Carnegie Mellon University
Software Engineering Institute

Robustness

Rick Kazman
Phil Bianco
Sebastián Echeverría
James Ivers

March 2022

TECHNICAL REPORT
CMU/SEI-2022-TR-004
DOI: 10.1184/R1/16455660

Software Solutions Division

http://www.sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100

DM21-0920

Table of Contents

List of Figures

List of Tables

Abstract

This report summarizes how to systematically analyze a software architecture with respect to a quality attribute requirement for robustness. The report introduces the quality attribute of robustness and common forms of robustness requirements for software architecture. It provides a set of definitions, foundational concepts, and a framework for reasoning about robustness and the satisfaction of robustness requirements by an architecture and by a system that realizes the architecture. It describes a set of architectural mechanisms—patterns and tactics—that are commonly used to satisfy robustness requirements. It also provides a set of steps that an analyst can use to determine whether an architecture documentation package provides enough information to support analysis and, if so, to determine whether the architectural decisions made contain serious risks relative to robustness requirements. An analyst can use these steps to determine whether those requirements, represented as a set of scenarios, have been sufficiently well specified to support the needs of analysis. The reasoning around this quality attribute should allow an analyst, armed with appropriate architectural documentation, to assess the robustness risks inherent in today's architectural decisions, in light of tomorrow's anticipated needs.

1 Goals of This Document

This document serves several purposes. It is

- an introduction to the quality attribute of robustness and common forms of robustness requirements
- a description of a set of mechanisms—patterns and tactics—that are commonly used to satisfy robustness requirements
- a means for an analyst to determine whether an architecture documentation package provides enough information to support analysis and, if so, to determine whether the architectural decisions made contain serious risks relative to robustness requirements
- a means for an analyst to determine whether those robustness requirements, represented as a set of scenarios, have been sufficiently well specified to support the needs of analysis

This document is one in a series of documents that, collectively, represent our best understanding of how to systematically analyze an architecture with respect to a set of well-specified quality attribute requirements [Kazman 2020a, 2020b]. The purpose of this document, as with all the documents in this series, is to provide a workable set of definitions, core concepts, and a framework for reasoning about quality attribute requirements and their satisfaction (or not) by an architecture and, eventually, a system. In this case, the quality attribute under scrutiny is *robustness*. The reasoning around this quality should allow an analyst, armed with appropriate architectural documentation, to assess the risks inherent in today's architectural decisions in light of tomorrow's anticipated tasks.

There are several commonly used and documented views of software and system architectures [Clements 2010]. The Comprehensive Architecture Strategy, for example, proposes four levels of architecture, each of which may be documented in terms of one or more views [Padilla 2019]:

1. functional architecture: The Functional Architecture provides a method to document the functions or capabilities in a domain by what they do, the data they require or produce, and the behavior of the data needed to perform the function.
2. hardware architecture: A Hardware Architecture specification describes the interconnection, interaction and relationship of computing hardware components to support specific business or technical objectives.
3. software architecture: A Software Architecture describes the relationship of software components, and the way they interact to achieve specific business or technical objectives.
4. data architecture: A Data Architecture provides the language and tools necessary to create, edit, and verify Data Models. A Data Model captures the semantic content of the information exchanged.

The focus of this document is almost entirely on the *software* architecture because a software architecture is the major carrier of and enabler of a system's driving quality attributes. And since software typically changes much more frequently than hardware, it is often the focus of maintenance effort. There will, however, be implications of architectural decisions made on each of the other views.

In addition, other important decisions within a project will impact robustness—or any other quality attribute, for that matter. Even the best architecture will not ensure success if a project's governance is not well thought out and disciplined; if the developers are not properly trained; if quality assurance is not well executed; and if policies, procedures, and methods are not followed. Thus, we do not see architecture as a panacea but rather as a necessary precondition to success, and one that depends on many other aspects of a project being well executed.

As we will show, there is not one single way to analyze for robustness. One can (and should) analyze for robustness at different points in the software development lifecycle, and at each stage in the lifecycle this analysis will take different forms and produce results accompanied by varying levels of confidence. For example, if there are documented architecture views but no implementation, the analysis will be less detailed and there will be less confidence in the results than if there were an existing implementation that could be scrutinized, tested, and measured. We will return to this issue of types of analysis and confidence in their outputs several times in this document.

2 On Robustness

Robustness has traditionally been thought of as the ability of a software-intensive system to keep working, consistent with its specifications, despite the presence of internal failures, faulty inputs, or external stresses, over a long period of time. Robustness, along with other quality attributes such as security and safety, is a key contributor to our trust that a system will perform today and tomorrow in a reliable manner. In addition, the notion of robustness has more recently come to encompass a system's ability to withstand changes in its stimuli and environment without compromising its essential structure and characteristics. In this latter notion of robustness, systems should be malleable, not brittle, with respect to changes in their stimuli or environments. As such it is a highly important quality attribute to design into a system from the start, as it is unlikely that any nontrivial system could achieve this quality without conscientious and deliberate engineering. This is why we are interested in understanding robustness and how it is supported by appropriate architectural decisions.

This report begins with a survey of definitions for robustness. We introduce a set of *quality attribute scenarios*, including a *general scenario*, to define robustness requirements more precisely. This is followed by a discussion of the mechanisms that can be employed in a software architecture to promote robustness. And we conclude with a discussion of the various ways that an analyst can analyze for robustness, focusing on analysis checklists and analysis models and methods.

We create definitions for software and system quality attributes, like robustness, so that we can label and categorize quality requirements. These labels are then used by several groups during the development phase. Stakeholders and requirements engineers use the labels during requirements elicitation to create checklists, to assess coverage and completeness, and to collect similar requirements. This group is often concerned with *why* the software must be robust.[1] A second group using the quality attribute labels is architects, who use the labels to identify the relevant parts of the design body of knowledge to help them choose and instantiate mechanisms that promote the desired quality and satisfy the requirement. Finally, analysts and evaluators use the labels to choose methods to apply to validate and verify that the requirement is achieved. These latter groups are usually less concerned with the *why* of the requirement and more concerned with the scope and impact of what must be robust and constraints on how the robustness will be achieved. Also, these groups need enough precision in the requirement definition that it is actionable and verifiable.[2]

How can we define robustness? As with many notions in software, we take inspiration and guidance from more traditional areas of engineering. In civil engineering, for example, "robustness is taken to imply tolerance to damage from extreme loads or accidental loads, although the framework here is applicable to other adverse effects such as sensitivity to human error and deterioration" [Baker 2008].

[1] More formally, they are concerned that a requirement is *necessary* and *appropriate* [BKCASE 2018, Table 3].

[2] More formally, they are concerned that the requirements are *unambiguous*, *complete*, and *verifiable* [BKCASE 2018, Table 3].

But robustness is certainly an important quality of software systems. Gerald Jay Sussman, in his essay "Building Robust Systems: An Essay," defines robust systems as "systems that have acceptable behavior over a larger class of situations than was anticipated by their designers" [Sussman 2007]. He goes on to say that "the most robust systems are evolvable: they can be easily adapted to new situations with only minor modification." Finally, and perhaps most troubling considering the subject of this report, he notes that "common practice of computer science actively discourages the construction of robust systems." This is because

> *[i]n software engineering we are taught that the "correctness" of software is paramount, and that correctness is to be achieved by establishing formal specification of components and systems of components and by providing proofs that the specifications of a combination of components are met by the specifications of the components and the pattern by which they are combined. I assert that this discipline enhances the brittleness of systems. In fact, to make truly robust systems we must discard such a tight discipline. [Sussman 2007]*

A literature survey reviewed approaches to software robustness [Shahrokni 2013], and this term (and related terms such as "dependability" [Avizienis 2001]) also appears in quality attribute taxonomies such as that of the International Organization for Standardization (ISO) [ISO/IEC 2011]. However, the definitions used in these taxonomies are somewhat broad and the terminology varies. While ISO 25010(E) does not define robustness, it does define reliability as the "degree to which a system, product or component performs specified functions under specified conditions for a specified period of time." This concept is closely related to the following concepts:

- *maturity: degree to which a system, product or component meets needs for reliability under normal operation*
- *availability: degree to which a system, product or component is operational and accessible when required for use*
- *fault tolerance: degree to which a system, product or component operates as intended despite the presence of hardware or software faults*
- *recoverability: degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system [ISO/IEC 2011]*

Similarly, the SWEBOK (*Guide to the Software Engineering Body of Knowledge*) [SWEBOK 2014] does not mention robustness at all, but it does mention several related terms such as reliability and fault tolerance. Avizienis and colleagues define robustness as "dependability with respect to erroneous input" [Avizienis 2001].

Synthesizing these definitions, we can make some statements about robustness as a quality attribute. A system is "robust" if it

1. has acceptable behavior in normal operating conditions over its lifetime
2. has acceptable behavior in stressful environmental conditions (e.g., spikes in load)
3. can recover from or adapt to states that are outside its proper operating specification
4. can evolve and adapt to changes in its environment and stimuli with only minor changes

The fourth point deserves some elaboration. Ideally a robust system can handle changes in its environment and stimuli that we cannot anticipate. If a system needs to be easily extended—such as

to accommodate new data sources, new data types, and new sensors—then it can be said to be robust with respect to those stimuli. If a system scales linearly with spikes in demand with little effect on latency or throughput, then it can be said to be robust with respect to those spikes. If a system can accommodate changes in its environment—perhaps a platform upgrade—with relatively little expenditure of time and effort, and with few new bugs generated, it can be said to be robust with respect to that kind of maintenance stimulus. But each of these kinds of robustness involves other quality attributes—extensibility, performance, and maintainability. Robustness, viewed in this expansive way, potentially cuts across *all* quality attributes and is really the system's ability to respond appropriately in the face of changing requirements and environments. Thus, this notion of robustness is not localized to a single quality attribute. While these notions of robustness are clearly important, they are not the focus of this report.

Finally, while the definitions of robustness provided above are helpful and provide context and scope for a robustness requirement, they do not have any criteria for satisfaction. For example, while the definitions suggest that it is important to *measure* robustness, they do not specify any specific measures. How can we say that one architecture is more robust than another? How can we say that an architecture is sufficiently robust? Further, the definitions do not distinguish among robustness challenges. For any system, some operating conditions will be easy to deal with, while others will be more difficult. Thus, these definitions allow us to talk about robustness in general, but to specify requirements for robustness we need a more precise definition.

In the field of software quality attributes, we can use *quality attribute scenarios* to create operational definitions. The next section defines quality attribute scenarios for the quality attribute of robustness.

3 Evaluating the Robustness of an Architecture

While we can precisely evaluate the robustness of an existing *implementation* of an architecture—by examining its bugs, faults, and failure history—we do not have this historical record when evaluating new architectures. Thus, we must analyze and evaluate new architectures in terms of their discernable characteristics. We use concrete scenarios to guide this analysis.[3]

We cannot precisely evaluate the robustness of an architecture any more than we can evaluate its performance, availability, or integrability. All quality attribute names are categories, and categories are too imprecise to be used for evaluation. Thus, we are better served by speaking of and measuring the robustness of an architecture, or a major subsystem, with respect to a set of anticipated and unanticipated faults or failures. And we specify these as scenarios. We will define a set of robustness scenarios in Section 4 as examples of the kinds of faults or failures that a system might be subjected to, and we will use these scenarios in our architecture analysis playbook (in Section 7).

To understand what it means to measure the robustness of a system, we need to understand the things that are involved in detection and recovery from faults and failures. To this end, we have surveyed the techniques that the software engineering research literature has proposed for robustness.

It is important to reiterate that while we restrict our attention in this section to analyzing *architectural* information for robustness, historically robustness analyses have focused on richer sets of information derived from a project's logging, health monitoring, error handling, and history. The advantage of these richer sets of information is that we can potentially create more precise analyses. The disadvantage is that, to acquire such rich information, we need to build, deploy, and actually observe the system. At that point it can be very expensive and time consuming to mitigate problems relating to robustness. Thus, our objective in analyzing an *architecture* for robustness is to find a sweet spot wherein we can gain insight into the potential robustness characteristics of a system before much, if any, code has been committed.

3.1 Measuring Robustness

When considering the robustness of a system, we must consider not only how resistant it is to failure but also the support that it provides to detect and recover from failures. Thus, we typically consider several system-level measures of robustness such as

- mean time between failures (MTBF) – a prediction, based on historical data, of how much time can be expected to elapse between system failures
- mean time to repair (MTTR)/estimated time of repair (ETR) – the time from a failure to when the system is once again operating according to its service-level agreement

[3] In this report we primarily focus on the aspects of robustness dealing with failures. For the aspects of robustness dealing with future changes, see the sidebars on "Designing for Unknown Unknowns" and "Architecting for the Unknown with Growth and Exploratory Scenarios" in Section 4, as well as the sidebar on "Assessing Brittleness" in Section 7. In addition, this topic is addressed in our technical report *Maintainability* [Kazman 2020b].

- availability/uptime percentage (the "nines") – a measure of the orders of magnitude of a system's predicted uptime

But these measures only give insight into the overall behavior of the system, from a robustness perspective. We can refine such measures into a more detailed set of questions about a system's architecture. Consideration of such questions and their measures, and analysis of the architectural decisions that led to those measures, can lead to improved architectures.

A set of architectural robustness concerns, the questions related to those concerns, and potential measures that shed light on each of these questions are summarized in Table 1.

Table 1: Robustness Concerns, Questions, and Example Measures

Concern	Supporting Questions	Example Measure
Prevention of faults or failures	How many components have critical operational thresholds defined and monitored? Does the system provide predictive modeling for thresholds (e.g., resource utilization)? Does the software use mature components that are known to be reliable? Does the system use hardware components that are known to be reliable?	• Software MTBF • Hardware MTBF • Number of false positives for predictive analysis • Number or percentage of faults masked by detecting when the system is close to failure threshold
	Does the system quarantine unanticipated requests or bad inputs?	• % of confirmed unanticipated requests and invalid inputs quarantined
	Does the system support retry?	• Number of retries needed or allowed • Percentage of operations or requests that need retries • % of retries successful
Detection of faults, error conditions, unsafe operating conditions	How many critical components can be monitored?	• % of system components that are monitored • % of components that support runtime diagnostics
	How effective is the monitoring?	• % of system faults or failures that are detected • Time to detect fault or failure or mean time to discover (MTTD) • Number of false positives
	How complete is the fault modeling (e.g., failure modes and effects analysis [FMEA])?	• Number of fault types identified • Number of fault types with detection mechanism and recovery strategies defined • % of anticipated faults that have explicit error handling
	What confidence do you have in detection, self-test, or voting?	• % of tests that produce correct/incorrect results at runtime
Recovery from faults	How do you restart failed nodes? Does the system support failover? Does the system support rollback to safe states?	• MTTR • System uptime and downtime • Failover time • Failover success rate
	Does the system support degraded modes?	• Time spent in degraded mode of operations • % of time spent in degraded modes

3.2 Reasoning About Robustness Properties

For a large, complex software-intensive system to be robust, its architecture will typically support a mix of capabilities to detect and recover from failures, and it will typically contain redundant (backup) resources that can be invoked in the case of a failure of a "live" resource. When evaluating an architecture with respect to robustness, we need to assess the design decisions in an architecture that lead to robustness. That is, we need to understand how well the architecture has been designed with sufficient resources to withstand failures of individual system elements. And we need to further understand how the architecture has been designed to monitor and manage these resources.

Thus, we need to gauge the degree to which the architecture supports the following strategies:

- management of system resources, which we categorize into two main approaches:[4]
  - *capacity sparing*: providing more resources than what is strictly necessary to accomplish the system's functions, where some of these resources act as spares to replace failed resources. There are three main approaches to achieve this:
    - *hardware redundancy* – We would like to understand the degree to which portions of the hardware architecture are protected by some sort of backup or redundant capability.
    - *software redundancy* – Similarly such redundant capabilities may be spare processes, threads, containers, virtual machines, and so forth that can be used in case of a failure of the active component. We would like to understand the degree to which and ease with which software components can be replaced.
    - *analytic redundancy* – A special case of software or hardware redundancy worth calling out is analytic redundancy [Bodson 1994, Sha 1998], where a complex component is mirrored by a simpler one that provides reduced, but more robust, functionality. For example, manual steering is analytically redundant to power steering in automobiles. We seek to understand which system functions have analytic spares to protect them in case of failure.
  - *capacity management:* what functions to allocate to which resources
    - growth potential: ease of adding capacity
    - ease of matching system resources to tasks (e.g., matching a task's resources to a suitable execution environment) and the ease with which a task can be moved around (dynamically or statically)
- management of system state, which has two approaches [Binder 2000]:
  - *state observability* – We would like to understand the degree to which it is possible to examine critical properties of the system's state such as memory and storage usage, processor utilization, liveness of processes and processors, communication channel utilization, latency, and transaction volume. These properties need to be observed and monitored because when such properties attain or exceed threshold or critical values, this is an indicator of a potential or actual fault. Thus, we need to determine what state properties are visible and what we can infer from these properties.

[4] We conceive of *approaches* as generalizations of tactics, which we describe in Section 5.1.

- *state controllability* – Coupled with state observability is state controllability. Given that we are able to observe the state of the system, for the system to be truly robust we must also be able to control that state, for example, by restarting bad processes, clearing memory, switching from a primary processor to a backup processor, choosing a level of service for a component, and choosing which components are active and which ones are passive or standby. Thus, we need to determine what state properties are controllable and what kinds of control we can we exert.

Different scenarios will of course emphasize these approaches to different degrees. For example, detecting that memory is nearly full and instituting garbage collection involves state observability (examining current memory usage) and state controllability (putting the process into the garbage collection mode). Consider another scenario involving detecting a database failure and replacing it with its hot spare (a "mirror"). This scenario relies on being able to observe that the primary database has failed and on being able to control system state—directing all database connections to its mirror (a redundant copy of the database).

Analyzing for robustness, then, is about examining the mix of mechanisms selected for a set of robustness scenarios and predicting the percentage of faults and failures that can be observed (detected), prevented, masked, and recovered from (potentially deploying spare capacity such as redundant resources) to achieve system uptime requirements.

Note that many of the measures of robustness in Table 1 cannot be estimated from architectural artifacts alone. Many of these measures can only be measured off of a built and deployed system. However, this does not mean that we can do no useful analysis of an architecture with respect to robustness. As we will show in our "Playbook" for architecture analysis in Section 7, even where a measure does not exist or cannot be reliably obtained, an architecture can still be examined for its *fitness for purpose* with respect to the questions and measures described in Table 1.

3.3 Operationalizing the Measurement of Robustness

Thus, when analyzing an architecture for robustness, we have some analysis tradeoffs to make. We can analyze with respect to a particular set of scenarios and obtain a reasonably precise understanding of the architecture's accommodation of those scenarios. But that understanding is necessarily narrow—limited to just those scenarios that we have considered. Alternatively, we can analyze with respect to metrics and get a broad understanding of the architecture's overall level of predicted robustness—as measured by the metrics—but this gives us no insight into the specific risks involved in responding to specific scenarios. Furthermore, scenario-based analyses and design-level measures (like the degree of replication of critical resources) can be used to gain insight into a design. This insight therefore can be achieved *before* committing to an implementation. But precisely because these are measuring artifacts—like design specifications—that are created earlier in a system's lifetime, they may not accurately reflect the eventual state of the system. This more accurate level of knowledge can only be achieved by measuring the system in operation.

For this reason, we recommend doing both: evaluating with respect to scenarios to get a deep understanding of some *anticipated* forms of robustness threats and, later in the lifecycle, adding analyses using measures from a system model or from the system in operation to obtain a more precise understanding of the qualities that the architecture (or any major subsystem within the architecture) helps to realize.

4 Robustness Scenarios

As stated in the book *Software Architecture in Practice*, quality attribute names themselves are of little use, as they are vague and subject to interpretation. The antidote to this vagueness is to specify quality attribute requirements as scenarios [Bass 2012]. A quality attribute scenario is simply a brief description of how a system is required to respond to some stimulus. Quality attribute scenarios, different from use cases, are architectural test cases. That is, they provide insights into the qualities that the architecture supports and any risks associated with the fulfillment of these scenarios.

A quality attribute scenario provides an operational definition of a quality of a system. The use of scenarios to specify quality attribute requirements for software dates back at least to 1994 [Kazman 1994]. Published examples include scenarios to specify requirements for seven of the most commonly occurring quality attributes [Bellomo 2015]: availability, interoperability, modifiability, performance, security, usability, and testability [Bass 2012]. More recently we have seen characterizations of the qualities of scalability and consistency [Klein 2015], integrability [Kazman 2020a], and maintainability [Kazman 2020b].

A quality attribute scenario has six parts [Bass 2012]. The two most important parts are a *stimulus* and a *response*. The stimulus is some event that arrives at the system, either during runtime execution (e.g., an invalid message arrives on a particular interface) or during development (e.g., a development iteration completes). The response defines how the system should behave when the stimulus occurs. For example, in response to an invalid message arriving, the system should log the event and send an error response message. In response to a development iteration completing, the unit and integration tests should be run and the test results reported.

The stimulus and response form the core of our operational definition by specifying the operation that we will measure. The third part of a scenario, the *response measure*, defines how we will measure the response and the satisfaction criteria. The response measure includes a metric and a threshold.

The other three parts of the scenario provide more details. We specify the *source* of the stimulus, to provide context for the scenario. We also specify the *environment*, which is the conditions under which the stimulus occurs and the response is measured. Finally, we specify the *artifact*, which is the portion of the system to which the requirement applies. Often, the artifact is the entire system, but in the example above, we might treat invalid messages on external interfaces differently from invalid messages on internal interfaces.

During requirements elicitation, we may specify the parts of a scenario in any order. We often begin with stimulus and response, although environment, source, or artifact may be the initial trigger for the requirement. In any case, once the scenario is specified, we usually arrange the parts to tell a story, as shown in Figure 1.

