humphrey. Why every business is a software business. `http://www.awprofessional.com/articles/article.asp?p=25491`, 2002.

[43] IEEE. Ieee standard computer dictionary: A compilation of ieee standard computer glossaries. New York, NY, 1990.

[44] K. Im and J.D. McGregor. Debugging support for security properties of software architectures. In *Proceedings of CSIIRW 2009*, 2009.

[45] T. Im. Clemson medical chart system. `http://www.cs.clemson.edu/~tim/labs.html`, 2010.

[46] T. Im and J.D. McGregor. Security in the context of dependability. *Proceedings of CSIIRW 2007*, 2007.

[47] T Im and J.D. McGregor. Toward a reasoning framework for dependability. In *DSN 2008 Workshop on Architecting Dependable Systems*, 2008.

[48] T. Im and J.D. McGregor. Evaluating confidence levels for security scenarios in attribute architectures. In *6th Annual Cyber Security and Information Intelligence Research Workshop (CSIIRW 2010)*, 2010.

[49] T. Im and J.D. McGregor. Reasoning about attribute architectures. In *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE 2010)*, 2010.

[50] T. Im, S. Vullam, and J.D. McGregor. Reasoning about safety during software architecture design. In *Proceedings of the 19th International Conference on Software Engineering and Data Engineering (SEDE 2010)*, 2010.

[51] Y. Iwasaki. Real-world applications of qualitative reasoning. *Expert, IEEE [see also IEEE Intelligent Systems and Their Applications]*, 12(3):16–21, 1997.

[52] F. Jay and R. Mayer. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610:1990, 1990.

[53] J. Juerjens. *Secure Systems Development With UML*. Springer, 2005.

[54] B. Kaiser, P. Liggesmeyer, and O. Mäckel. A new component concept for fault trees. In *Proceedings of the 8th Australian workshop on Safety critical systems and software-Volume 33*, pages 37–46. Australian Computer Society, Inc. Darlinghurst, Australia, Australia, 2003.

[55] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE software*, 13(6):47–55, 1996.

[56] SJ Keene. Modeling Software R&M Characteristics. *ASQC Reliability Review, Part I and II*, 17(2&3):13–22, 1997.

[57] N. Kew. *The Apache modules book: application development with Apache*. Prentice Hall PTR, 2007.

[58] K.M. Khan and J. Han. Assessing Security Properties of Software Components: A Software Engineer's Perspective. *Proceedings of the Australian Software Engineering Conference (ASWEC'06)-Volume 00*, pages 199–210, 2006.

[59] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 1997.

[60] D. Lapierre and J. Moro. *Five past midnight in Bhopal*. Grand Central Publishing, 2002.

[61] J.C. Laprie. Dependability: from concepts to limits. *SAFECOMP'93: 12th International Conference on Computer Safety, Reliability and Security*, pages 157–168, 1993.

[62] NG Levenson. Software Safety in Embedded Computer Systems. *Communications of the ACM*, 34:34–46, 1991.

[63] N.G. Leveson. Safeware: System Safety and Computers. 1995.

[64] R.R. Lutz. Targeting safety-related errors during software requirements analysis. *The Journal of Systems & Software*, 34(3):223–230, 1996.

[65] LWN.net. Lpc: Booting linux in five seconds. `http://lwn.net/Articles/299483/`, 2010.

[66] B.A. Malloy, J.D. McGregor, A. Krishnaswamy, and M. Medikonda. An extensible program representation for object-oriented software. *ACM Sigplan Notices*, 29(12):38–47, 1994.

[67] J.D. McGregor. Arcade gamemaker pedagogical product line. `http://www.sei.cmu.edu/productlines/ppl/software_architecture_views.html`, 2007.

[68] J.D. McGregor. Clemson transit assistance system. `http://www.cs.clemson.edu/~johnmc/courses/cpsc875/Telematics.htm`, 2007.

[69] J.D. McGregor, F. Bachman, L. Bass, P. Bianco, and M. Klein. Using an Architecture Reasoning Tool to Teach Software Architecture. In *Proceedings 20th Conference on Software Engineering Education & Training (CSEE&T 2007)*, pages 275–282, 2007.

[70] J.D. McGregor, F. Bachmann, L. Bass, P. Bianco, and M. Klein. An Experience Using ArchE in the Classroom. *Software Engineering Institute (CMU/SEI-2007-TN-001)*, 2007.

[71] J.D. McGregor and T. Im. A Qualitative Approach to Dependability Engineering. *Proceedings of Dagstuhl Seminar 07031*, 2007.

[72] N.R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, pages 178–187. ACM New York, NY, USA, 2000.

[73] J.D. Musa, A. Iannino, and K. Okumoto. *Software reliability: measurement, prediction, application (professional ed.).* McGraw-Hill, Inc. New York, NY, USA, 1989.

[74] NASA. Nasa software safety standard. http://satc.gsfc.nasa.gov/assure/nss8719_13.html, 2009.

[75] M. Newman. Software errors cost u.s. economy 59.5 billion annually. `http://www.nist.gov/public_affairs/releases/n02-10.htm`, 2002.

[76] A. Onabajo and J.H. Jahnke. Modelling and Reasoning for Confidentiality Requirements in Software Development. *Proc. of ECBS*, 6:460–467, 2006.

[77] E. Ovaska, A. Evesti, K. Henttonen, M. Palviainen, and P. Aho. Knowledge based quality-driven architecture design and evaluation. *Information and Software Technology*, 2009.

[78] R. Roshandel. Calculating architectural reliability via modeling and analysis. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 69–71, 2004.

[79] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives.* Addison-Wesley Professional, 2005.

[80] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The.* Pearson Higher Education, 2004.

[81] RS Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[82] B. Schneider. *Beyond Fear: Thinking Sensibly About Security in an Uncertain World.* Copernicus Books, 2003.

[83] G.G. Schulmeyer and J.I. McManus. *Handbook of software quality assurance.* Prentice Hall PTR Upper Saddle River, NJ, USA, 1998.

[84] J. Scott, R. Kazman, and CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST. Realizing and Refining Architectural Tactics: Availability. Technical report, 2009.

[85] C. Shelton and C. Martin. Using Models to Improve the Availability of Automotive Software Architectures. *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, 2007.

[86] J.A. Stafford, A.L. Wolf, et al. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–451, 2001.

[87] E.B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, pages 492–497. IEEE Computer Society Press Los Alamitos, CA, USA, 1976.

[88] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley, 2002.

[89] T. Warns. Engineering intrusion-tolerant software systems. Dagstuhl Workshop, Dagstuhl Germany, 2005.

[90] A. Welling and A. Burns. Real-Time Systems and Their Programming Languages, 1996.

[91] PJ Wilkinson and TP Kelly. Functional hazard analysis for highly integrated aerospace systems. In *IEEE Certification of Ground/Air Systems Seminar (Ref. No. 1998/255)*, page 4, 1998.

[92] R. Wirfs-Brock and A. McKean. *Object Design: Roles, Responsibilities, and Collaborations.* Boston, MA: Addison-Wesly, 2003.

[93] W. Wu and T. Kelly. Safety tactics for software architecture design. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 368–375, 2004.

[94] L. Xu, H. Ziv, T.A. Alspaugh, and D.J. Richardson. An architectural pattern for non-functional dependability requirements. *The Journal of Systems & Software*, 79(10):1370–1378, 2006.