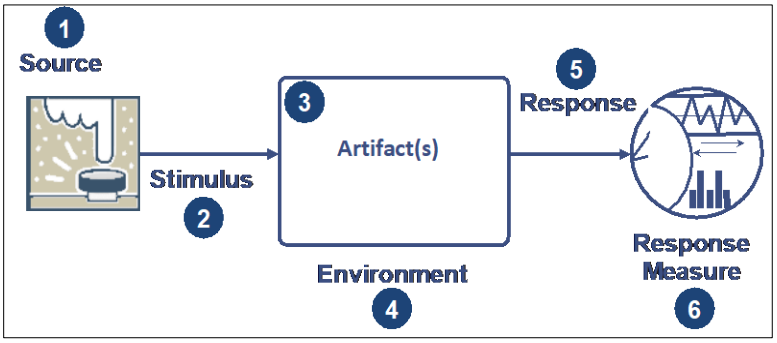


Figure 1: The Form of a General Scenario

In this way, the quality of the architecture, including measures that reflect on its robustness, can be continuously tracked and assessed. And if changes are made that undermine some architectural characteristic, the test case fails and appropriate remedial action can be taken.

Sidebar: Scenarios as Architectural Test Cases

In architecture analysis, scenarios are "architectural test cases." We use them to determine whether the architecture—as envisioned or as created—is consistent with its specification. Before the system is built, we use scenarios to assess the quality of the architectural decisions. Once the system exists, we can continue to use scenarios to assess the quality of the architecture as it evolves.

For runtime quality attributes, scenarios may become much more than simply guides for analysts. They can be used as acceptance tests and made part of the regression test suite. Or they can even be manifested as system health measures that are logged or monitored continuously at runtime. If the checks are at runtime, checking can be built into a system monitor; if the checks are run at build time, checking can be built into a continuous integration pipeline. In either case, checking requires appropriate visibility into system response measures (e.g., the ability to track latency, resource usage, and mean time to failure [MTTF]). For non-runtime quality attributes (assuming that source code is available), we can monitor the quality or degradation of the architecture's modular structure via architecture analysis tools, or we can monitor project management measures of the effort required to make changes.

4.1 General Scenario for Robustness

As we noted in the previous section, operational definitions are not exclusive: There is not a single scenario that specifies all the possible measurements that could characterize a quality like robustness. However, if we look at the definitions of robustness, we find some common themes. A *general scenario* maps those common themes into the parts of a quality attribute scenario, providing a template that we can use to create *concrete scenarios* for a particular system. The general scenario defines the *type* of the values for each part of the scenario, and a concrete scenario for robustness of a system is created by specifying one or more system-specific values of the selected type for each part of the scenario. (We say values—plural—because, for example, a scenario might have more than one response measure.)

Here is the general scenario for robustness:[5]

Scenario Part	Possible Type for Each Value
Source	Software, external system, or hardware
Stimulus	One of the following: • Software fault or failure • Hardware failure • Unanticipated message • Spike in demand • Invalid input • Diagnostic test fails • Unresponsive component • Responds after deadline • Defined threshold (e.g., processor utilization)
Artifact	One of the following: • Single software element • Multiple software elements • Hardware element • Entire software system
Environment	One of the following: • Normal operations • Degraded modes
Response	One or more of the following: • Fault detected and administrators notified • System operates in degraded mode • Fault detected, logged, and reported • Fault repaired • Fault prevented, logged, and reported • Diagnostics completed
Response Measure	One or more of the following: • MTTD • MTTR • Percentage uptime • Failover time • Uptime • % messages delivered • % of responses received

4.2 Example Scenarios for Robustness

Each of the following example scenarios is constructed by selecting one or more of the types of values from each of the six parts of the general scenario and specifying a system-specific value. For each example, we will use an easy-to-understand "typical" system. In practice, you would choose values that are as precise as possible, in the context of your system. In each example, notes

[5] This general scenario is adapted from Bass and colleague's general scenario for modifiability [Bass 2012, §7.1].

in square brackets are added to trace back to general scenario types in cases where the traceability is not obvious.

Sidebar: Architecting for the Unknown with Growth and Exploratory Scenarios

As we will discuss in Section 5, a great challenge of building robust systems is architecting for the unknown. Since we cannot enumerate all possible future changes or failure conditions, we cannot create architectural responses to all unknowns. And even if we could, the time to complete a project would grow enormously, and the cost/benefit ratio of architecting for less and less likely failures and changes would shrink to vanishingly small. So what is the prudent architect or analyst to do?

We advocate employing growth and exploratory scenarios as a means of exploring the space of conditions that we want to explicitly consider in our architecture design and analysis process. Growth scenarios represent anticipated growth and anticipated stresses on a system. For example, if we know based on our history that we are likely to add sensors to the system on a regular basis, we can create a growth scenario that captures this anticipated future change in requirements.

Exploratory scenarios are ideally used to ask "what if" questions, to help probe our understanding of more extreme potential changes and more extreme environmental conditions. What would happen if all our processors failed simultaneously? How much work would be required to change the architecture to manage tighter coordination among system instances? What would happen if our backup network failed shortly after our main network failed? What would happen if we needed to report updates every second instead of reporting them hourly? What would happen under conditions of extreme heat, extreme load, or extreme growth in user requests? These kinds of scenarios, while perhaps unlikely, help us understand the limits of our architecture and the tradeoffs that have been made. They can also be used with respect to any quality attribute, such as in evaluating the robustness of an architecture to future security or maintainability requirements. As such they are a crucial tool in the toolbox of the designer and the analyst.

Collectively, growth and exploratory scenarios allow architects or analysts to explore the robustness of an architecture with respect to future changes or failure conditions that fall outside of current requirements but are perhaps more likely than not to become future requirements. Growth and exploratory scenarios aren't a panacea for the unknown though, as they still require someone to think of potential changes or failure conditions prior to making architectural decisions or evaluating an architecture.

4.2.1 Scenario 1: System Initialization Times Out

This example scenario describes how failure can be detected and recovered from, for system initialization of a networked avionics system such as a navigation system.

Scenario Part	Value
Source	Navigation System initialization configuration file errors
Stimulus	Navigation System initialization times out.
Artifact	Navigation System initialization component
Environment	Normal operations
Response	The timeout is detected, and the system is initialized using a standard configuration.
Response Measure	Timeout is detected 100% percent of the time. System initialization is restarted within 10 ms.

Note that in this scenario—as in many scenarios—multiple response measures are specified. Furthermore, in some cases, multiple scenarios are needed to completely specify the quality attribute requirement. For instance, the environment in the example scenario above was normal operations, and one of the responses was to reinitialize using a standard configuration. Another scenario, with identical stimulus and response measures, might specify an environment of the system being restarted in a degraded mode, and the response measure for restarting the system initialization could be changed to 20 ms.

4.2.2 Scenario 2: Software Fault and Recovery

This example scenario describes the robustness of the Flight Management System to recover from the failure of a software element.

Scenario Part	Value
Source	RADAR Altimeter Manager
Stimulus	RADAR Altimeter Manager does not report sensor data by deadline.
Artifact	Flight Management System
Environment	Normal operation
Response	Restart RADAR Altimeter Manager. Detect fault. Log error. Failover to backup.
Response Measure	Detect fault within 2 ms. Restart within 10 ms. Switch to backup in 5 ms. Altimeter data is available 100% of operation.

4.2.3 Scenario 3: Resource Threshold Is Approached

This example scenario describes how the system responds when a critical system resource is near its capacity.

Scenario Part	Value
Source	Processor Monitor
Stimulus	Total CPU utilization is at 85%.
Artifact	Flight Management System
Environment	Normal operation
Response	Disable noncritical functions or message throttling. Detect fault. Log error.
Response Measure	Detect fault within 2 ms. Restart within 10 ms. All critical deadlines are met.

4.2.4 Scenario 4: Hardware Failure and Restart

This example scenario describes how the system responds when a critical system resource fails.

Scenario Part	Value
Source	Processor Monitor
Stimulus	CPU overheats and shuts down.
Artifact	Flight Management System
Environment	Normal operation
Response	Failover to other CPUs (hot spare). Detect fault. Log error.
Response Measure	Detect fault within 2 ms. Restart within 10 ms. All critical deadlines are met.

5 Mechanisms for Achieving Robustness

We have thus far focused most of our attention on analyzing an architecture for robustness. But analysis and design are two sides of the same coin. Now we turn our attention to the architectural design task of *achieving* robustness.

An architect must choose a set of design concepts to construct a solution for any quality attribute requirement [Cervantes 2016], and the architecture that the analyst is given to examine will contain design decisions regarding such concepts. Here we generically refer to these design concepts as "mechanisms." We will discuss and provide examples of two important kinds of architectural design mechanisms: *tactics* and *patterns*. These mechanisms are the architect's main tools to achieve a desired set of robustness characteristics.

5.1 Tactics

A mechanism is an architectural approach that we can take to control a quality attribute. Many discussions of mechanisms—for example, Bass and colleagues [Bass 2012]—focus on *technical mechanisms*, such as architectural patterns and tactics. Technical mechanisms are sufficient to satisfy requirements for quality attributes such as availability or consistency in a big data system. For other quality attributes such as security and robustness, technical mechanisms are necessary but not sufficient to satisfy some system-level requirements, and the technical mechanisms must be accompanied by *governance mechanisms*. For example, security defense-in-depth might begin with physical security, which requires governance to enforce access procedures. For robustness, any modification to the software will be extremely difficult without governance such as acquisition practices that ensure that appropriate architecture, design, and code documentation are produced, that code reviews are performed, that test suites are maintained, and that employees are appropriately trained so that they do not undermine the integrity of the architecture with the changes they implement.

Governance mechanisms related to robustness in the Department of Defense (DoD) context appear in discussions of the Modular Open System Approach (MOSA) [ODASD 2017] and DoD software acquisition practices [DIB 2019]. The rest of this section focuses on technical mechanisms for robustness: Architecture approaches are commonly employed to satisfy the types of scenarios that we outlined in the previous section.

In practice, the terminology used for technical mechanisms is informal, and often the term is used to refer to any decision made during the architecture design process or to any fragment of the architecture that is intended to address some particular functional or quality attribute-related concern. In this report, we will consider two specific types of mechanisms:

- Architectural Patterns. *Design patterns* are conceptual solutions to recurring design problems that exist in a defined context. A pattern is *architectural* when its use directly and substantially influences the satisfaction of an architecture driver such as a quality attribute scenario [Cervantes 2016]. An architectural pattern defines a set of element types and interactions; the topological layout of the elements; and constraints on topology, element behavior, and interactions [Bass 2012].

- Architectural Tactics. *Tactics* are smaller building blocks of design than architectural patterns, focused on a single element or interaction, in contrast to a pattern that defines a collection of elements [Bass 2012].

Since tactics are simpler and more fundamental than patterns, we begin our discussion of mechanisms for robustness with them. Tactics are the building blocks of design, the raw materials from which patterns, frameworks, and styles are constructed. Each set of tactics is grouped according to the quality attribute goal that it addresses. The goals for the robustness tactics shown in Figure 2 are to enable a system, in the face of a fault, to prevent, mask, or repair the fault so that a service being delivered by the system remains compliant with its specification. The tactic descriptions presented below are derived, in part, from the third edition of *Software Architecture in Practice* [Bass 2012]. We discuss each of the tactics presented in Figure 2 in more detail below. For each tactic that we discuss, we not only describe the tactic but also relate it to the measures described in Section 3.1 as a way of describing the intent and impact of the tactic.

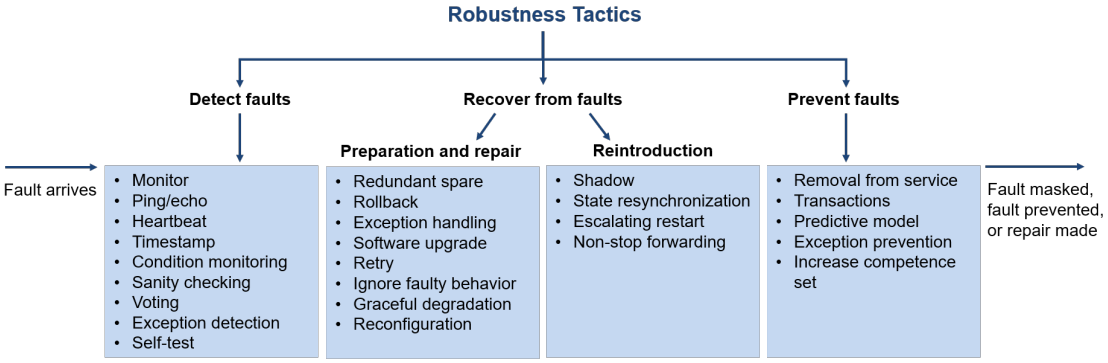


Figure 2: Robustness Tactics

These tactics are known to influence the responses (and hence the costs) in the general scenario for robustness (e.g., number of components affected, effort, calendar time, new defects introduced). Table 2 summarizes the tactics presented in this section, and how each relates to the characteristics and measures presented in Sections 3.1 and Figure 2. The table assesses the relationships between the availability (robustness) tactics and the architectural approaches of capacity sparing, capacity management, state observability, and state controllability, each of which can contribute to achieving higher measures of MTTF and MTTR. An architect, in designing for high availability, needs to make decisions to

- provision spare capacity (including, in most cases, providing for backup resources)
- manage the capacity of the resources that are available
- observe the state of the system to determine when the system, or some part of it, is inconsistent with respect to its specification
- control the state of the system to keep the system alive and healthy, consistent with its specification

By consciously managing these system strategies and concerns, the architect can design to reduce the likelihood of a failure, thus increasing the MTTF measure, or to recover from failures more quickly, thus reducing the MTTR measure.

Table 2: Robustness Tactics and Their Relationships to Architectural Approaches and Measures of Interest

Tactic	Architectural Approaches				Measures	
	Capacity Sparing	Capacity Mgmt.	State Observ.	State Control.	MTTF	MTTR
Monitor			+			+
Ping/Echo			+			+
Heartbeat			+			+
Timestamp			+			+
Condition Monitoring			+			+
Sanity Checking			+			+
Voting			+			+
Exception Detection			+			+
Self-test			+			+
Active Redundancy	+				+	+
Passive Redundancy	+				+	+
Spare	+					+
Rollback				+		+
Exception Handling				+		+
Software Upgrade				+		+
Retry				+	+	+
Ignore Faulty Behavior				+	*	+
Graceful Degradation		+		+	*	*
Reconfiguration	+	+		+	*	*
Shadow				+		+
State Resynchronization				+		+
Escalating Restart				+		+
Non-stop Forwarding		+		+	*	*
Removal from Service	+			+	+	
Transactions				+	+	+
Predictive Model			+		+	

Tactic	Architectural Approaches				Measures	
	Capacity Sparing	Capacity Mgmt.	State Observ.	State Control.	MTTF	MTTR
Exception Prevention				+	+	
Increase Competence Set				+	+	

Note: A plus sign indicates that the tactic positively addresses maintainability properties and hence measures, and an asterisk indicates that the tactic might positively or negatively address the measure, depending on its realization. A blank cell means that the property has no consistent effect on the measure.

Sidebar: Designing for Unknown Unknowns

Many of the robustness tactics are employed because we acknowledge that faults in *parts* of the system are a normal occurrence; hence we can architect to detect and recover from those faults, increasing the likelihood that faults do not become failures. But these tactics are primarily aimed at recovering from known, anticipated faults. Several of the robustness tactics presented below are, however, particularly helpful in considering how to deal with problems that we cannot anticipate—the so-called unknown unknowns. These tactics include *rollback, ignore faulty behavior, abort, analytic redundancy, masking, and return to safe state.*

Every architect must consider and balance cost, schedule, and risk when making design decisions. As such it is impossible to deal with *all* unknown unknowns. There is always an envelope of faults and error states that a prudent architect chooses to handle. The unknown unknowns are then the complement to these identified faults and states. One of the motivations for enumerating tactics is to help architects reflect on and, ideally, broaden that "known" envelope.

As an analyst, when you see a tactic such as *exception handling* being employed, you should ask which exceptional states that tactic handles and additionally ask what happens to exceptional states not captured. Models, such as fault tree analysis, can help analysts understand the scope of the tactics that architects are considering.

5.1.1 Detect Faults

Before any system can take action regarding a fault, the presence of the fault must be detected or anticipated. Tactics in this category include the following:

- *Monitor*. A monitor is a component that is used to monitor the state of health of various other parts of the system: processors, processes, input/output, memory, and so forth. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack. It orchestrates software using other tactics in this category to detect malfunctioning components. For example, the system monitor can initiate *self-tests* or be the component that detects faulty *timestamps* or missed *heartbeats*.[6]

[6] When the detection mechanism is implemented using a counter or timer that is periodically reset, this specialization of *system monitor* is referred to as a *watchdog*. During nominal operation, the process being monitored will periodically reset the watchdog counter/timer as part of its signal that it's working correctly; this is sometimes referred to as "petting the watchdog."

- *Ping/echo.* Ping/echo refers to an asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path. But the echo also determines that the pinged component is alive and responding correctly. The ping is often sent by a system monitor. Ping/echo requires a time threshold to be set; this threshold tells the pinging component how long to wait for the echo before considering the pinged to have failed ("timed out"). Standard implementations of *ping/echo* are available for nodes interconnected via the Internet Protocol (IP).
- *Heartbeat.* A heartbeat is a fault detection mechanism that employs a periodic message exchange between a system monitor and a process being monitored. A special case of heartbeat is when the process being monitored periodically resets the watchdog timer in its monitor to prevent it from expiring and thus signaling a fault. For systems where scalability is a concern, transport and processing overhead can be reduced by piggybacking heartbeat messages on to other control messages being exchanged between the process being monitored and the distributed system controller. The big difference between heartbeat and ping/echo is what holds the responsibility for initiating the health check—the monitor or the component itself.
- *Timestamp.* This tactic is used to detect incorrect sequences of events, primarily in distributed message-passing systems. A timestamp of an event can be established by assigning the state of a local clock to the event immediately after the event occurs. Simple sequence numbers can also be used for this purpose, if time information is not important.
- *Condition monitoring*. This tactic involves checking conditions in a process or device or validating assumptions made during the design. By monitoring conditions, this tactic prevents a system from producing faulty behavior. The computation of checksums is a common example of this tactic. However, the monitor must itself be simple (and, ideally, provable) to ensure that it does not introduce new software errors.
- *Sanity checking*. This tactic checks the validity or reasonableness of specific operations or outputs of a computation. This tactic is typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny. It is most often employed at interfaces to examine a specific information flow.
- *Voting.* The most common realization of this tactic is referred to as Triple Modular Redundancy (or TMR, as we will discuss in Section 5.2.2), which employs three components that do the same thing, each of which receives identical inputs and forwards its output to voting logic, used to detect any inconsistency among the three output states. Faced with an inconsistency, the voter reports a fault. It must also decide what output to use. It can let the majority rule or choose some computed average of the disparate outputs. This tactic depends critically on the voting logic, which is usually realized as a simple, rigorously reviewed, and tested singleton so that the probability of error is low.
  - *Replication* is the simplest form of *voting*; here, the components are exact clones of each other. Having multiple copies of identical components can be effective in protecting against random failures of hardware, but this approach cannot protect against design or implementation errors, in hardware or software, since there is no form of diversity embedded in this tactic.

- *Functional redundancy* is a form of voting intended to address the issue of common-mode failures (design or implementation faults) in hardware or software components. Here, the components must always give the same output given the same input, but they are diversely designed and diversely implemented.
- *Analytic redundancy* permits not only diversity among components' private sides but also diversity among the components' inputs and outputs. This tactic is intended to tolerate specification errors by using separate requirement specifications. In embedded systems, analytic redundancy also helps when some input sources are likely to be unavailable at times. For example, avionics programs have multiple ways to compute aircraft altitude, such as using barometric pressure, the radar altimeter, and the geometric straight-line distance and look-down angle of a point ahead on the ground. The voter mechanism used with analytic redundancy needs to be more sophisticated than just letting majority rule or computing a simple average. It may have to understand which sensors are currently reliable or not, and it may be asked to produce a higher fidelity value than any individual element can, by blending and smoothing individual values over time.

- *Exception detection.* This tactic is used for detecting a system condition that alters the normal flow of execution. The *exception detection* tactic can be further refined:
  - *System exceptions* will vary according to the processor hardware architecture employed and include faults such as divide by zero, bus and address faults, illegal program instructions, and so forth.
  - The *parameter fence* tactic incorporates an a priori data pattern (such as 0xDEADBEEF) placed immediately after any variable-length parameters of an object. This allows for runtime detection of overwriting the memory allocated for the object's variable-length parameters.
  - *Parameter typing* employs a base class that defines functions that add, find, and iterate over message parameters in Type-Length-Value (TLV) format. Derived classes use the base class functions to implement functions that provide parameter typing according to each parameter's structure. Use of strong typing to build and parse messages results in higher availability than implementations that simply treat messages as byte buckets. Of course, all design involves tradeoffs. When you employ strong typing, you typically trade higher availability against ease of evolution.
  - *Timeout* is a tactic that raises an exception when an element detects that it or another element has failed to meet its timing constraints. For example, an element awaiting a response from another element can raise an exception if the wait time exceeds a certain value.
- *Self-test.* Elements (often entire subsystems) can run procedures to test themselves for correct operation. Self-test procedures can be initiated by the element itself or invoked from time to time by a system monitor. These may involve employing some of the techniques found in condition monitoring such as checksums.

5.1.2 Recover from Faults

Recover from faults tactics are refined into *preparation and repair* tactics and *reintroduction* tactics. The latter are concerned with reintroducing a failed (but rehabilitated) element back into normal operation.

Preparation and repair tactics are based on a variety of combinations of retrying a computation or introducing redundancy. They include the following:

- *Redundant spare*. This tactic has three major manifestations:
  - *Active redundancy (hot spare).* In this configuration, all the nodes (active or redundant spare) in a protection group[7] receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s). Because the redundant spare possesses an identical state to the active processor, it can take over from a failed element in a matter of milliseconds. The simple case of one active node and one redundant spare node is commonly referred to as 1+1 ("one plus one") redundancy. *Active redundancy* can also be used for facilities protection, where active and standby network links are used to ensure highly available network connectivity.
  - *Passive redundancy (warm spare).* In this configuration, only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates. Because the state maintained by the redundant spares is only loosely coupled with that of the active node(s) in the protection group (with the looseness of the coupling being a function of the checkpointing mechanism employed between active and redundant nodes), the redundant nodes are referred to as warm spares. Depending on a system's availability requirements, *passive redundancy* provides a solution that achieves a balance between the more highly available but more compute-intensive (and expensive) *active redundancy* tactic and the less available but significantly less complex *cold spare* tactic (which is also significantly cheaper).
  - *Spare (cold spare).* Cold sparing refers to a configuration where the redundant spares of a protection group remain out of service until a failover occurs, at which point a power-on-reset procedure is initiated on the redundant spare before it is placed in service. Due to its poor recovery performance, cold sparing is better suited for systems having only high-reliability (MTBF) requirements as opposed to those also having high-availability requirements.
- *Rollback.* This tactic permits the system to revert to a previous known good state, referred to as the "rollback line"—rolling back time—upon the detection of a failure. Once the good state is reached, then execution can continue. This tactic is often combined with active or passive redundancy tactics so that after a rollback has occurred a standby version of the failed element is promoted to active status. Rollback depends on a copy of a previous good state (a checkpoint) being available to the elements that are rolling back. Checkpoints can be stored in a fixed location and updated at regular intervals or at convenient or significant times in the processing, such as at the completion of a complex operation.
- *Exception handling.* Once an exception has been detected, the system must handle it in some fashion. The easiest thing it can do is simply to crash, but of course that's a terrible idea from the point of availability, usability, testability, and plain good sense. There are much more productive possibilities. The mechanism employed for *exception handling* depends largely on the programming environment employed, ranging from simple function return codes (error codes) to the use of exception classes that contain information helpful in fault correlation,

[7] A protection group is a group of processing nodes where one or more nodes are "active" and the remaining nodes in the protection group serve as redundant spares.

such as the name of the exception thrown, the origin of the exception, and the cause of the exception. Software can then use this information to mask the fault, usually by correcting the cause of the exception and retrying the operation.

- *Software upgrade.* The goal of this tactic is to achieve in-service upgrades to executable code images without affecting services. These are commonly used in routers, telecommunications switches, and similar contexts where reboots and downtime are not practical. The actual upgrade may be realized as a function patch, class patch, or hitless in-service software upgrade (ISSU). A function patch is used in procedural programming and employs an incremental linker/loader to store an updated software function into a pre-allocated segment of target memory. The new version of the software function will employ the entry and exit points of the deprecated function. Also, upon loading the new software function, the symbol table must be updated and the instruction cache invalidated. The class patch realization of this tactic is applicable for targets executing object-oriented code, where the class definitions include a backdoor mechanism that enables the runtime addition of member data and functions. Hitless ISSU leverages the *active redundancy* or *passive redundancy* tactics to achieve non-service-affecting upgrades to software and associated schema. In practice, the function patch and class patch are used to deliver bug fixes while the hitless ISSU is used to deliver new features and capabilities.
- *Retry.* The retry tactic assumes that the fault that caused a failure is transient and retrying the operation may lead to success. This tactic is used in networks and server farms where failures are expected and common. There should be a limit on the number of retries that are attempted before a permanent failure is declared.
- *Ignore faulty behavior.* This tactic calls for ignoring messages sent from a particular source when the system determines that those messages are spurious. For example, we could instruct it to ignore messages from an external element launching a denial-of-service attack, such as by establishing filters for an access-control list.
- *Graceful degradation.* This tactic maintains the most critical system functions in the presence of element failures, dropping less critical functions. This is done to ensure that failures of individual system elements gracefully reduce system functionality but do not cause a complete system failure.
- *Reconfiguration.* Using this tactic, a system attempts to recover from failures of a system element by reassigning responsibilities to the resources left functioning, while maintaining as much of the critical functionality as possible.

Reintroduction is where a failed element is reintroduced after a repair has been effected. *Reintroduction* tactics include the following:

- *Shadow.* This tactic refers to operating a previously failed or in-service upgraded element in a "shadow mode" for a predefined duration of time prior to reverting the element back to an active role. During this duration its behavior can be monitored for correctness, and it can repopulate its state incrementally.
- *State resynchronization.* This tactic is a reintroduction partner to the *active redundancy* and *passive redundancy* preparation and repair tactics. When used alongside the *active redundancy* tactic, the *state resynchronization* occurs organically, since the active and standby elements each receive and process identical inputs in parallel. In practice, the states of the active and standby elements are periodically compared to ensure synchronization. This comparison

may be based on a cyclic redundancy check calculation (checksum) or, for systems providing safety-critical services, a message digest calculation (a one-way hash function). When used alongside the *passive redundancy* (warm spare) tactic, *state resynchronization* is based solely on periodic state information transmitted from the active element(s) to the standby element(s), typically via checkpointing. A special case of this tactic is found in stateless services, whereby any resource can handle a request from another (failed) resource.

- *Escalating restart.* This reintroduction tactic allows the system to recover from faults by varying the granularity of the element(s) restarted and minimizing the level of service affectation. For example, consider a system that supports four levels of restart, as follows. The lowest level of restart (call it Level 0), and hence least impacting on services, employs *passive redundancy* (warm spare), where all child threads of the faulty element are killed and recreated. In this way, only data associated with the child threads is freed and reinitialized. The next level of restart (Level 1) frees and reinitializes all unprotected memory, while protected memory remains untouched. The next level of restart (Level 2) frees and reinitializes all memory, both protected and unprotected, forcing all applications to reload and reinitialize. And the final level of restart (Level 3) would involve completely reloading and reinitializing the executable image and associated data segments. Support for the *escalating restart* tactic is particularly useful for the concept of graceful degradation, where a system is able to degrade the services it provides while maintaining support for mission-critical or safety-critical applications.
- *Non-stop forwarding.* The concept of non-stop forwarding originated in router design. In this design, functionality is split into two parts: supervisory, or control plane (which manages connectivity and routing information), and data plane (which does the actual work of routing packets from sender to receiver). If a router experiences the failure of an active supervisor, it can continue forwarding packets along known routes—with neighboring routers—while the routing protocol information is recovered and validated. When the control plane is restarted, it implements what is sometimes called "graceful restart," incrementally rebuilding its routing protocol database even as the data plan continues to operate.

5.1.3 Prevent Faults

Instead of detecting faults and then trying to recover from them, what if your system could prevent them from occurring in the first place? Although this sounds like some measure of clairvoyance might be required, it turns out that in many cases it is possible to do just that.[8]

- *Removal from service.* This tactic refers to temporarily placing a system element in an out-of-service state for the purpose of mitigating potential system failures. One example involves taking an element of a system out of service and resetting the element in order to scrub latent faults (such as memory leaks, fragmentation, or soft errors in an unprotected cache) before the accumulation of faults becomes service affecting (resulting in system failure). Another term for this is *software rejuvenation*.

---

[8] These tactics deal with runtime means to prevent faults from occurring. Of course, an excellent way to prevent faults—at least in the system you're building, if not in systems that your system must interact with—is to produce high-quality code. This can be done by means of code inspections, pair programming, solid requirements reviews, and a host of other good engineering practices.

- *Substitution*. This tactic employs safer protection mechanisms—often hardware-based—for software design features that are considered critical. For example, hardware protection devices such as watchdogs, monitors, and interlocks are often used in lieu of software versions, as these are typically more reliable. Note that substitution is typically only feasible and beneficial when the function being replaced is relatively simple.
- *Transactions*. Systems targeting high-availability services leverage transactional semantics to ensure that asynchronous messages exchanged between distributed elements are atomic, consistent, isolated, and durable. These four properties are referred to as the "ACID properties." The most common realization of the *transactions* tactic is "two-phase commit" protocol. This tactic prevents race conditions caused by two processes attempting to update the same data item.
- *Predictive model*. A predictive model, when combined with a *monitor,* is employed to monitor the state of health of a system process to ensure that the system is operating within its nominal operating parameters and to take corrective action when conditions are detected that are predictive of likely future faults. The operational performance metrics monitored are used to predict the onset of faults; examples include session establishment rate (in an HTTP server), threshold crossing (monitoring high- and low-water marks for some constrained, shared resource), maintaining statistics for process state (in-service, out-of-service, under maintenance, idle), and message queue length statistics.
- *Exception prevention.* This tactic refers to techniques employed for the purpose of preventing system exceptions from occurring. The use of exception classes, which allows a system to transparently recover from system exceptions, was discussed above. Other examples of *exception prevention* include abstract data types such as smart pointers and the use of wrappers to prevent faults such as dangling pointers and semaphore access violations from occurring. Smart pointers prevent exceptions by doing bounds checking on pointers and by ensuring that resources are automatically deallocated when no data refers to them. In this way resource leaks are avoided.
- *Increase competence set.* A program's competence set is the set of states in which it is "competent" to operate. For example, the state when the denominator is zero is outside the competence set of most divide programs. When an element raises an exception, it is signaling that it has discovered itself to be outside its competence set; in essence, it doesn't know what to do and quits in defeat. Increasing an element's competence set means designing it to handle more cases—faults—as part of its normal operation. For example, an element that assumes it has access to a shared resource might throw an exception if it discovers that access is blocked. Another element might simply wait for access or return immediately with an indication that it will complete its operation on its own the next time it does have access. In this example, the second element has a larger competence set than the first.
- *Abort*. If an operation is determined to be unsafe, it is aborted before it can cause damage. This tactic is a common strategy employed to ensure that a system fails safely.
- *Masking*. A system may mask a fault by comparing the results of several redundant upstream components and employing a voting procedure in case one or more of the values output by these upstream components differ. In such a case the majority wins, and any erroneous values are never seen by downstream components. For this tactic to work, the voter must be simple and highly reliable (perhaps employing the substitution tactic).

5.2 Patterns

As stated above, architectural tactics are the fundamental building blocks of design. Hence, they are the building blocks of architectural patterns. By way of analogy, we say that tactics are atoms and patterns are molecules. During analysis it is often useful for analysts to break down complex patterns into their component tactics so that they can better understand the specific set of quality attribute concerns that patterns address, and how. This approach simplifies and regularizes analysis, and it also provides more confidence in the completeness of the analysis.

Next, we provide a brief description of a set of patterns, a discussion of how the patterns promote robustness, and the other quality attributes that are negatively impacted by these patterns (tradeoffs). Note that just because a pattern negatively impacts some other quality attribute, this does not mean that the levels of that quality attribute will be unacceptable. For example, the use of the *transactions* tactic always negatively affects performance (specifically latency) as transactions add some overhead. This is inevitable; the inclusion of the transaction protocol adds processing and communication steps. This is not to say, however, that the resulting latency of the system will be unacceptable. Perhaps the added latency is only a small fraction of end-to-end latency on the most important use cases. In such cases the tradeoff is a good one, providing benefits for robustness while "costing" only a small amount of latency.

It is also important to note that the tradeoffs described below are general. Other architectural mechanisms or decisions applied with the pattern may change the impacts. For example, if one chose the *active redundancy (hot spare)* tactic, one could employ Mesos, in "high availability" mode, employing Apache Zookeeper. Zookeeper provides an infrastructure for synchronization of nodes in a "quorum," where the nodes act as hot spares for each other, typically with one being the "master" and the remainder of the nodes being "backups." Updates get sent to the master, which automatically informs the backups. The use of such a tool might greatly ease the implementation burden of this tactic. These are the kinds of assessments that analysts need to make when assessing the appropriateness of the tactics and patterns selected and implemented.

This pattern list is not meant to be exhaustive. The purpose of this section is to illustrate the most common robustness patterns—Process Pairs, Triple Modular Redundancy, N+1 Redundancy, Circuit Breaker, Recovery Blocks, Forward Error Recovery, Health Monitoring, and Throttling—and to show how analysts can break patterns down into tactics that allow them to understand the patterns' quality attribute characteristics, strengths, weaknesses, and tradeoffs.

5.2.1 Process Pairs

The Process Pairs pattern combines software (and sometimes hardware) redundancy tactics with transactions and checkpointing. Two identical processes are running, with one process being designated the "primary" or "leader." This primary process is the one that clients interact with at runtime, under normal circumstances. As the primary process processes information, it bundles its execution into transactions. These transactions typically result in a change to data in the primary process. When a transaction has successfully completed, a "checkpoint" is sent to the backup process so that the backup can align the state of its data with that of the primary. In this way, if the primary process fails, the backup always has a consistent state that is as complete as the state of the failed primary (except for any transaction that failed in mid-execution on the primary).

Benefits for robustness:

- The benefit of Process Pairs, over simply using a transaction mechanism, is that upon failure of the primary process the recovery is very fast (as compared with restarting the primary process and playing back the transaction log to recreate the state just prior to the failure).

Tradeoffs:

- The Process Pairs pattern requires the expenditure of additional software, networking, and potentially hardware resources.
- Adding the checkpointing and failover mechanisms increases up-front complexity.

5.2.2 Triple Modular Redundancy

The Triple Modular Redundancy (TMR) pattern is one of the earliest known robustness patterns. Its roots can be traced back to at least 1951 in computer hardware, where TMR was used in magnetic drum memory to ameliorate the inherent unreliability of individual elements. It builds upon the *active redundancy* tactic, where two or more elements process the same inputs in parallel. Many variants of this pattern exist, such as quad-modular redundancy (QMR) and N-modular redundancy. In each case one node may be elected as "active" with the other nodes processing all inputs in parallel, but only being activated in case the active node fails. In other versions there is a voting process where the voter collects and compares the "votes" from each of the replicated nodes; if a node disagrees with the majority, it is marked as failed and its outputs are ignored.

Benefits for robustness:

- The most obvious benefit of TMR is the avoidance of a single point of failure.
- If a voter is used, then this pattern also includes a fault detection mechanism.

Tradeoffs:

- Redundancy greatly increases the hardware costs for the system, its complexity, and its initial development time. Also, systems using this pattern consume substantially more resources at runtime (e.g., energy and network bandwidth). Finally, there is the added complexity of determining which of the nodes to anoint as the "active" node and, in case of failure, which backup to promote to active status.
- Some systems do not use exact replicas, but rather use N-version programming [Avizienis 1985] to produce functionally equivalent nodes. N-version programming can result in a more robust system, since the different versions are less likely to suffer from a common mode failure than pure replicas, but it greatly increases the cost and complexity of the system's software.

5.2.3 N+1 Redundancy

The N+1 Redundancy pattern builds upon one or more redundancy tactics. In this pattern there are N active nodes, with one spare node. The assumptions are that the active nodes have similar functionality and the spare node can be introduced to replace any of the N active nodes if one of them has failed. The one spare node may be an active spare, meaning that it processes all the same inputs as the system(s) that it is mirroring; it may be a passive spare, meaning that the active nodes periodically send it updates; or it may be a cold spare, meaning that when it takes the place of a failed node it initially has none of that node's state.

Benefits for robustness:

- Clearly N+1 Redundancy provides the benefit of any redundancy pattern, which is the avoidance of a single point of failure.
- Just as clearly, N+1 Redundancy is much less expensive than TMR, QMR, or similar patterns that require a heavy investment in software and hardware, since a single backup node can back up any chosen number of active nodes.

Tradeoffs:

- The higher the N, the greater the likelihood that more than one failure could occur.
- The lower the N, the more an implementation of this pattern costs, in terms of redundant hardware and the attendant energy costs.

5.2.4 Circuit Breaker

The Circuit Breaker pattern is used to detect failures and prevent the failure from constantly reoccurring or cascading to other parts of a system. It is commonly used in cases where failures are intermittent. A Circuit Breaker is a combination of a timeout (an exception detection tactic) and a monitor, which is an intermediary between services [Kazman 2020b]. With a Circuit Breaker [Nygard 2017], a service is wrapped and the wrapper monitors the state of the service. If it is determined that the service is not operating consistently with its specification, the breaker is tripped, and all subsequent calls to the service return an error immediately. This ensures that such dependencies do not slow down other parts of the system due to, for example, repeated timeouts, which increases the controllability of the deployment.

Benefits for robustness:

- The use of this pattern limits the consequences of a failure by wrapping the interface to that element and returning immediately if a failure has been detected. This can greatly reduce the amount of resources wasted on retrying a service that is known to have failed.

Tradeoffs:

- The use of a Circuit Breaker will negatively affect performance. Like many robustness patterns, this tradeoff is often considered to be justifiable, particularly if services experience intermittent and transient failures.

5.2.5 Recovery Blocks

The Recovery Blocks pattern is used when there are several possible ways to process a result based on an input and one is chosen as the primary processing capability. After the primary processing capability returns a result, it is passed through an acceptance test. If this test fails, this pattern then tries passing the input to a second processing capability. This second processing capability acts as a "recovery block" for the primary. This process can continue for any number of backup processing capabilities. This is a kind of N-version programming, or it may be realized as a form of analytic redundancy.

Another variant of this pattern occurs when all the components process the input in parallel—an instance of active redundancy of software components—and some selection logic looks at the results of each component's acceptance tests in order, starting from the primary. The major

difference between these two variants is that the first variant invokes the processing components serially, and the second one invokes them in parallel.

Benefits for robustness:

- This pattern is useful in cases where the processing is complex, where high availability is desired, but where hardware redundancy is not a viable option. This pattern does not protect against hardware failures, of course, but it does provide some protection against software failures and bugs.

Tradeoffs:

- If the serial variant of this pattern is employed, latency (from the time the input arrives to the time that an acceptable result is produced) will be increased in cases where one or more acceptance tests fail.
- If the parallel variant of this pattern is employed, substantially more CPU resources will be consumed to process each input.

5.2.6 Forward Error Recovery

The Forward Error Recovery pattern is a kind of active redundancy employed in situations where relatively high levels of faults are expected. The idea of Forward Error Recovery originated in the telecommunications domain, where communication over noisy channels resulted in large numbers of packets being damaged, resulting in large numbers of packet retries. This was expensive, particularly in the early days of telecommunications or in cases where latency was very large (for example, communication with space probes). To attempt to address this shortcoming, packets were encoded with redundant information so that they could self-detect and self-correct a limited number of errors. This approach to detecting and correcting errors has been manifested in many other domains, such as in RAID (Redundant Arrays of Inexpensive Disks). In this case, with large numbers of disks, failures are expected to occur on a regular basis, so redundant information is stored on the disk array (hence the R in RAID), enabling the array to function at a high level of availability, masking errors on individual disks.

Benefits for robustness:

- This pattern is useful in cases where the underlying hardware or software is unreliable and where it is possible to encode redundant information.

Tradeoffs:

- As with most patterns for robustness, higher levels of availability can be costly. For example, the higher the level of redundancy in a disk array, the more expensive it is (on a per-kilobyte stored basis), and the more redundant information included in a packet, the lower the effective bandwidth of the network.
- If the number of errors and combinations of errors are expected to be high, then creating such Forward Error Recovery schemes can be complex and costly and may only cover a small set of the erroneous states.

5.2.7 Health Monitoring

In complex networked environments, just determining the health of a remote service may be challenging. To achieve high levels of availability, it is necessary to be able to tell, with confidence, whether a service is operating consistently with its specifications. The Health Monitoring pattern (sometimes called "Endpoint Health Monitoring") addresses this need. The monitor is a separate service that periodically sends a message to every endpoint that needs to be monitored. The simplest form of this pattern is ping/echo, where the monitor sends a ping message, which is echoed by the endpoint. But more sophisticated checks are common—instances of the *monitor* tactic—such as measuring the round-trip latency for send/response messages and checking on various properties of the monitored endpoints such as CPU utilization, memory utilization, application-specific measures, and so forth.

Benefits for robustness:

- This pattern is useful in cases where the system is distributed and where the health of the distributed components cannot be assessed locally in a timely fashion (for example, by waiting for messages to time out).
- This pattern also allows for arbitrarily sophisticated measures of health to be implemented.

Tradeoffs:

- As with the other patterns for robustness, monitoring requires more up-front work than not monitoring. It also requires additional runtime processing and network bandwidth.

5.2.8 Throttling

In contexts where demand on the system, or a portion of the system, is unpredictable, the Throttling pattern can be employed to ensure that the system will continue to function consistently with its service-level agreements and that resources are apportioned consistently with system goals. The idea is that a component, such as a service, monitors its own performance measures (such as its response time), and when it approaches a critical threshold it throttles incoming requests. A number of throttling strategies can be employed—each of these corresponding to a "Control Resource Demand" tactic [Bass 2012]. For example, the throttling could mean rejecting requests from certain sources (perhaps based on their priority, criticality, or the amount of resources that they have already consumed), disabling or slowing the response for specific request types (for less essential functions), or reducing response time evenly for all incoming requests.

Benefits for robustness:

- As we described in Section 2, the goal of robustness is for a software-intensive system to keep working, consistently with its specifications, despite the presence of external stresses, over a long period of time. The Throttling pattern aids in this objective by ensuring that essential services remain available, at the cost of degrading some kinds or qualities of the system's functionality.

Tradeoffs:

As with the other patterns for robustness, throttling requires more up-front work than not throttling, and it requires a small amount of runtime processing to monitor critical resource usage levels and to implement the throttling policy.

6 Analyzing for Robustness

An analyst's job is to judge the appropriateness of the mechanisms built into the architecture of a system S, in light of the robustness stimuli that the system will need to withstand. And as stated above, "appropriateness" is really a function of the risks and costs of the anticipated integrations. Analysts can specify these potential or anticipated integrations using scenarios, as we exemplified above, and for consistency and repeatability they can guide stakeholders to derive those scenarios from the robustness general scenario.

Analyzing for robustness at different points in the software development lifecycle will take different forms. The different analysis options are sketched in Table 3. If analysts only have a reference architecture or a functional architecture, for example, then they cannot make detailed predictions or claims about the level of difficulty associated with achieving a desired robustness response measure. What the analyst can employ, at that early stage, is a checklist or tactics-based questionnaire. These analysis techniques will reveal the designer's intentions with respect to robustness.

On the other hand, if the analysts have received a defined and documented [Clements 2010] product architecture—perhaps including views such as Functional, Hardware, and Software Architecture—but little or no coding has been done, they can still employ checklists and tactics-based questionnaires to understand the design intent. But as shown in Table 3, the analysts can also begin to think about employing analysis models.

The point is that there is no one-size-fits-all analysis methodology and tools that we can recommend: The analysis team needs to respond appropriately to whatever artifacts have been made available for analysis. And the analysis team and the product owner need to understand that the accuracy of the analysis and expected degree of confidence in the analysis results will vary according to the quality of the available artifacts.

Table 3: Lifecycle Phases and Possible Analyses for Robustness

Lifecycle Phase	Typical Available Artifacts	Possible Analyses
Early Design	Set of selected mechanisms/tactics/patterns	Checklist Tactics-based questionnaire
Software Architecture Defined	Set of containers for functionality (e.g., modules, services, microservices) and their interfaces	Checklist Tactics-based questionnaire Model-based analyses
Implemented System	Set of elements—services, processes, threads, etc.—and their interaction mechanisms—calls, pub/sub, messages, etc.—along with the mapping of these elements to hardware and networks	Checklist Measurement-based analyses Model-based analyses

6.1 Tactics-Based Questionnaire

Architectural tactics have been presented thus far as design primitives, following the concepts and principles introduced in *Software Architecture in Practice* [Bass 2012] and *Designing Software Architectures* [Cervantes 2016]. However, since tactics are meant to cover the entire space of architectural design possibilities for a quality attribute, we can use them in analysis as well. Each

tactic is a design option for the architect at design time. But used in hindsight, they represent a taxonomy of the entire design space for robustness.

Specifically, we have found these tactics to be very useful guides for interviews with the architecture team. (Although the information could be derived from other sources such as document review or reverse engineering, interviews with the architect are typically quite efficient and can be very revealing.) These interviews help analysts gain rapid insight into the design approaches taken, or not taken, by the architect and the risks therein. These might be risks of omission (e.g., the architect did not use this tactic and should have), risks of commission (e.g., this tactic is not really required, which increases costs with little or no commensurate benefit), risks on how a tactic was implemented (e.g., the team implemented a tactic themselves when a better, more mature alternative already existed), or managerial risks (e.g., the tactic has not been properly communicated to the team).

For example, consider the list of robustness tactics-inspired questions presented in Table 4. The analyst asks each question and records the answers in the table.

Table 4: Example Tactics-Based Robustness Questions

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Detect Faults	Does the system use an element to **monitor** the state of health of other parts of the system? A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.				
	Does the system use **ping/echo** to detect a failure of an element or connection or network congestion?				
	Does the system use a **heartbeat**—a periodic message exchange between a system monitor and a process—to detect a failure of an element or connection or network congestion?				
	Does the system use a **timestamp** to detect incorrect sequences of events in distributed systems?				
	Does the system employ **condition monitoring** to check conditions in a process or device or to validate assumptions made during the design?				

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Does the system employ **sanity checking** to check the validity or reasonableness of specific operations or outputs of a computation?				
	Does the system use **voting** to check that replicated elements are producing the same results? The replicated elements may be identical replicas, functionally redundant, or analytically redundant.				
	Does the system use **exception detection** to detect a system condition that alters the normal flow of execution (e.g., system exception, parameter fence, parameter typing, or timeout)?				
	Can the system do a **self-test** to test itself for correct operation?				
Recover from Faults (Preparation and Repair)	Does the system employ **active redundancy** (hot spare)? In active redundancy, all nodes in a protection group (a group of nodes where one or more nodes are "active," with the remainder serving as redundant spares) receive and process identical inputs in parallel, allowing redundant spares to maintain synchronous state with the active node(s).				
	Does the system employ **passive redundancy** (warm spare)? In passive redundancy, only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates.				
	Does the system employ **spares** (cold spares)? Here, redundant spares of a protection group remain out of service until a failover occurs, at which point a power-on-reset procedure is initiated on the redundant spare before it is placed in service.				

Tactics Group	Tactics Question	Sup-ported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Does the system employ **rollback**, so that it can revert to a previously saved good state (the "rollback line") in the event of a fault?				
	Does the system employ **exception handling** to deal with faults? Typically, the handling involves either reporting the fault or handling it, potentially masking the fault by correcting the cause of the exception and retrying.				
	Can the system perform in-service **software upgrades** to executable code images in a non-service-affecting manner?				
	Does the system systematically **retry** in cases where the element or connection failure may be transient?				
	Can the system simply **ignore faulty behavior** (e.g., ignoring messages sent from a source when it is determined that those messages are spurious)?				
	Does the system have a policy of **degradation** when resources are compromised, maintaining the most critical system functions in the presence of element failures and dropping less critical functions?				
	Does the system have consistent policies and mechanisms for **reconfiguration** after failures, reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible?				
Recover from Faults (Reintroduction)	Can the system operate a previously failed or in-service upgraded element in a "**shadow** mode" for a predefined time prior to reverting the element back to an active role?				

Tactics Group	Tactics Question	Sup-ported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	If the system uses active or passive redundancy, does it also employ **state resynchronization**, to send state information from active to standby elements?				
	Does the system employ **escalating restart** to recover from faults by varying the granularity of the element(s) restarted and minimizing the level of service affected?				
	Can message processing and routing portions of the system employ **non-stop forwarding**, where functionality is split into supervisory and data planes? In this case, if a supervisor fails, a router continues forwarding packets along known routes while protocol information is recovered and validated.				
Prevent Faults	Can the system **remove elements from service**, temporarily placing a system element in an out-of-service state for the purpose of mitigating potential system failures?				
	Does the system employ **transactions**, bundling state updates so that asynchronous messages exchanged between distributed elements are *atomic*, *consistent*, *isolated*, and *durable*?				
	Does the system use a **predictive model** to monitor the state of health of an element to ensure that the system is operating within nominal parameters? When conditions are detected that are predictive of likely future faults, the model initiates corrective action.				

When using this set of questions in an interview, the analyst records whether or not each tactic is supported by the system's architecture, according to the opinions of the architect. When analyzing an existing system, the analyst can additionally investigate

- whether there are any obvious risks in the use (or non-use) of this tactic. If the tactic has been used, record how it is realized in the system (e.g., via custom code, generic frameworks, or externally produced elements).

- the specific design decisions made to realize the tactic and where in the code base the implementation (realization) may be found. This is useful for auditing and architecture reconstruction purposes.
- any rationale or assumptions made in the realization of this tactic.

These questionnaires can be used by an analyst, who poses each question to the architect and records the responses, as a means of conducting an architecture analysis. To use these questionnaires, simply follow these four steps:

1. For each tactics question, fill the "Supported" column with Y if the tactic is supported in the architecture and with N otherwise. The tactic name in the "Tactics Question" column is bolded.
2. If the answer in the Supported column is Y, then in the "Design Decisions and Location" column describe the specific design decisions made to support the tactic and enumerate where these decisions are manifested (located) in the architecture. For example, indicate which code modules, frameworks, or packages implement this tactic.
3. In the "Risk" column, indicate the anticipated or experienced difficulty or risk of implementing the tactic using a scale: H = High, M = Medium, L = Low. For example, a tactic that is of medium difficulty or risk to implement (or which is anticipated to be of medium difficulty, if it has not yet been implemented) would be labeled M.
4. In the "Rationale and Assumptions" column, describe the rationale for the design decisions made (including a decision to *not* use this tactic). Briefly explain the implications of this decision. For example, explain the rationale and implications of the decision in terms of the impact on cost, schedule, evolution, and so forth.

While this interview-based approach might sound simplistic, it can be very powerful and insightful. In architects' daily activities, they likely do not take the time to step back and consider the bigger picture. A set of interview questions such as those shown in Table 4 forces an architect to do just that. And this process can be quite efficient: A typical interview for a single quality attribute takes between 30 and 90 minutes.

6.2 Architecture Analysis Checklist for Robustness

As presented in the work of Bass and colleagues, one can view an architecture design as the result of applying a collection of design decisions [Bass 2012]. We view architecture design and analysis as two sides of the same coin [Cervantes 2016]: any design decision made by an architect should be analyzed. Design and analysis are not distinct activities—they are intimately related. What we present next is a systematic categorization of these decisions so that an architect or analyst can focus attention on those design dimensions likely to be most troublesome.

There are seven major categories of design decisions that face an architect. These decisions will affect both software and, to a lesser extent, hardware architectures. These are

1. allocation of responsibilities
2. coordination model
3. data model
4. mapping among architectural elements

5. management of resources
6. binding time
7. choice of technology

These categories are not the only way to classify architectural design decisions, but they do provide a rational (and exhaustive) division of concerns. The concerns addressed in these categories might overlap, but it's all right if a particular decision exists in two different categories because the duty of the architect and of the analyst is to ensure that every important decision has been considered.

Some of these design decisions might be trivial. For example, an architect may have no choice of technology decisions to make if he is required to implement the software on a prespecified platform over which he has little or no control. Or for some applications, the data model might be trivial. But for other categories of design decisions, the architect might have considerable latitude.

For each quality attribute, we enumerate a set of questions—a checklist—that will lead an analyst to question the decisions made, or not made, by the architect, and for some of these decisions to refine the questions into a deeper analysis. The checklist for robustness is presented below.

Category	Checklist
Allocation of responsibilities	Determine the system responsibilities that need to be robust. Within those responsibilities, ensure that additional responsibilities have been allocated to detect an omission, crash, incorrect timing, or incorrect response. Additionally ensure that there are responsibilities to • log the fault • notify appropriate entities (people or systems) • disable the source of events causing the fault • be temporarily unavailable • fix or mask the fault/failure • operate in a degraded mode
Coordination model	Determine the system responsibilities that need to be robust. With respect to those responsibilities, • ensure that coordination mechanisms can detect an omission, crash, incorrect timing, or incorrect response. Consider, for example, whether guaranteed delivery is necessary. Will the coordination work under conditions of degraded communication? • ensure that coordination mechanisms enable the logging of the fault, notification of appropriate entities, disabling of the source of the events causing the fault, fixing or masking the fault, or operating in a degraded mode. • ensure that the coordination model supports the replacement of the artifacts used (processors, communications channels, persistent storage, and processes). For example, does replacement of a server allow the system to continue to operate? Determine if the coordination will work under conditions of degraded communication, at startup/shutdown, in repair mode, or under overloaded operation. For example, how much lost information can the coordination model withstand and with what consequences?
Data model	Determine which portions of the system need to be robust. Within those portions, determine which data abstractions, along with their operations or their properties, could cause a fault of omission, a crash, incorrect timing behavior, or an incorrect response. For those data abstractions, operations, and properties, ensure that they can be disabled, be temporarily unavailable, or be fixed or masked in the event of a fault. For example, ensure that write requests are cached if a server is temporarily unavailable and performed when the server is returned to service.

Mapping among architectural elements	Determine which artifacts (processors, communication channels, persistent storage, or processes) may produce a fault: omission, crash, incorrect timing, or incorrect response. Ensure that the mapping (or re-mapping) of architectural elements is flexible enough to permit recovery from the fault. This may involve a consideration of • which processes on failed processors need to be reassigned at runtime • which processors, data stores, or communication channels can be activated or reassigned at runtime • how data on failed processors or storage can be served by replacement units • how quickly the system can be reinstalled based on the units of delivery provided • how to (re-)assign runtime elements to processors, communication channels, and data stores When employing tactics that depend on redundancy of functionality, the mapping from modules to redundant elements is important. For example, it is possible to write one module that contains code appropriate for both the active element and backup elements in a protection group.
Management of resources	Determine what critical resources are necessary to continue operating in the presence of a fault: omission, crash, incorrect timing, or incorrect response. Ensure there are sufficient remaining resources in the event of a fault to log the fault; notify appropriate entities (people or systems); disable the source of events causing the fault; be temporarily unavailable; fix or mask the fault/failure; and operate normally, in startup, shutdown, repair mode, degraded operation, and overloaded operation. Determine the availability time for critical resources, what critical resources must be available during specified time intervals, time intervals during which the critical resources may be in a degraded mode, and repair time for critical resources. Ensure that the critical resources are available during these time intervals. For example, ensure that input queues are large enough to buffer anticipated messages if a server fails so that the messages are not permanently lost.
Binding time	Determine how and when architectural elements are bound. If late binding is used to alternate between elements that can themselves be sources of faults (e.g., processes, processors, communication channels), ensure the chosen robustness strategy is sufficient to cover faults introduced by all sources. For example: • If late binding is used to switch between artifacts such as processors that will receive or be the subject of faults, will the chosen fault detection and recovery mechanisms work for all possible bindings? • If late binding is used to change the definition or tolerance of what constitutes a fault (e.g., how long a process can go without responding before a fault is assumed), is the recovery strategy chosen sufficient to handle all cases? For example, if a fault is flagged after 0.1 ms, but the recovery mechanism takes 1.5 s to work, that might be an unacceptable mismatch. • What are the robustness characteristics of the late binding mechanism itself? Can it fail?
Choice of technology	Determine the available technologies that can (help) detect faults, recover from faults, and reintroduce failed elements. Determine what technologies are available that support the response to a fault (e.g., event loggers). Determine the robustness characteristics of chosen technologies themselves: What faults can they recover from? What faults might they introduce into the system?

6.3 Robustness Models and Analysis Techniques

The field of reliability and availability modeling has existed for decades, dating back at least to World War II era research, where it was applied to physical systems, typically control systems and their associated electronics. Traditional reliability approaches are based on the hardware "wear-out" model. As software became more prominent in systems, many of the hardware approaches were "translated" to software. As time went on, software engineers discovered that

software failure mechanisms are different than hardware failure mechanisms, and so they developed new techniques and approaches.

Typically, evaluation of reliability falls into two broad categories: measurement-based techniques and model-based techniques. In this section we provide a brief overview of model-based techniques. We focus on models because these can be applied earlier in the system lifecycle so that designers can explore alternative design options. In this way the designer can satisfy robustness requirements while balancing other concerns such as cost, performance, and resource utilization.

Modern reliability and availability analysis models broadly fall into three categories:

1. black-box models, where a system is viewed as a monolith, where there is no insight into its internal structure, and where all measurement is focused on its input and outputs
2. white-box models, where the internal structure of a system is explicitly considered and modeled using probabilistic methods
3. combined models, which treat subsystems as black boxes but which analyze the system as a white box (sometimes termed "gray-box testing")

And the most common modeling formalisms can be categorized as either state-space or non-state-space techniques, as shown in Figure 3.

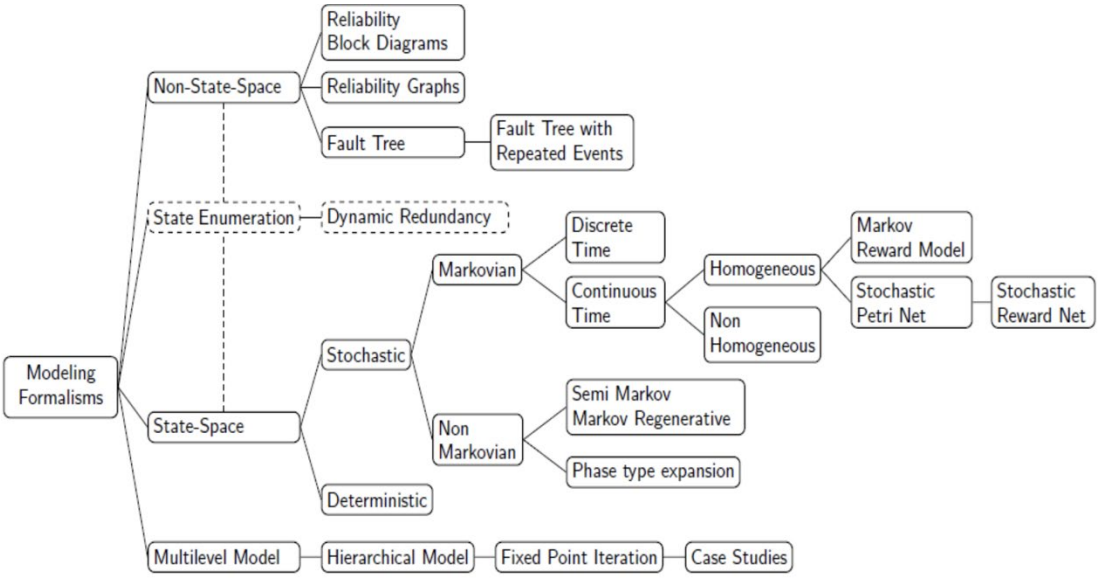


Figure 3: Reliability Modeling Formalisms [Source: Trivedi 2017, Figure 2.6, p. 28]

Architecture is about structure and relationships; accordingly, we will focus on modeling techniques where analysts need to know something about the internal structure of a system (i.e., white-box and combined models). These models help designers to perform "what if" analyses by allowing predictions to be made when changes to components' configurations, their relationships, or properties are made in the early design process. Modeling also aids in evolutionary design, where a set of iterative refinements of and predictions about the impact of changes in design decisions are key considerations in analyzing alternatives. "Having dependability modeling tools

continuously available for use … by the system designers is important because of the exploratory nature that tends to characterize human creative work." [Boyd 1998]. Boyd and Lau offer some advice on selecting the modeling technique that will best serve a system's needs: "Generally, the best strategy is to match the modeling method to the characteristics and required level of detail in the behavior of the system that must be modeled" [Boyd 1998]. They also recommend selecting the simplest appropriate modeling method. To get this decision right, therefore, an architect should understand "the characteristics, capabilities, and limitations of all modeling methods in the spectrum" [Boyd 1998].

Many questions can be answered through such modeling. Here are some examples:

- How reliable are the non-repairable components and systems?
- How available is the repairable system?
- Where in the system are there most likely to be faults or failures?
  - What are the failure rates of these components?
  - What are the desired repair rates?
  - What are the failure modes of these components?
- Where could the system benefit from redundancy?
- Where are the single points of failure?
- Which are the critical components and hardware?
  - Where is it important to lower fault detection and recovery times?
  - Where can monitoring have the most benefit? What are the monitoring frequencies?
  - What are the required repair rates?
  - Where could the system benefit from fault prevention strategies (e.g., defining and monitoring thresholds, retry)?
- How important is it to have spare capacity readily available?
- Where are the opportunities to apply tactics and patterns likely to have significant impact on important measures?

When considering modeling to answer important architectural questions, an important tradeoff between simplicity and fidelity needs to be accounted for when deciding which modeling techniques to use to analyze the design decisions. Figure 4 represents a spectrum of modeling techniques that generally become increasingly more complex and provide higher fidelity from left to right.

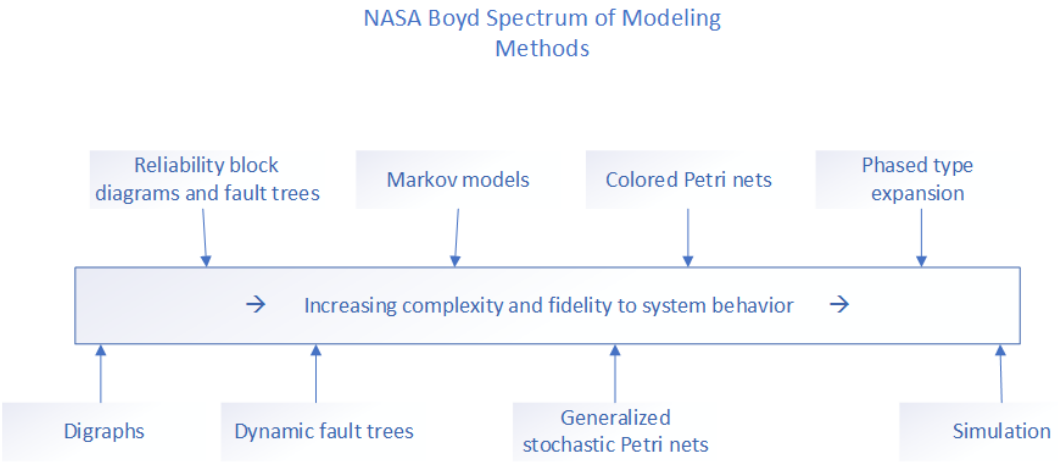


Figure 4: A Spectrum of Modeling Methods [Derived from Boyd 1998]

Non-state-based techniques such as reliability block diagrams (RBDs) and fault tree analysis (FTA) can be done early during the design process, often without experts. The initial iterations of these models provide only modest confidence in success or failure predictions, but they do not require detailed knowledge about the eventual system. The low up-front investment in these techniques can be easily recouped in terms of providing the ability to try "what if" design modifications early in the lifecycle and to predict the impact on important reliability measures. Both RBDs and FTA are typically refined as the team learns, through iterative design, to provide better predictions on system reliability and availability. These techniques are common starting points and provide inputs to modeling techniques further to the right in Figure 4.

State-based techniques such as Markov modeling, generalized stochastic Petri nets, and colored Petri nets are higher fidelity modeling techniques that allow designers to model more than success or failure. These state-based techniques allow them to model complex sets of failures and repair modes. Designers can predict the probability of being in a certain state (or place for Petri nets) at a certain step or time. These models often require modeling experts and deeper knowledge of the system. The barrier to adoption is thus higher since the modeling experts and designers often need to work together for long periods of time to generate and validate the models.

Both non-state-based and state-based techniques can help analysts make predictions for system robustness requirements and provide the ability to consider the consequences of different design decisions. These techniques are often used together to reason about reliability and availability in a system.

Our treatment of modeling is not meant to be complete or exhaustive. Our goals are to provide a window into a subset of modeling techniques available for analysts to explore and to show how they can be used to support analysis by illustrating how a design can be modeled and important quality attribute characteristics can be predicted.

We will briefly describe two non-state-based modeling techniques (RBDs and fault trees) and finish with a description of two stochastic state-based techniques (Markov models and Petri nets). We will show how an architect can walk an analyst through design changes (for example,

applying tactics or patterns to evolve an existing design) to show how they impact reliability and availability predictions from the models. These predictions can be used to provide evidence that important quality attribute measures, such as uptime or MTTF, will be met.

6.3.1 Non-state Based Modeling Techniques

The non-state-based techniques we will describe are RBDs and fault trees. These two modeling techniques can easily be converted into each other, but there may be information loss during the transformations (for example, identified causes could be lost converting a fault tree to an RBD).

Reliability block diagrams, or RBDs, are a graphical representation of the system that is used to assess the probability of successful operation. The blocks, or components, are linked based on the impact to robustness and they are the smallest unit considered in the analysis. These block diagrams primarily model system behavior using series and groups of components. Reliability for a series of components is predicted by taking the product of each component's probability of being operational.

The example in Figure 5 shows a series consisting of three components, where each is associated with a probability of being operational, estimated as .9 (sensor), .97 (controller), and .99 (actuator), respectively. In this case, the predicted reliability is $R = .9 * .97 * .99 = \sim.864$. Based on this prediction, the architect made a second iteration of the design and decided to use sensor redundancy to improve the overall reliability.

Figure 5: A Simple Series RBD Diagram

For systems that use redundancy or parallelism, the reliability for the group of components uses the probability for each redundant component to calculate the reliability of the group. Figure 6 represents the new design.

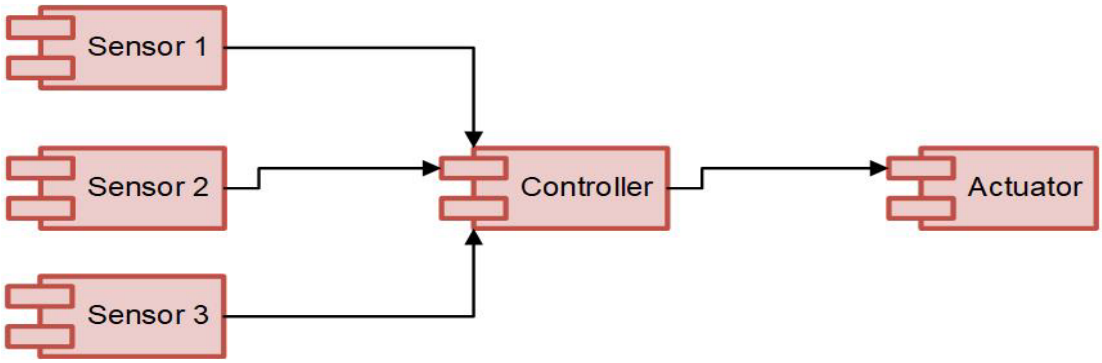


Figure 6: A Group of Sensors (Employing an Active Redundancy Tactic) to Improve Robustness

The designers changed the design and the model to introduce redundancy in the form of three redundant sensors. They focused on improving the lowest probability block, which is currently .9. Thus, the reliability of this component will need to be recalculated. The calculation for the reliability of a group of components is represented by the following equation [Cepin 2011]:

$$R = 1 - \prod_{i=1}^{n} (1 - R_i)$$

In this case, the calculation for the reliability of the group of sensors is

$$R = 1 - (1-.9)(1-.9)(1-.9) = .999$$

The reliability of this system can now be calculated as .999 * .97 * .99. This "what if" design change dramatically improves the reliability for the system from ~.864 to an overall predicted reliability of ~.959. As an analyst you can see the power of this type of rapid analysis: it quickly shines a light on tradeoffs that designers can make. This example shows that three redundant lower cost sensors may provide better reliability than one higher cost sensor that has a higher individual probability of being in service. The optimal decision may also depend on other contextual factors such as resource utilization, power consumption, and weight. The analyst should be aware of these other factors during an evaluation.

The graph in Figure 7 shows examples of how the reliability of a component or subsystem improves as the number of redundant components increases.

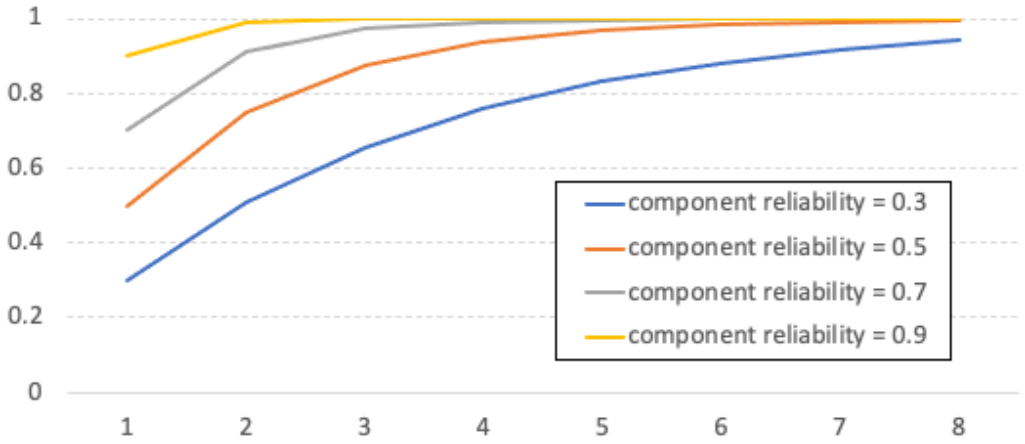


Figure 7: Number of Parallel Components [Derived from Cepin 2011]

The strengths of RBDs are that they are fairly simple to create and the calculations for the analysis are straightforward. They are an excellent way for analysts to explore what-if scenarios with designers and to identify components where design changes such as redundancy have a large impact on overall reliability. The weakness of this technique is that the calculation is only as good as the estimates of the probabilities of the components being operational. RBDs also provide an all-or-nothing result, ignoring partial failures or degraded modes.

A *fault tree* is a top-down logical diagram that displays the interrelationships between a critical system event and its causes. Fault trees are concerned with the failure case, in contrast to RBDs,

which focus on probability of success. A fault tree has a top event that provides a description of the critical system event, basic events at the lowest level that have identified causes, and logic gates (e.g., and, or, voting or) that provide the logical relationship between the top event and the basic events [Vesely 2002].

The analyses that can be done with fault trees are both qualitative (e.g., identifying single points of failure) and quantitative (e.g., calculating system uptime, failure probabilities of operations). FTA can be used to calculate probabilities of failure for continuously operating non-repairable systems, and there are dynamic fault trees for continuously operating repairable systems [NRC 2015]. Figure 8 and Figure 9 are the equivalent fault trees for the simple series and the parallel sensors from the previous RBD examples.

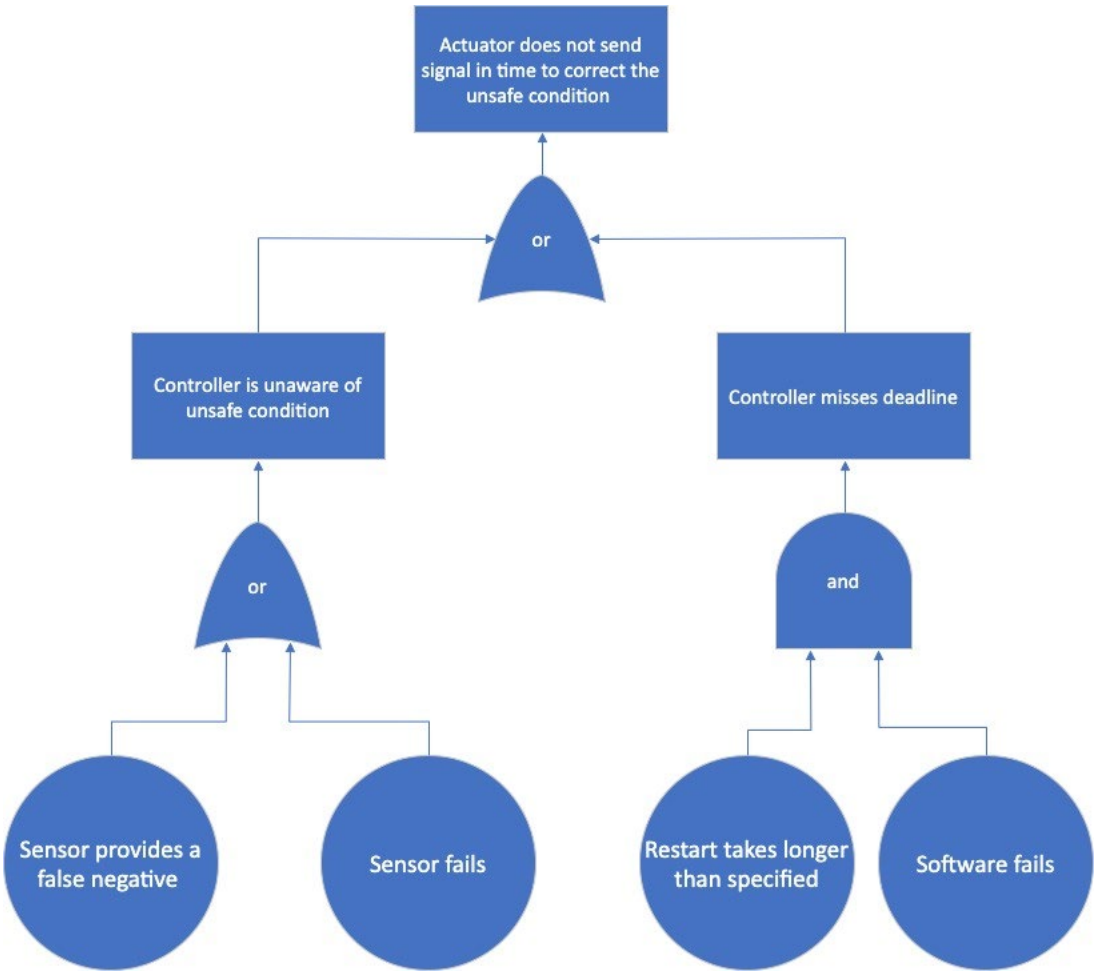


Figure 8: Equivalent Fault Tree of Simple Series RBD Diagram

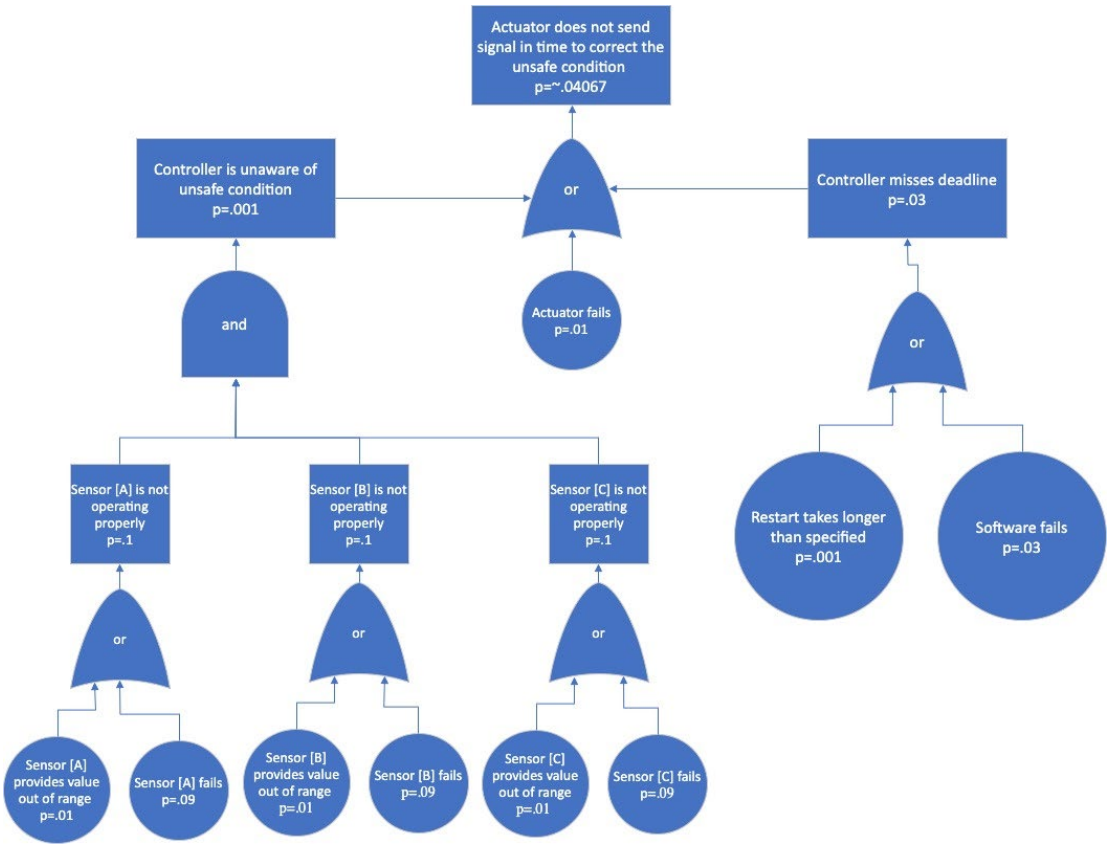


Figure 9: Equivalent Fault Tree for a Group of Sensors (Employing a Redundance Tactic) to Improve Robustness

We will not show the calculations for the fault trees, but note that the numbers are for probability of failure. For the top event, the failure probability is ~.041, in contrast to the RBD calculation of ~.959 reliability.

Below we will discuss the characteristics and metrics as well as the strengths and weaknesses of RBDs and fault trees. We discuss these together since many sources highlight the similarities of the techniques. It is important to acknowledge that fault trees focus on failures and their causes, while RBDs focus on successful operation.

Example Characteristics/Metrics from FTA and RBDs:

- We can calculate MTTF using the failure rate of our non-repairable example. Failure rate = .041 per 100 hours of operation. The MTTF = (1 /.041) * 100 = ~2439.24 hours.
- We can calculate the reliability at points in time for a non-repairable system [Reliability Analytics 2010–2020]. Units are hours.
  - $R(100) = .95983$
  - $R(200) = .92127$
  - $R(500) = .81465$
  - $R(2{,}500) = .35879$
  - $R(5{,}000) = .12873$

- If we change our model to a repairable system and estimate the MTTR, then we can calculate uptime and downtime percentages for our system. Assume our MTTR = 1 hour per failure, including detection time, and our MTTF previously calculated as ~2,439.24 hours will be our estimate for MTBF. If we run the system continuously for 1,000,000 hours, we would expect ~409.96 failures during this time period.
  - Predicted downtime would be ~409.96 hours or ~.04096%.
  - Predicted uptime would be ~999,590.04 hours or ~99.95904%.
- We can predict the probabilities of failure and success as illustrated in the example.
- With FTA and RBDs, analysts can reason about
  - where in the system developers can improve the failure rate and MTBF through retries
  - critical system components that require monitoring to reduce MTTR through early detection (You can also use these models to know where monitoring can be used to prevent failures, such as when resource thresholds are reached.)
  - critical system components that could be made more reliable with runtime diagnostics

Advantages/Strengths:

- These techniques support qualitative analysis to identify critical components that need monitoring, low MTTR, and other strategies that improve robustness.
- There is a lower barrier to adoption than with state-based techniques. RBDs and fault trees can be used by designers. Markov models and Petri nets generally require more modeling expertise.
- They allow concise description of the system under study and can be evaluated quickly as the architect changes the design.
- The calculations of component uptime, downtime, and probability of failure are supported by tools.
- These techniques provide insight into where hardware, software, and analytic redundancy can improve robustness.
- The fault trees and RBD models are often more closely aligned with the structure of the design than Markov models and Petri nets.

Disadvantages/Weaknesses:

- It can be difficult to model a complex system that includes a large number of components. They can become difficult to manage and require significant effort to complete.
- These models do not consider partial failures and degraded modes.
- The probability calculations are only as good as the inputs. Early in the lifecycle analysts often deal with estimates, and high confidence analysis requires more information about the system. This disadvantage will be alleviated somewhat when the models are kept up to date to better support analysis through information gained during each design iteration.
- FTA and RBDs “cannot represent dependencies occurring in real systems, such as imperfect coverage, correlated failures, repair dependencies, non-zero detection/reconfiguration time, performance-reliability dependence, and phased mission system models” [Trivedi 2017].
- The FTA modeling approach is not useful for systems where components have interdependencies.

6.3.2 State-Based Modeling Techniques

The state-based technique we primarily explore is Markov modeling. We also summarize Petri nets and a few of the challenges of Markov models that Petri nets address.

The basic components of Markov models are discrete and countable states and a set of transitions between these states. The model must always be in a state and cannot be in more than one state at a time. Also, from time to time the model must transition to other states. The states often represent configurations or operational states. In their introduction to Markov modeling, Boyd and Lau explain that when modelling for robustness, states can be normal operation, presence of faults, failure, and degraded modes. The transitions define where the model can go from one state to another and the length of time needed to go from one state to another, and the transition time can be constant or time dependent [Boyd 1998]. For robustness, the transition rates are often related to failure rates and repair rates.

A Markov process is stochastic, and probability distributions for the future behavior of a system depend only on the current state of the system and not on any previous state. A Markov chain is a Markov process that represents system behavior in terms of random transitions between discrete states.

Markov models of systems can be represented through a transition probability matrix where each row is a state and each column represents the states that can be transitioned to from the current state. They are also represented as acyclic graphs for non-repairable systems and cyclic graphs for repairable systems. For systems that are non-repairable, the model's prediction is the probability that the system has not experienced failure (e.g., reliability). For systems that are repairable, the prediction from the model is the probability that the system will be available or operational. The Markov models allow analysts to predict the probabilities that the system in question will be in undesirable states based on the current design, and this knowledge allows designers to make informed decisions to reduce unacceptable probabilities.

In models of Markov processes, time can be modeled as discrete (i.e., transitions occur at unit-timed intervals, and a transition must occur at each interval) and represented as a discrete-time Markov chain (DTMC). Time can also be modeled as continuous (i.e., transitions can occur at any real timed interval) and represented as a continuous-time Markov chain (CTMC). CTMCs can be time homogeneous or nonhomogeneous. To be considered time homogenous, the CTMC must have the same holding time every time the process enters state x. When exiting state x, all possible transitions and their probabilities must remain the same. Nonhomogeneous CTMCs can be modeled using Weibull distributions that allow for modeling of failure rates that vary over time (e.g., hardware burn-in and end of life).

"Markov chains can be used to analyze system reliability in terms of error states, occurrences, and propagations" [Delange 2014]. We will first explore a simple example of a DTMC to estimate the probability of being in certain states after a number of steps for a repairable system. We will then explore a time-homogenous CTMC and use it to show a few iterations of analysis and refactoring to improve the prediction of certain characteristics.

Our abstract DTMC example models a system that has three controllers, which each have the capacity to handle the management of 100 sensors and 50 actuators. This simple model can give analysts a prediction of the probability that the system will be in degraded modes of operation.

Let's suppose we have the following scenario for degraded modes that we must satisfy:

Scenario Part	Value
Source	Internal
Stimulus	A controller fails.
Artifact	A controller
Environment	There are three controllers. One controller provides minimally acceptable service.
Response	The system continues to process inputs from 200 sensors and 100 actuators.
Response Measure	Minimally acceptable service is available greater than 99.99% of the time.

The resulting model will have four states. The states are the following: all three controllers are fully operational with the system at full capacity; two controllers are operational, reducing capacity; one controller is operational, further reducing system capacity; and all controllers are offline, resulting in no capacity to manage sensors and actuators. As we learn more about the system, additional transitions could be added (e.g., two controllers are repaired at the same step, adding a transition from one controller being operational back to three being operational). All possible transitions from the current state to the potential next state must be equal to 1. The diagram and matrix in Figure 10 illustrate how our Markov process can be represented.

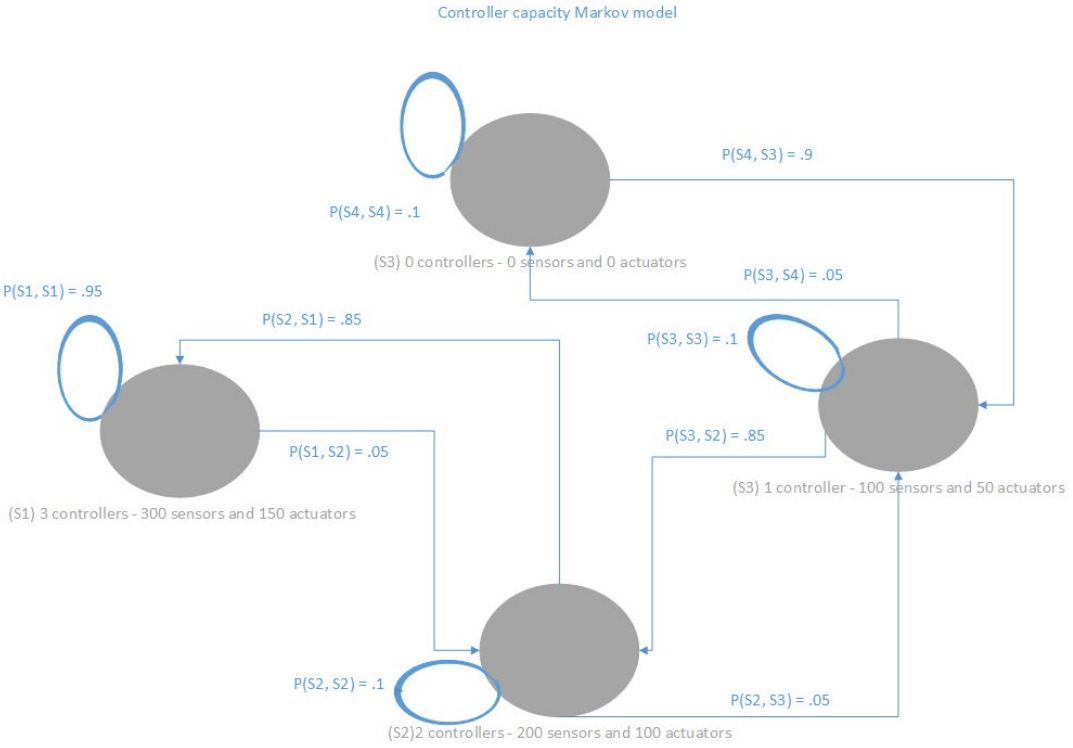


Figure 10: Markov Model Representing the States That Correspond to System Capacity

$$P = \begin{pmatrix} .95 = (S1,S1) & 0.05 = (S1,S2) & 0 = (S1,S3) & 0 = (S1,S4) \\ .85 = (S2,S1) & .1 = (S2,S2) & .05 = S2,S3) & 0 = S2,S4) \\ 0 = (S3,S1) & 0.85 = (S3,S2) & .1 = (S3,S3) & .05 = (S3,S4) \\ 0 = (S4,S1) & 0 = (S4,S2) & 0.9 = (S4,S3) & 0.1 = (S4,S4) \end{pmatrix}$$

The matrix above shows the capacity Markov model matrix (P) transition probabilities for the four states. The state transitions (in parentheses) are usually omitted but are included here for clarity. This transition matrix was input into a DTMC simulation tool provided by the Technische Universität Clausthal Institute of Mathematics [Clausthal n.d.]. The results—the relative frequencies of each state—are shown in Figure 11.

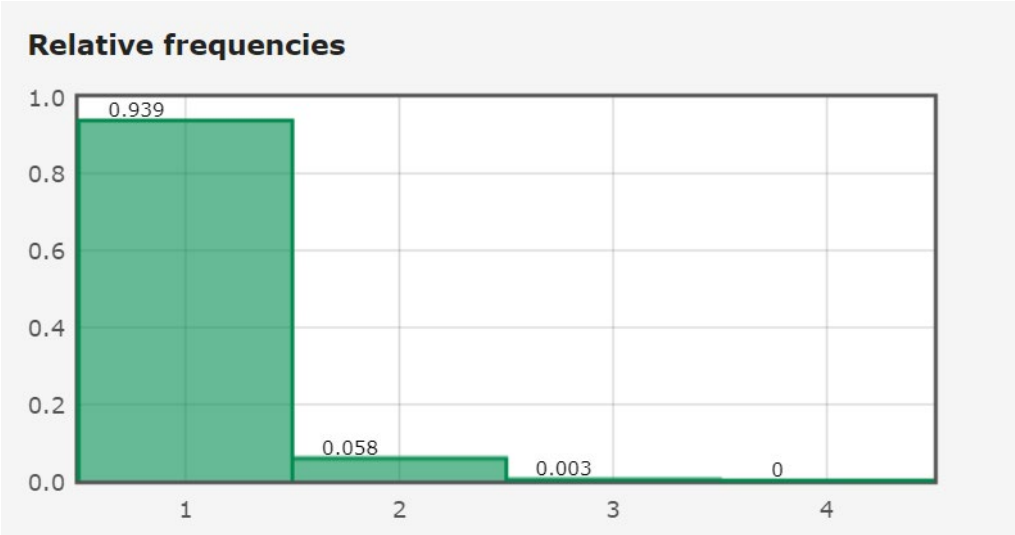


Figure 11: Relative Frequencies Calculated for Each State in Our Markov Model from Our Initial Design

What information does this analysis provide to an analyst?

- There is a ~93.9% probability that the system is at full capacity (State 1 probability).
- There is a ~99.7% probability that the system will be able to handle 200 sensors and 100 actuators (State 1 and State 2 combined probabilities).
- There is a ~0.3% probability that the system can only handle 100 sensors and 50 actuators (the State 3 probability).
- We estimate a ~99.999% uptime, assuming uptime is defined as minimally acceptable service reflected by States S1, S2, and S3. The graph shows zero for State S4, but that appears to be an artifact of rounding. We do observe that in simulation State S4 was reached on occasion.
- The initial design does fulfill the 99.99% availability of minimal service requirement defined in our scenario.

We will now present a simple CTMC that models two redundant controllers. We will show the relationship between design decisions and how you can use modeling as an effective tool for analyzing the initial design. Then we will show how design changes can affect the predictions from the model. By modeling in this way, you can collect evidence that important quality attribute goals are met. Lastly, we illustrate how decisions made to support one quality attribute can have an impact on the quality attribute measures that you are currently analyzing, and analysts can therefore note tradeoffs. If designers understand the relative priorities of the two competing

qualities, then they can make design changes in a disciplined way and analyze their impact through the model.

Let's suppose we have a scenario that we must satisfy the following:

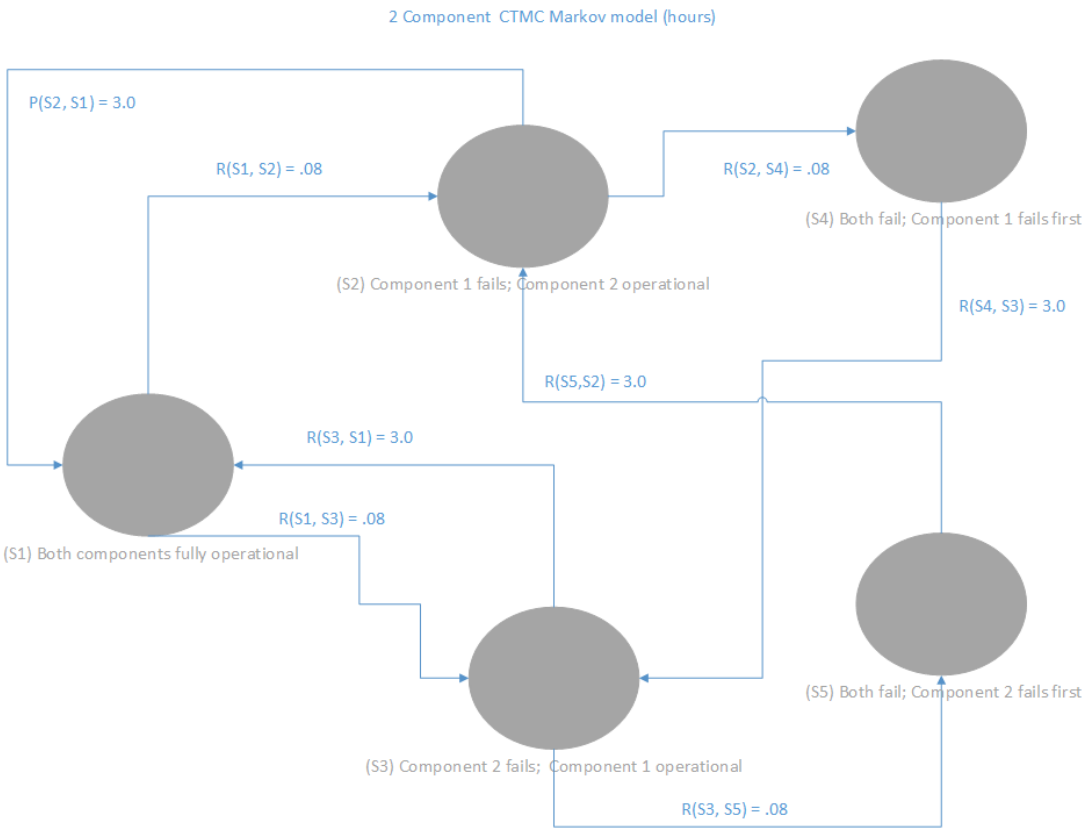
Scenario Part	Value
Source	Internal
Stimulus	A controller fails.
Artifact	A controller
Environment	There are two controllers. One controller is sufficient to process all critical features.
Response	The system continues to process all critical inputs.
Response Measure	Critical features are available 99.9% of the time.

The inputs to the CTMC model are the transition rates in the transition rate matrix. The mean failure rate is .08 failures per hour, and the mean repair rate is 3 per hour. We add an assumption that components are repaired in the order they fail and that they are identical and fully synchronized. The model has no absorbing states; therefore, it is repairable.

The CTMC states are as follows:

- State 1, both controllers are operational. There are two transitions: (S1,S2) and (S1,S3).
- State 2, Controller 1 fails, and Controller 2 is operational. There are two transitions: (S2,S1) and (S2,S4).
- State 3, Controller 2 fails, and Controller 1 is operational. There are two transitions: (S3,S1) and (S3,S5).
- State 4, both controllers fail, and Controller 1 fails first. There is one transition: (S4,S3).
- State 5, both controllers fail, and Controller 2 fails first. There is one transition: (S5,S2).

The diagram in Figure 12 represents the states and the transition rates.



$$Q = \begin{pmatrix} -.16 = (S1,S1) & .08 = (S1,S2) & 0.08 = (S1,S3) & 0 = (S1,S4) & 0 = (S1,S5) \\ 3.0 = (S2,S1) & -3.08 = (S2,S2) & 0 = (S2,S3) & .08 = (S2,S4) & 0 = (S2,S5) \\ 3.0 = (S3,S1) & 0 = (S3,S2) & -3.08 = (S3,S3) & 0 = (S3,S4) & .08 = (S3,S5) \\ 0 = (S4,S1) & 0 = (S4,S2) & 3.0 = (S4,S3) & -3.0 = (S4,S4) & 0 = (S4,S5) \\ 0 = (S5,S1) & 3.0 = (S5,S2) & 0 = (S5,S3) & 0 = (S5,S4) & -3.0(S5,S5) \end{pmatrix}$$

Figure 12: CTMC State Transition Diagram and CTMC State Transition Matrix

We now have all the inputs, and our model is set up. The transition matrix can be fed into a tool that supports CTMCs. We used the CTMC simulation tool provided by the Technische Universität Clausthal Institute of Mathematics for our simulation.

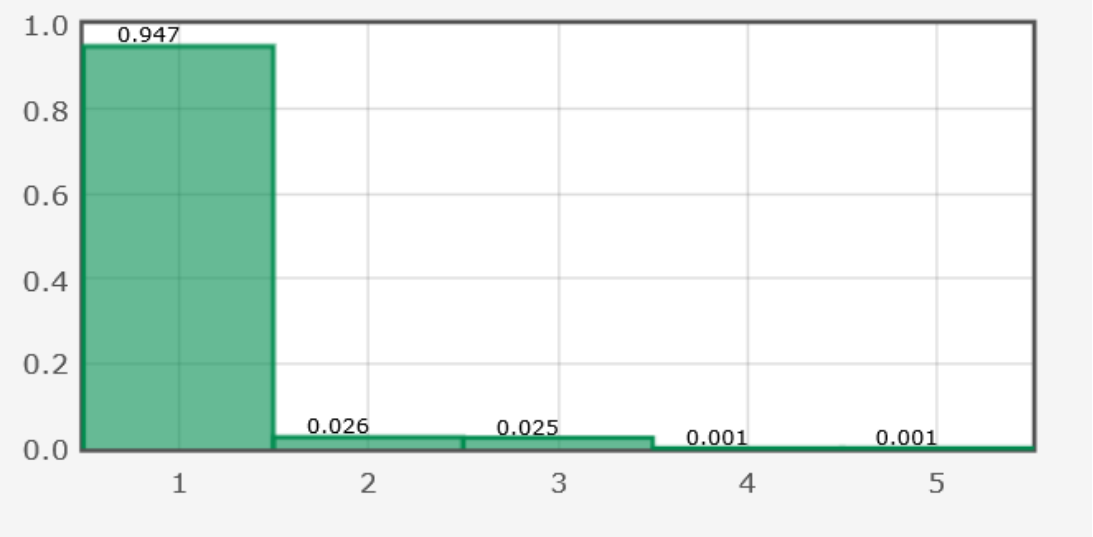


Figure 13: Relative Frequencies Calculated for Each of the States in Our Markov Model from Our Initial Design

What information does this provide to an analyst?

- There is a ~94.7% probability that both components are operational (State 1 probability).
- ~5.1% of the time, only one component is operational (State 2 and State 3 combined probability).
- ~0.2% of the time, both components are down (State 4 and State 5 combined probability).
- The uptime is ~99.8% (assuming uptime is defined as one component being operational and processing critical features).
- The initial design does not fulfill the 99.9% availability of critical features defined in our scenario.

Given this result, the designers could now refactor the design to attempt to improve the availability of critical features. The first step is to find the parameters that can be impacted through design changes. In this model the parameters are the failure rate of .08 per hour and the repair rate of 3 per hour. The designers should therefore look for tactics that reduce the failure rate or reduce the restart time. For example, looking through the FTA that provided the initial failure rate, they might have discovered the following two causes: there were memory leaks in a subcomponent S1, causing hanging processes that led to failures.

The designers then made two choices to address these causes: to use the substitution tactic by integrating a highly reliable commercial off-the-shelf product that implements the features of the S1 and to improve the process monitoring, by adding more monitors, to restart problematic processes when faults are detected. Given these changes, they update the fault tree and estimate that these changes will reduce the failure rate to .07 per hour.

After improving the failure rate by adding more process monitors and using substitution to reduce the number of failures, we now predict an improvement.

$$Q2 = \begin{pmatrix} -.14 = (S1,S1) & .07 = (S1,S2) & 0.07 = (S1,S3) & 0 = (S1,S4) & 0 = (S1,S5) \\ 3.0 = (S2,S1) & -3.07 = (S2,S2) & 0 = (S2,S3) & .07 = (S2,S4) & 0 = S2,S5) \\ 3.0 = (S3,S1) & 0 = (S3,S2) & -3.07 = (S3,S3) & 0 = (S3,S4) & .07 = (S3,S5) \\ 0 = (S4,S1) & 0 = (S4,S2) & 3.0 = (S4,S3) & -3.0 = (S4,S4) & 0 = (S4,S5) \\ 0 = (S5,S1) & 3.0 = (S5,S2) & 0 = (S5,S3) & 0 = (S5,S4) & -3.0(S5,S5) \end{pmatrix}$$

Figure 14: CTMC State Transition Matrix for the Modified Design

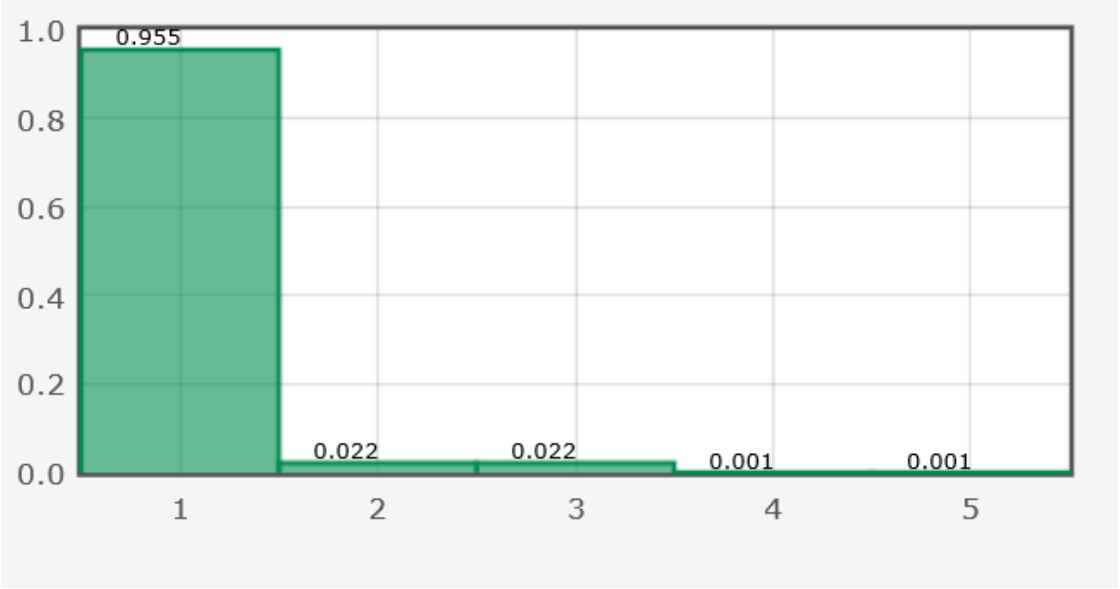


Figure 15: Relative Frequencies Calculated for Each of the States in Our Markov Model from Our New Design Reducing the Failure Rate

What information does this provide to an analyst?

- There is a ~95.5% probability that both components are operational (State 1 probability).
- ~4.4% of the time, only one component is operational (State 2 and State 3 combined probability).
- ~0.2% of the time, both components are down (State 4 and State 5 combined probability).
- The overall uptime is ~99.8% (assuming uptime is defined as one component being operational and processing critical features). This has improved the design, but not enough to reduce the State 4 and 5 probability predictions to zero at 3 decimal points of precision.
- The new design, while improved, does not fulfill the 99.9% availability of critical features defined in our scenario.

Given that the revised design has still not satisfied our scenario, we can examine contributors to restart time. The designers can search for approaches for other quality attributes that impact restart time. During this examination they might determine a few contributors to restart time:

- Late binding – Many components are bound at startup.
- There is an extensive use of configuration files.
- A large cache is populated at startup.

Since modifiability has the same priority as availability, we will avoid tradeoffs that affect modifiability (e.g., the use of late binding and configuration files) unless absolutely necessary. Performance is always important, but in this case slightly less so than availability. Upon examining the cache, the designers conclude that many data items related to soft deadlines are cached at startup. The designers thus opt to take a slight latency hit to fetch those items on an as-needed basis, rather than pre-fetching and caching them. This incremental state resynchronization will lead to a tradeoff with slightly greater latency, but it will improve availability. They estimate this will increase the repair rate to 4 per hour.

$$Q3 = \begin{pmatrix} -.14 = (S1,S1) & .07 = (S1,S2) & 0.07 = (S1,S3) & 0 = (S1,S4) & 0 = (S1,S5) \\ 340 = (S2,S1) & -4.07 = (S2,S2) & 0 = (S2,S3) & .07 = (S2,S4) & 0 = (S2,S5) \\ 4.0 = (S3,S1) & 0 = (S3,S2) & -4.07 = (S3,S3) & 0 = (S3,S4) & .07 = (S3,S5) \\ 0 = (S4,S1) & 0 = (S4,S2) & 4.0 = (S4,S3) & -4.0 = (S4,S4) & 0 = (S4,S5) \\ 0 = (S5,S1) & 4.0 = (S5,S2) & 0 = (S5,S3) & 0 = (S5,S4) & -4.0(S5,S5) \end{pmatrix}$$

Figure 16: CTMC State Transition Matrix for the Second Modification of the Design That Improves the Repair Rate

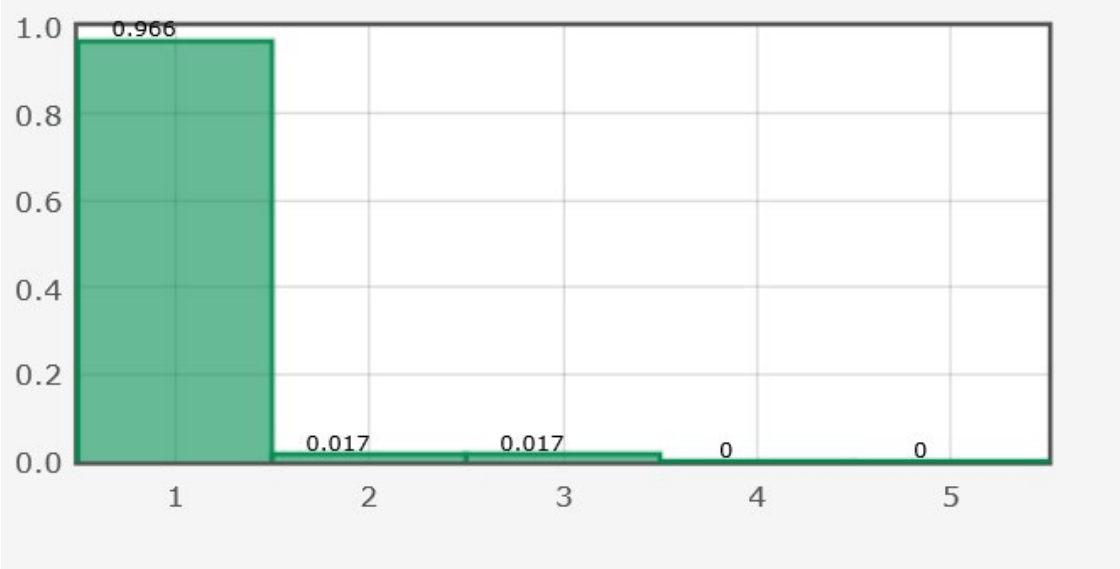


Figure 17: Relative Frequencies Calculated for Each of the States in Our Markov Model from Our New Design Improving the Repair Rate

What information does this provide to an analyst?

- There is a ~96.6% probability both components are operational (State 1 probability).
- ~3.4% of the time, only one component is operational (State 2 and State 3 combined probability).
- Less than 0.1% of the time, both components are down (State 4 and State 5 combined probability). The simulation did reach State 4 and 5, but less than .05% each.
- There is now greater than ~99.9% uptime predicted (assuming uptime is defined as one component being operational and processing critical features). The latest design change improved the design enough to reduce the State 4 and 5 probability predictions to zero at 3 decimal points of precision.
- The new design does fulfill the 99.9% availability of critical features defined in our scenario.

This simple example illustrates the relationships between design decisions and key model parameters that provide predictions of important system quality attribute characteristics for analysts. We also showed why it is important for designers and analysts to understand design decisions that affect other quality attributes and the tradeoffs that can impact other response measures. The existing modifiability and performance approaches had a negative impact on availability, so the designers determined that a tradeoff needed to be made. In this case, they were able to modify the cache to improve startup time, at the cost of runtime latency.

Below we will highlight some metrics that can be ascertained by Markov modeling and then list the strengths and weaknesses of this modeling.

Characteristics/Metrics:

- Uptime, downtime, calculating the number of steps or time spent in the states that correspond to failure
- Probability of failure in a number of steps or time period from the current state
- MTTF for non-repairable systems within a number of steps or time period from the current state
- MTBF for repairable systems within a number of steps or time period from the current state
- Percentage of operations or requests that need retries
- Time to detect fault or failure or MTTD
- Failover time
- Failover success rate
- Time spent in degraded modes of operation
- % of time spent in degraded modes

Advantages/Strengths:

- Markov modeling simplifies the transitions between states. Since the Markov property states that the next possible step depends only on where you are now and not any previous steps, it is easier to predict the probability of being in a certain state in a number of transitions.
- Many tools are available that support solving Markov models to abstract away complex mathematics. You only need to define your states and transition rates or probabilities.

Disadvantages/Weaknesses:

- A complete accounting of all possible states and transitions is often not possible. When formulating a Markov model of a complex system, it is difficult to ensure that all the possible combinations of events in a subsystem have been considered. In our simple CTMC example, we had 5 states. If we expanded to model five controllers, we would need 32 states plus an additional number to track the order in which failures occurred.
- The property of not relying on previous states and only the current state may not allow you to properly model your system. If a meaningful prediction relies on how or when you arrived in the current state, then more complex modeling techniques will be needed.

Petri nets are often used to model systems with many components and alleviate, to some degree, the state management issues of Markov models. A Petri net is a directed graph with places and transitions. The places each define a capacity for tokens and represent conditions within the

system being modeled. The transitions represent events occurring in the system that may cause change in the condition of the systems. Arcs connect places to transitions and transitions to places. There are never arcs directly from place to place or from transition to transition. A Petri net is an *n*-tuple (set of places, set of transitions, input arcs, output arcs, or markings).

We adopt the following definitions for modeling Petri nets [Bobbio 1990]:

- Input arcs are directed arcs drawn from places to transitions. They represent the conditions that need to be satisfied for an event to be activated.
- Output arcs are directed arcs drawn from transitions to places. They represent the conditions resulting from the occurrence of the event.
- Input places of a transition are the set of places that are connected to the transition through input arcs.
- Output places of a transition are the set of places to which output arcs exist from the transition.
- Tokens are represented as dots or integers associated with places.
- A marking of a Petri net is a vector listing the number of tokens in each place of the net. Every Petri net defines an initial marking.
- When input places of a transition have the required number of tokens, the transition is enabled.
- An enabled transition may fire removing one token from each input place and depositing one token in each of its output places.

Petri nets are also often extended by associating time with the firing of transitions. In a stochastic Petri net (SPN), the firing times are considered random variables. Our previous CTMC example assumed exponential distributions, which can be modeled by an SPN with the same assumption.

Below is an example of an SPN with the assumption that the firing times are exponentially distributed. It is roughly equivalent to our CTMC Markov example of two components with failure and repair, minus the tracking of which component failed first. It is important to note that with the SPN we can model the example with two places instead of four states with the use of tokens. The complexity of the Petri net does not depend on the number of components. We could add tokens to model additional components and modify the initial marking. In this case, adding a third token in the operational place would model three components. "[T]he dynamic behaviour of the PN can be mapped into a time-continuous homogeneous Markov chain with state space isomorphic to the reachability graph of the PN" [Bobbio 1990]. These Petri nets are often converted to an underlying CTMC for solving. Figure 18 shows the initial marking of both components as operational. Figure 19 shows one component failed and one operational. Figure 20 shows both components failed.

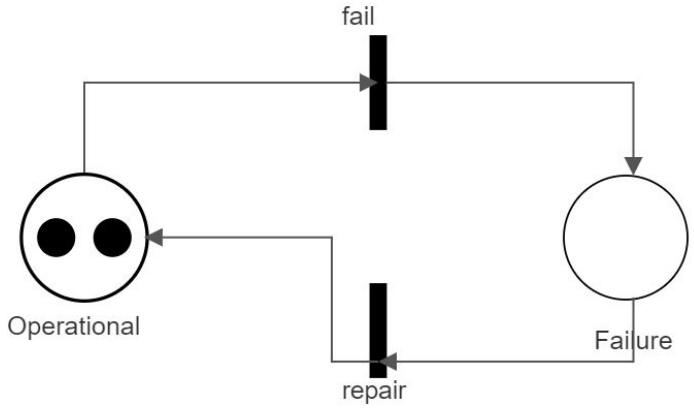


Figure 18: Two Components, Both Operational

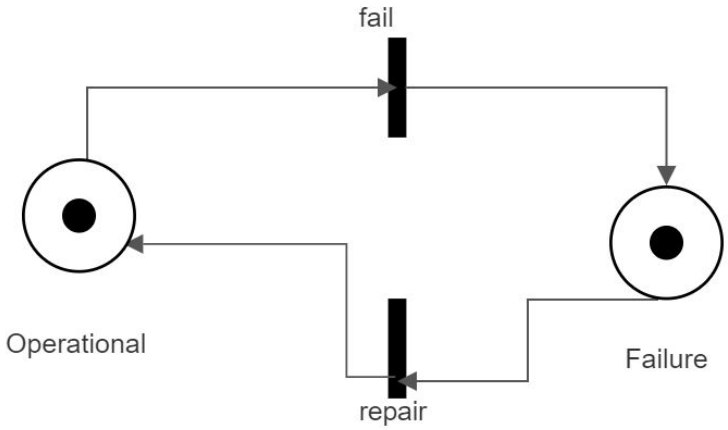


Figure 19: One Component Operational and One Failed

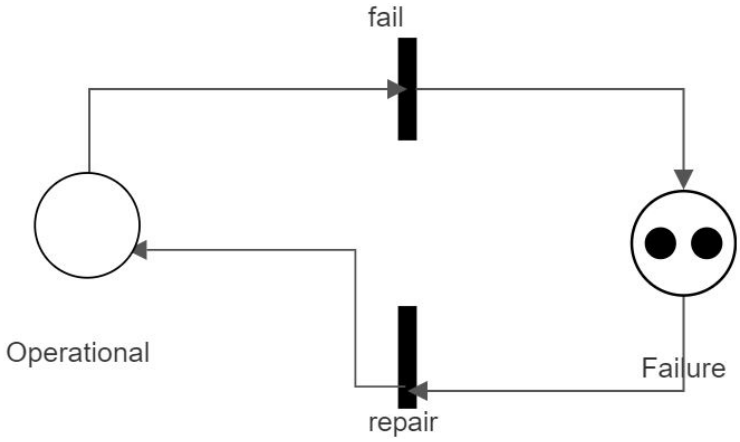


Figure 20: Two Components, Both Failed

There are extensions to SPNs that allow for more powerful modeling. We will summarize generalized SPNs and colored Petri nets.

A generalized SPN (GSPN) is used to model events that have extremely short times to occur, and it is useful to model them as instantaneous activities. The model is further extended to have

immediate transitions with zero firing times. All other transitions maintain their exponentially distributed firing times. Immediate timed transitions have a higher priority than any enabled time transition. If two or more immediate transitions are enabled, the firing is random with probabilities assigned to each.

Colored Petri nets (CPNs) enable powerful modeling. In a standard Petri net, tokens are indistinguishable and do not allow you to follow a token through the model. A colored Petri net can overcome this by giving each token an attribute or data associated with it. The places, arcs, and transitions can have enabling functions (i.e., Boolean expressions that must evaluate to true), which depend on the color of the token. This refinement again reduces the number of places needed to model complex systems and allows for more detail in the model to reflect the system under study more accurately.

We have now briefly summarized three extensions to Petri nets: SPNs, GSPNs, and CPNs. Note that the "resulting net with all these extensions can still be converted to a CTMC. However, there are tradeoffs with these extensions. Whereas these extensions make the task of modeling very simple and reduce the size of the net considerably, the complexity of understanding does increase" [Malhotra 1995].

6.3.3 Sample Tool Support for Robustness Modeling

All the information in Table 5 is taken directly from the web pages describing the various tools.

Table 5: Robustness Modeling and Analysis Tools

Tool Name	Modeling Techniques Supported	License
OSATE	Supports the SAE Standard Aerospace Recommended Practice (ARP) 4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. The processes and techniques of the ARP4761 standard addressed by the tool are the Functional Hazard Assessment, FTA, Failure Modes and Effects Analysis, and dependence diagrams, also referred to as RBDs.	Open source https://osate.org
PRISM	PRISM can build and analyze several types of probabilistic models: • discrete-time Markov chains (DTMCs) • continuous-time Markov chains (CTMCs) • Markov decision processes (MDPs) • probabilistic automata (PAs) • probabilistic timed automata (PTAs) Extensions of these models are available with costs and rewards.	Open source https://www.prismmodelchecker.org
BlockSim	ReliaSoft BlockSim provides a comprehensive platform for system reliability, availability, maintainability, and related analyses that allows you to model the most complex systems and processes using RBDs, FTA, or Markov diagrams.	Commercial https://www.reliasoft.com/products/blocksim-system-reliability-availability-maintainability-ram-analysis-software
SHARPE, by Duke University	SHARPE includes algorithms for analysis of fault trees, RBDs, acyclic series-parallel graphs, acyclic and cyclic Markov and semi-Markov models, GSPNs, and closed single- and multi-chain products from queueing networks.	Freely available for university usage; for company usage, there is contact information. https://sharpe.pratt.duke.edu

Tool Name	Modeling Techniques Supported	License
GRIF, by Total	The BFiab module is based on RBD modeling. The Tree module serves to model a system as a fault tree. The Markov module serves to model and compute small dynamic systems as Markov graphs. This module uses the analytical computation engine Albizia-Markov, which processes multi-phase systems. The Petri module serves to model large, complex industrial systems using SPNs with predicates and assertions. The BStok module is used to model complex systems based on stochastic block diagrams. The Reseda module serves to model reliability networks.	Commercial http://grif-workshop.com
CPN tools	Colored Petri nets	GNU General Public License (GPL), Version 2 http://cpntools.org/2018/01/15/simulation-replications
REALIST	Within the modeling capability of REALIST using extended SPNs (ESPNs), extended colored stochastic Petri nets (ECSPNs), or conjoint system models (CSMs), many aspects of a modern technical system can be captured. Reliability structure, system and component states Constant failure rates, dynamic changing failure behavior Failure propagation, failure dependencies, aging, several operative states	Contact is listed to obtain licensing information. https://www.ima.uni-stuttgart.de/en/research/reliability/realist/

7 Playbook for an Architecture Analysis of Robustness

This playbook outlines an approach to combine the checklists and questionnaires presented in the previous sections with information about mechanisms to analyze an architecture to validate the satisfaction of a robustness requirement. The playbook provides a process, illustrated with a running example, that will guide experts to perform architecture analysis in a more repeatable way.

The process has three phases and seven steps. The Preparation phase gathers the artifacts needed to perform the analysis and evaluation. The Orientation phase uses the information in the artifacts to understand the architecture approach to satisfying the quality attribute requirement. The process ends with the Evaluation phase, when the analysts apply their understanding of the requirement and architecture solution approaches to make judgments about those approaches. The phases and steps are summarized in Table 6.

Table 6: Phases and Steps to Analyze an Architecture

Phase	Step
Preparation	Step 1–Collect artifacts
Orientation	Step 2–Identify the mechanisms used to satisfy the requirement
	Step 3–Locate the mechanisms in the architecture
	Step 4–Identify derived decisions and special cases
Evaluation	Step 5–Assess requirement satisfaction
	Step 6–Assess impact on other quality attribute requirements
	Step 7–Assess the cost/benefit of the architecture approach

The analysts might identify missing artifacts during the Preparation phase and missing or incomplete information within those artifacts during the Orientation Phase. At the end of each step in the Preparation and Orientation phases, the analysts must decide whether there is sufficient information available to proceed with the process.

This process can be applied at almost any point in the development lifecycle. The quality of the architecture artifacts—breadth, depth, and completeness—will inform the type of analysis and evaluation performed in Step 5 and the degree of confidence in the results. Early in the development lifecycle, lower confidence may be acceptable, and the analyst can work with lower quality artifacts and simpler analyses, as suggested in Table 3. Later in the lifecycle, the analyst needs higher confidence and therefore higher quality artifacts and more and deeper analyses.

7.1 Step 1–Collect Artifacts

In this step, analysts collect the artifacts that they will need to perform the analysis. These include quality attribute requirements and architecture documentation.

The first artifact an analyst needs is the robustness requirement to validate. The requirement must be stated so that it is measurable, for example, as a quality attribute scenario as discussed above. Let's use variants of example scenarios 1 and 3 from Section 4.2, where we have specified the artifact as "Flight Management System." Let's call these Scenarios 5 and 6, respectively.

Scenario 5: Software does not report by deadline

Scenario Part	Value
Source	GPS is unavailable
Stimulus	GPS positioning is not reported by deadline
Artifact	Flight Management System
Environment	Normal operation
Response	The system detects the missed deadline, logs the error, and restarts reporting position from dead reckoning calculation
Response Measure	Detects fault within 2 ms and switches to dead reckoning within 200 ms to calculate position (until GPS becomes available)

Scenario 6: System resources reaching thresholds that predict overload

Scenario Part	Value
Source	Health monitor
Stimulus	Processor overheats and shuts down
Artifact	Flight Management System
Environment	Normal operation
Response	Failover to another CPU
Response Measure	Within 50 ms

Next, the analyst needs to consider the system's other quality attribute requirements. As noted above, architecture designs embody tradeoffs, and decisions that improve robustness may have a negative impact on the satisfaction of other important quality attribute requirements. In Step 6, the analyst will check that the architecture decisions made to satisfy this requirement do not adversely affect other quality attribute requirements, and more information about the complete set of quality attribute requirements means greater confidence in the results of that step.

Finally, the analyst needs architecture documentation. Early in the architecture development lifecycle, the documentation may be just a list of mechanisms mapped to quality attribute requirements, perhaps identifying tradeoffs. As the architecture is refined, partial models or structural diagrams become available, accompanied by information about key interfaces, behaviors and interactions, and rationale that provides a deeper link between the architecture decisions and quality attribute requirements. When the architecture development iteration is finished, then the documentation should include complete models or structural diagrams, along with specification of interfaces, behaviors and interactions, and rationale.

7.2 Step 2–Identify the Mechanisms Used to Satisfy the Requirement

To begin the Orientation phase, there are several places to look to identify mechanisms used in the architecture. If the architecture documentation includes a discussion of rationale, that can provide unambiguous identification of the mechanisms used to satisfy a quality attribute requirement. Other activities include looking at the structural and behavior diagrams or models and recognizing architecture patterns. Naming of architecture elements may indicate the mechanism being used. The analyst may also look at the file structure and naming of source code repositories, if they

exist, to locate mechanisms. The analyst may need to use all of these to identify the mechanism or mechanisms that are being used to satisfy the robustness requirement. Frequently, multiple mechanisms are needed to satisfy a requirement [Kazman 1997]. If the analyst has access to the architect(s), this is an excellent time to use the tactics-based questionnaires, as described in Section 6.1. In a short period of time, using these questionnaires, the analyst can enumerate all of the relevant mechanisms chosen (and not chosen).

Considering the example requirement above, "GPS positioning is not reported by deadline," let's assume that the project has not started development and that the architecture is largely conceptual in nature. *Health and performance monitoring* are specified as requirements, but the final selection of technologies has been delayed by a late discovery of an enterprise-wide common capability that the team has recently been made aware of. The architecture team has created documentation that includes a requirement to monitor key elements and resources along with a summary of rationale justifying other decisions relating to robustness. The rationale states that the system will use a *ping/echo* mechanism to detect failures in critical elements along with hardware and software redundancy to achieve the robustness requirements for the flight management system. The documentation also mentions the use of the Circuit Breaker pattern to reduce the likelihood of repeated or cascading failures, thus improving overall system robustness.

In a technical interchange meeting with architects, the analyst discovered that *voting* and *analytic redundancy* are being used for comparisons of important calculations for accuracy. *Voting* and *analytic redundancy* were not mentioned in their rationale for robustness, but preliminary discussions about a simple *ping/echo* has the architects thinking that this mechanism was more important in the satisfaction of their robustness requirements than they initially realized because it can help detect faulty states that a simple *ping/echo* would miss. Another mechanism is the use of *executable assertions* to help determine when and where a program is in a faulty state.

The analyst performs a quick check to decide if the referenced mechanisms are likely to contribute to satisfying the robustness requirement. In this case, all mechanisms that are enumerated above are known to positively impact robustness measures. The architects describe six mechanisms for robustness that seem to address the robustness requirements as they are currently understood.

In contrast, if the documented rationale (or the architect) stated that the architecture used a *record and playback* mechanism to achieve the above robustness requirements, this would raise a red flag since *record and playback* is usually associated with improving testability, but not robustness. The analyst might decide to pause the architecture analysis at this point and gather more information from the architect. The point of this quick check is not to analyze the mechanism or decision in detail but simply to assess whether the architecture analysis is on the right track, in terms of the available information and the mechanisms that have been chosen, before devoting more effort to a deeper analysis.

In some cases, the appropriateness of a mechanism is less clear. For example, the rationale for robustness design choices might specify that a *load balancing* mechanism is used. Load balancing can support robustness to some extent, but this mechanism by itself is usually insufficient since it only ensures that all resources are being used but ignores their health and utilization. In cases like this, the analyst should proceed carefully: The architect may have chosen an inappropriate mechanism or used a mechanism in an inappropriate way.

7.3 Step 3–Locate the Mechanisms in the Architecture

Following our example, the analyst needs to use the architecture documentation, an interview with the architect(s), or reverse-engineering to locate where these mechanisms are realized in the architecture. As seen in the tactics-based questionnaire, it is important to consider *how* a mechanism—tactic or pattern—is implemented.

Scenario 5 is concerned with a software element—the GPS—being unresponsive in the flight management system. The analyst may be able to look at the documentation and find a structural diagram sketch that includes where the health monitoring and redundancy are realized in the architecture. With this diagram in hand, finding instances of the health monitoring—for example, using a *ping/echo* mechanism for detecting faults—should not be difficult because it is a major abstraction in the system. The analyst should also be able to locate some information, perhaps a diagram, that provides insight into which critical elements have been replicated—that is, which elements have two or more instances, using one of the redundancy tactics, that allow the system to failover to other hardware or software. Another useful artifact for identifying critical elements is a fault tree analysis. Such an analysis might have been created, for example, from an AADL (Architecture Analysis & Design Language) model of the system [Delange 2014].

The analyst might next look for documentation relating to the *hardware and software redundancy* and *circuit breaker* mechanisms that describes how cascading failures are detected and prevented. In practice you may find this information in models, detailed designs, or higher level requirements statements. These mechanisms could be documented using, for example, sequence diagrams showing the path through the system when faults occur and accompanying annotations that detail the timing of each activation. In reality, it is often the case that important mechanisms are not specifically described in the architecture documentation but are discovered during interviews. In this case, during the technical interchange meeting the architects were describing the redundant analytics of both GPS and dead reckoning and using each as a check on the other for accuracy. The analysts note that this *voting* mechanism not only improves robustness, verifying that the elements are alive, but also can help determine whether the elements are operating according to their specifications.

Finally, the analyst must be able to conceptually integrate the mechanisms. The rationale for satisfying the requirement (e.g., GPS fails to respond) said that a redundancy pattern was used to allow failover to another alternative when a software element does not respond. This raises a question: Are all elements active, or is there a set of passive elements with only one active element at a time? This is an issue of how the system can *recover from faults* (and be resilient to changes in the environment), one of the categories of questions in the Architecture Analysis Checklist. One answer could be that the multiple redundant elements are active and the remaining elements only become active in a degraded mode. Another is a single active element and the health monitoring element switches to a passive element, making it the active element. However, the analyst finds that, in reality, the architecture is a hybrid where there are certain element types with only one active element and others where multiple elements are active.

Before finishing this step, the analysts should check that the mechanisms are being used in parts of the architecture that relate to the requirement that they are analyzing. To assess how well a mechanism contributes to satisfying a quality attribute requirement, it is not sufficient to stop after the sanity check in Step 2. That establishes only the presence of the mechanisms, not their

suitability or adequacy for meeting the response goal of the scenario being considered. The analysts must identify where and how the mechanism was instantiated in the architecture to assess whether it will have the desired effect. For example, they find a *ping/echo* mechanism, but it is used merely to determine that an element is alive and can respond as stated previously. This use of the mechanism, without self-diagnostics, health checks, or other tests of accuracy, is not likely to fully support the robustness required for the flight management system.

7.4 Step 4–Identify Derived Decisions and Special Cases

Most architecture mechanisms are not simple, one-size-fits-all constructs. The instantiation of a mechanism requires making a number of decisions, with some of those decisions involving choosing and instantiating other mechanisms. For instance, our example employs a redundancy type of pattern for hardware and software. One set of decisions about using that mechanism for software is concerned with details of the spare relating to state synchronization derived from the *mapping among architectural elements* category in the Architecture Analysis Checklist. This case includes several alternatives:

- Does the system employ active redundancy (hot spare)? In active redundancy, all nodes in a protection group receive and process identical inputs in parallel, allowing redundant spares to maintain synchronous state with the active element(s).
- If the system uses passive redundancy, how does it employ state resynchronization from active to standby elements?
- Does the system employ spares (cold spares) that are out of service until needed? How does the architect determine from, where, and how state will be copied or transferred to the cold spare when it is started?

If the architect decided to support *passive redundancy* (the second alternative above), then there is a set of subsequent derived decisions about how the synchronization is realized. The first is the frequency of updates for synchronization. Would there be a snapshot of the state sent at specific intervals, or would the passive elements receive messages of all state changes? The second derived decision would be where in the architecture is the responsibility to control routing to the active element so that all incoming inputs are processed by the newly activated element?

To assess how well a mechanism contributes to satisfying a quality attribute requirement, it is not sufficient to stop after the quick check in Step 2. The analyst must evaluate how the mechanism was instantiated, which usually involves tracing the decisions about the mechanism instantiation to the derived decisions and the selected alternatives that address them. And the analyst must endeavor to determine whether the mechanism, as envisioned and instantiated, is likely to actually meet the requirement.

As analysts identified the mechanisms in Step 3 above, they also started to identify derived decisions. For example, in the options outlined above, the analysts identified that the synchronization frequency will need to be tuned as the system evolves. Hence, they should pay attention to this parameter to determine the impact on the passive elements' ability to process inputs in a timely manner that will achieve the robustness objectives of the system.

The analyst's next derived decision might be "Do we use simple *ping/echo*, or do we have a more complex *ping/echo*—perhaps employing a monitoring pattern—that is in effect a small diagnostic

test?" This is a *detect faults* decision in the Architecture Analysis Checklist. For the robustness requirement that the analyst is validating, good answers to this question include the following:

- The system supports *self-tests* for correct operation.
- The system uses *voting* for critical elements to ensure they are not in a faulty state.
- The system has a well-defined fault tree analysis, and exception handling is defined for all known high-impact faults.

If these statements are all true, you would have reasonably high confidence that fault detection is adequate. (On the other hand, if the driving quality attribute requirement was, for example, to improve resource utilization rather than robustness, then the architect might have chosen to forgo the expense of self-testing, health monitoring, and voting. The removal of such mechanisms would limit the fault detection capability to determining if an element is alive and healthy, thereby impacting confidence in robustness for faulty states.)

The decision to use *voting* and *analytic redundancy* has other derived decisions. The first decision is which elements will be considered critical enough to require *voting* and *analytic redundancy* so that there will be a high level of confidence that features are working as intended. This decision impacts the system's ability to switch to a redundant positioning calculation within 200 ms. The software may also use a predictive model to monitor the health of certain less critical elements to ensure that the system is operating within nominal parameters, and the predictive model could make some instances of *voting* and *analytic redundancy* unnecessary.

Another derived decision related to voting is whether the *removal from service* tactic should be applied when erroneous inputs are detected (see the *prevent faults* category in the Tactics-Based Questionnaire, Section 6.1). When *voting* indicates a problem, then a corrective action such as removing the element from service can have a strong impact on robustness. This is especially true when a system is already degraded so that scarce resources can be applied to the critical and fully operational elements.

Finally, some mechanisms have special cases that warrant special attention. For example, the newly identified *health monitoring* option uses a rule-based throttling mechanism to manage policies for degradation when resources are compromised, maintaining the most critical system functions in the presence of element failures and dropping less critical functions. This can lead to increased complexity. A rule-based throttling mechanism may be overkill in instances where the criticality of the functions is relatively simple and straightforward. In other instances, where the criticality can change during operation or when capabilities change, a rule-based system can make the system easier to maintain due to its inherent flexibility.

In this example, analysts should pay attention to synchronization frequencies of the redundancy (i.e., active, passive, spare) strategy chosen so that resources are used efficiently and so that important measures such as failover time can be met. A fully synchronized active spare is already fully operational so messages or requests can be switched very quickly.

Another important concern is the self-tests that would enable the system to detect more faulty states in the system than a simple *ping/echo*. This is another decision that will impact the time it takes to detect faulty states that would be missed using a simple *ping/echo* mechanism.

7.5 Step 5–Assess Requirement Satisfaction

The analyst has completed preparation and orientation and begins the Evaluation phase. The analysis performed to assess whether the architecture satisfies the quality attribute requirement will depend on the nature of the requirement and the mechanism(s) being applied. For example, the analyst assesses a quality attribute requirement for robustness to the loss of a processor, and the mechanism used is a hardware redundancy pattern. The analysis should include checks for detection and failover (e.g., rerouting messages), which introduces new responsibilities. The analysis should also include the criticality of features and the envisioned degraded modes of operation.

Recall that the requirement in Scenario 5 is that a GPS is unavailable. Our measures are "Detect fault within 2 ms and switch to dead reckoning within 200 ms." The architecture mechanisms identified are the *ping-echo, voting and analytic redundancy, hardware and software redundancy,* and *executable assertions* tactics, and the Health Monitoring and Circuit Breaker patterns. In Step 4, the analyst identified several derived decisions that need to be considered in the analysis:

- Does the system use active, passive, or spares for redundancy?
- Does the system support simple *ping/echo* or more sophisticated *self-tests*?
- What is the frequency of *ping/echo*?
- Which capabilities will use voting, and will outliers use the *removal from service* tactic?
- Is there a rule-based system for managing policies for critical capabilities and degraded modes?

The analyst might begin by examining the realization of the Health Monitoring pattern and *ping/echo* tactic—and could ask the following questions. What type of analysis has been completed on the common capability recently discovered for Health Monitoring? How will *ping/echo* be implemented? The architect acknowledges there has not been time to complete analysis on the new health monitoring alternative (it should be noted that the architect plans to look at analysis of other systems that use the element), and a simple *ping/echo* response implementation at to-be-determined frequencies is documented in the architect's rationale. The analysts record the following issues:

- Issue 1: There has not been time to complete any significant analysis on the impact of switching from the previously selected element to the Health Monitoring common capability. This alternative was discovered during technical interchange meetings when constraints from the enterprise architecture were brought up and the alternative was first mentioned. Since health monitoring is a key capability for robustness, the impact of this change on the scenario response measures is not known. This is an important omission that could have been avoided through architectural governance practices specifically relating to knowledge management. The stakeholders have acknowledged that often key documents are out of date and not properly disseminated.
- Issue 2: The simple response version of *ping/echo* may miss important faulty states of critical elements and will be revisited by the architecture team.
- Issue 3: The *ping/echo* frequency may impact resource utilization and needs to be better defined.

This analysis thread is based on an observation that the team was given an alternative solution just a few days before the analysis and that they had only done superficial thinking on other important tactics (e.g., *ping/echo*).

Continuing our example, the analysts find that they were given conflicting answers when asking how the system would detect and then handle elements that return erroneous inputs when making calculations. While pulling this thread, some mentioned self-testing to detect erroneous inputs, which was not documented in the architecture. Others focused on the example of using voting to detect erroneous inputs, specifically multiple GPS positioning and multiple dead reckoning calculations. A related decision to support degraded modes disables dead reckoning calculations in the event of hardware failure, as dead reckoning is considered to be a lower criticality function. This, however, reduces the confidence in the *voting* mechanism since it reduces the number of voters. The analyst records the issues:

- Issue 4: Handling of elements that provide erroneous input has not been completely designed.
- Issue 5: Voting loses inputs in degraded modes since dead reckoning calculations are not considered critical until the GPS is unavailable, and calculations are disabled when hardware fails.
- Issue 6: No analysis was completed to determine how long it would take to restart dead reckoning calculation in degraded modes of operation when GPS becomes unavailable.

In this simple example, the analyst rapidly identified six issues where architecture decisions impact the ability to achieve the desired response measures in Scenario 5. Some of the issues, such as Issue 1 where a new alternative must be considered, should be carefully managed, and analysis of the alternatives must be done before implementation. Other issues, such as Issue 3, can be managed by experimentation. They plan to use similar systems built to experiment with *ping/echo* frequency and adjust the frequencies using the similar system's simulators to assess the impact on resource utilization.

7.6 Step 6–Assess Impact on Other Quality Attribute Requirements

Architecture decisions rarely affect just one quality attribute requirement. The tradeoffs inherent in design decisions mean that the mechanisms and decisions that the analyst assessed in Step 5 may be detrimental to the satisfaction of other quality attribute requirements.

Typical tradeoffs impact software performance (throughput or latency), testability, robustness, availability, maintainability, and usability. In Step 1–Collect Artifacts, the analysts collected other quality attribute requirements that were available at this point in the development lifecycle. Now, they will scan those and select the ones that might be impacted by the architecture mechanisms and decisions analyzed in Step 5–Assess Requirement Satisfaction. For example, there may be quality attribute requirements that cover concerns such as the following:

- Latency and resource utilization will be impacted by *ping/echo* using system resources (CPU and network bandwidth). Key decisions to be assessed revolved around defining the frequency of the *ping/echo* for health monitoring and the complexity of the ping message transferred.

- If the voting mechanism is implemented in hardware, then it is likely to be fast and highly reliable (particularly if it is simple, as most voters are), but at the cost of decreased modifiability.
- Accuracy and timeliness can be impacted when critical capabilities consume resources used by other capabilities. Reporting GPS data and defaulting to dead reckoning when GPS is unavailable would be critical to supplying accurate and timely position information to pilots. The flight management system supports degraded modes of operations where the dead reckoning calculation is halted, to preserve limited resources for other critical features, when system resource thresholds are reached or failures are present. The accuracy and timeliness of position information can become stale when the system is already in a degraded mode and then the GPS becomes unavailable. There would be a delay since the system is required to restart dead reckoning calculations until GPS data is once again available, resulting in longer than normal latency for refreshing position.
- Degraded modes of operation can be complex and may need to be revised as the properties of elements and hardware change during upgrades. An example change for degraded modes could occur when changing from a regular GPS signal to an encrypted GPS signal requiring additional CPU to process. Since the GPS now requires more resources, other resource allocations might need to change. Suppose GPS is more critical than altitude when the system is at 37,000 feet and the system experiences a processor failure requiring degraded modes of operation. In this case, the system could change the altitude refresh rate or the number of redundant altitude calculations to make more resources available for GPS now that it requires more resources for decrypting the GPS signal.
- Latency can be impacted by the selection of hardware, specifically restrictions on sources. For some use cases, choices are being constrained to a trusted foundry. These processors may take a long time to be fabricated and often have lesser processing capability than those commercially available, impacting overall latency.
- The use of the Circuit Breaker pattern that was found in the rationale for our maintainability analysis also supports robustness by isolating failures that occur in critical elements by returning error codes as soon as a faulty state is detected. This is helpful in a couple of ways. First, the system does not waste resources attempting calculations in a faulty element until it is restored to health. Second, the error codes returned provide detailed information to the Health Monitor so that the correct actions can be taken to restore the element to health. The *wrapper* used to implement the Circuit Breaker has a slight impact on latency.

In this step, the analyst assesses how the mechanisms and decisions support detecting faults and switching to other capabilities for the satisfaction of scenarios related to robustness. In addition, Step 6 focuses on other quality attributes and concerns. For each requirement, the analysis may be fast (e.g., ping frequency versus resource utilization) or more involved (e.g., assessing the restart time when rolling back during significant updates). In any case, the analyst should expect to find at least a couple of additional issues.

7.7 Step 7–Assess the Costs/Benefits of the Architecture Approach

In carrying out the steps leading up to this point, the analyst should have developed a good understanding of the essential challenges in satisfying the quality attribute requirement, the approaches taken by the architect (choice of mechanisms, instantiation of the mechanisms, and how derived concerns are addressed), and the tradeoffs embodied in the approaches taken.

Any architecture approach adds new elements, interactions, or responsibilities and makes the solution more complicated. Some approaches add new *types* of elements and interactions and in doing so may make the solution substantially more complex. There is a level of complexity needed to solve real-world problems—this is unavoidable. The final step is to judge whether the complexity (and hence additional cost) introduced by this architecture approach is necessary and appropriate. This is a cost/benefit analysis. In some cases, the answer will be clear: in our example, if the elements are simple and have only a few states, then a simple *ping/echo* would suffice. If the elements are complex with many possible faulty states, then the complexity of self-diagnostics would be worth the effort

Often the cost/benefit analysis is not clear, but probing the design space and the design decisions taken with this analytical perspective in mind is still worthwhile as it will catalyze important analysis questions.

Sidebar: Assessing Brittleness

In this report, we have primarily focused on aspects of robustness that are related to reliability and availability. However, as we discussed briefly in Section 2, an architecture can be robust along other dimensions as well: it can be robust with respect to future modifications, accommodating them in a way that requires minimal disruption to the architecture (and, ideally, minimal effort); it can be robust with respect to spikes in demand at run-time, allowing resources to scale up without requiring substantial human intervention; it can be robust with respect to a change in its environment, allowing the system to, for example, be ported to a different operating system or processor with low effort and little impact to the majority of the code base. And so forth.

In these senses, and in these contexts, an architecture can be designed to be malleable, not brittle, with respect to changes in its stimuli or environment. How would we analyze for such a quality? Given that this broader notion of robustness encompasses characteristics consistent with maintainability, integrability, and so forth, the specific mechanisms for dealing with them and the specific analyses to probe them are beyond the scope of this report. We can, however, suggest three broad strategies for gaining insight into the brittleness of an architecture. They involve the use of (1) growth and exploratory scenarios, (2) metrics, and (3) tactics-based questionnaires. We briefly discuss the use of each of these strategies.

Growth and exploratory scenarios are speculative; they speculate on possible future states of an architecture or possible future stresses on an architecture. They also assist in making a design decision when there are multiple options (in which case we prefer the option that is sufficient for the anticipated system but also can support growth with little impact). We have greater confidence that growth scenarios will come to pass because, by definition, these describe planned

or expected dimensions of system growth. Exploratory scenarios, as their name implies, are much less certain. But consideration of such scenarios allows an analyst or architect to gain insight into how easily they will be accommodated. If most or all growth and exploratory scenarios present significant challenges and risks, then the architecture is brittle, at least with respect to that set of scenarios.

Architectural metrics are meant to measure characteristics of an architecture that correlate with a desired outcome. For example, in the *Maintainability* report in this series we discussed the use of the Decoupling Level metric, which measures how well an architecture is decoupled into independent modules, as a way to gain insight into the maintainability of an architecture [Kazman 2020b]. Assuming that a metric has been empirically validated and that the metric measures something you actually care about, it can provide broad insight into an architectural characteristic. Metrics are, by their nature, the complement of scenarios. Scenarios give you deep but narrow insight. Metrics provide broad but shallow insight.

Finally, tactics-based questionnaires allow an analyst to gain insight into the architectural approaches taken and not taken. This insight can be gained in a short time (typically around one hour per quality attribute analyzed) with a modest expenditure of effort. While this insight will not result in precise analyses, it will reveal where the architectural effort has been placed with respect to this quality attribute and any glaring omissions. This knowledge can help an analyst assess the likelihood that an architecture is brittle with respect to the quality under consideration.

Given that all three of the above techniques are limited, given that they are relatively inexpensive, and given that they provide different kinds of insights, we recommend using all of them. Together they can give insight into the likelihood of brittleness with respect to qualities of interest.

8 Summary

In this report, we have defined the quality attribute of robustness and focused on analyzing the challenges of achieving robust systems, and analyzing for robustness, based on an understanding of architectural mechanisms and their characteristics.

We have provided a set of sample scenarios for robustness and, from these and other examples, inferred a general scenario. This general scenario can be used as an elicitation device and also helps with analysis as it delineates the response measures that stakeholders will care about when they consider this quality attribute. We have also described the architectural mechanisms—tactics and patterns—for robustness. These mechanisms are useful in both design—to give a software architect a vocabulary of design primitives from which to choose—and in analysis, so an analyst can understand the design decisions made, or not made, their rationale, and their potential consequences.

To address the needs of analysts, we have described a set of analytical tools and discussed the artifacts upon which each of these analyses depends and the stage of the software development lifecycle in which each of these analyses could be employed.

In addition, we have provided a "playbook" for applying an architecture analysis for robustness. This playbook combines the checklists and questionnaires with information about architectural mechanisms to analyze an architecture to validate the satisfaction of a robustness requirement.

Finally, it must be emphasized that, in this report, we have focused on an aspect of robustness that dealt with an architecture's response to anticipated faults and failures. While this dimension is clearly important to system success, it is not the only way in which an architecture can be robust. To get insight into the other aspects of robustness that we mentioned in Section 2 of this report requires us to think about other quality attributes and their tradeoffs. We can use exploratory scenarios, metrics, and tactics-based questions, as was discussed in Sections 6 and 7, to get insight into such tradeoffs.

9 Further Reading

The original tactics on which much of this report are based were first described in a paper by Scott and Kazman [Scott 2009] and later elaborated in the book *Software Architecture in Practice* [Bass 2012].

For a discussion of the aspects of architectural robustness that are related to maintainability, such as being robust with respect to unknown changes in the future, see the *Maintainability* report in this series [Kazman 2020b].

More information on Petri nets can be found in work by Malhotra and Trivedi [Malhotra 1995]. A general introduction to Markov chains can be found in the book *Markov Chains* [Gagniuc 2017]. Fault modeling in AADL has been described in the work of Delange and colleagues [Delange 2014].

Bibliography

URLs are valid as of the publication date of this document.

[Avizienis 1985]
Avizienis, A. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*. Volume Se-I 1. Issue 12. December 1985. Pages 1491–1501.

[Avizienis 2001]
Avizienis, A.; Laprie, J.; & Randall, B. *Fundamental Concepts of Dependability*. Tech. Rep. 1145. University of Newcastle. 2001.

[Baker 2008]
Baker, J.; Schubert, M.; & Faber, M. On the Assessment of Robustness. *Structural Safety*. Volume 30. Issue 3. Pages 253–267. May 2008.

[Bass 2012]
Bass, L.; Clements, P; & Kazman, R. *Software Architecture in Practice*, 3rd ed. Addison-Wesley. 2012.

[Bellomo 2015]
Bellomo, S.; Gorton, I.; & Kazman, R. Insights from 15 Years of ATAM Data: Towards Agile Architecture. *IEEE Software*. Volume 32. Number 5. 2015. Pages 38–45.

[Binder 2000]
Binder, R. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley. 2000.

[BKCASE 2018]
Body of Knowledge and Curriculum to Advance Systems Engineering. System Requirements. In *Guide to the Systems Engineering Body of Knowledge (SEBoK)*. 2018. https://www.sebokwiki.org/wiki/System_Requirements

[Bobbio 1990]
Bobbio, A. System Modelling with Petri Nets. In *System Reliability Assessment*. Colombo, A. & Saiz de Bustamante, A. (eds.). Kluwer. Pages 103–144. 1990.

[Bodson 1994]
Bodson, M.; Lehoczky, J.; Rajkumar, R.; Sha, L.; & Stephan, J. Analytic Redundancy for Software Fault-Tolerance in Hard Real-Time Systems. In *Foundations of Dependable Computing*. Koob, G. M. & Lau, C. G. (eds.). Kluwer. Pages 183–212. 1994.

[Bolch 2006]
Bolch, G.; Greiner, S.; de Meer, H.; & Trivedi, K. S. *Queuing Networks and Markov Chains*, 2nd ed. Wiley. 2006.

[Boyd 1998]
Boyd, M. & Lau, S. *An Introduction to Markov Modeling: Concepts and Uses.* NASA. 1998. https://ntrs.nasa.gov/search.jsp?R=20020050518

[Cepin 2011]
Cepin, M. *Assessment of Power System Reliability*. Springer. 2011.

[Cervantes 2016]
Cervantes, H. & Kazman, R. *Designing Software Architectures: A Practical Approach*. Addison-Wesley. 2016.

[Clausthal n.d.]
Technische Universität Clausthal Institute of Mathematics. Simulation of a Homogeneous Markov Chain (in Discrete Time). https://www.mathematik.tu-clausthal.de/en/mathematics-interactive/simulation/markov-chain-discrete/

[Clements 2010]
Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Merson, P.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Views and Beyond*, 2nd ed. Addison-Wesley. 2010.

[Delange 2014]
Delange, Julien; Feiler, Peter; Gluch, David; & Hudak, John. *AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment.* CMU/SEI-2014-TR-020. Software Engineering Institute, Carnegie Mellon University. 2014. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=311884

[DIB 2019]
Defense Innovation Board. *Software Is Never Done: Refactoring the Acquisition Code for Competitive Advantage*. DoD. 2019. https://media.defense.gov/2019/Apr/30/2002124828/-1/-1/0/SOFTWAREISNEVERDONE_REFACTORINGTHEACQUISITIONCODEFORCOMPETITIVEADVANTAGE_FINAL.SWAP.REPORT.PDF

[Gagniuc 2017]
Gagniuc, P. *Markov Chains: From Theory to Implementation and Experimentation*. Wiley. 2017.

[ISO/IEC 2011]
International Organization for Standardization and International Electrotechnical Commission. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. ISO/IEC 25010:2011. ISO. 2011.

[Kazman 1994]
Kazman, R.; Abowd, G.; Bass, L.; & Webb, M. SAAM: A Method for Analyzing the Properties of Software Architectures. In *Proceedings of the 16th International Conference on Software Engineering*. Sorrento, Italy. ACM. 1994. Pages 81–90.

[Kazman 1997]
Kazman, R.; Clements, P.; Bass, L.; & Abowd, G. Classifying Architectural Elements as a Foundation for Mechanism Matching. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*. Washington, DC. IEEE Computer Society. 1997. Pages 14–17.

[Kazman 2020a]
Kazman, Rick; Bianco, Philip; Ivers, James; & Klein, John. *Integrability.* CMU/SEI-2020-TR-001. Software Engineering Institute, Carnegie Mellon University. 2020. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=637375

[Kazman 2020b]
Kazman, Rick; Bianco, Philip; Ivers, James; & Klein, John. *Maintainability.* CMU/SEI-2020-TR-006. Software Engineering Institute, Carnegie Mellon University. 2020. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=650480

[Klein 2015]
Klein, J. & Gorton, I. Design Assistant for NoSQL Technology Selection. In *Proceedings of the First International Workshop on Future of Software Architecture Design Assistants*. ACM. 2015. Pages 7–12.

[Malhotra 1995]
Malhotra, M. & Trivedi, K. S. Dependability Modeling Using Petri Nets. *IEEE Transactions on Reliability*. Volume 44. Issue 3. Pages 428–440. September 1995.

[NRC 2015]
National Research Council. *Reliability Growth: Enhancing Defense System Reliability*. National Academies Press, 2015. https://doi.org/10.17226/18987

[Nygard 2017]
Nygard, Michael. *Release It! Design and Deploy Production-Ready Software*. Pragmatic Programmers. 2017.

[ODASD 2017]
Office of the Deputy Assistant Secretary of Defense. Initiatives: Modular Open Systems Approach. 2017. https://www.dsp.dla.mil/Programs/MOSA/

[Padilla 2019]
Padilla, M.; Davis, J.; & Jacobs, W. Comprehensive Architecture Strategy (CAS). The Open Group. September 2019. https://www.opengroup.us/face/documents.php?action=show&dcat=87&gdid=21082

[Reliability Analytics 2010–2020]
Reliability Analytics Toolkit. Reliability Analytics. 2010–2020. https://reliabilityanalyticstoolkit.appspot.com

[Scott 2009]
Scott, James & Kazman, Rick. *Realizing and Refining Architectural Tactics: Availability.* CMU/SEI-2009-TR-006. Software Engineering Institute, Carnegie Mellon University. 2009. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9087

[Sha 1998]
Sha, L.; Goodenough, J.; & Pollak, B. Simplex Architecture: Meeting the Challenges of Using COTS in High-Reliability Systems. *CrossTalk*. Pages 7–10. April 1998.

[Shahrokni 2013]
Shahrokni, A. & Feldt, R. A Systematic Review of Software Robustness. *Information and Software Technology*. Volume. 55. Number 1. Pages 1–17. January 2013.

[Sussman 2007]
Sussman, G. J. Building Robust Systems: An Essay. 2007. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.113.1324

[SWEBOK 2014]
Bourque, P. & Fairley, R. E. (eds.). *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society. 2014. https://www.computer.org/education/bodies-of-knowledge/software-engineering

[Trivedi 2016]
Trivedi, K. S. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, 2nd ed. Wiley. 2016.

[Trivedi 2017]
Trivedi, K. S. & Bobbio, A. *Reliability and Availability: Modeling, Analysis, Applications*. Cambridge University Press. 2017.

[Vesely 2002]
Vesely, W.; Dugan, J.; Fragola, J.; Minarick, J.; & Railsback, J. *Fault Tree Handbook with Aerospace Applications*. NASA. 2002.
http://www.mwftr.com/CS2/Fault%20Tree%20Handbook_NASA.pdf

[Wang 2001]
Wang, C.; Sklar, D.; & Johnson, D. Forward Error-Correction Coding. *Crosslink*. Volume 3. Issue 1. Pages 26–29. 2001.

[Yang 2007]
Yang, G. *Life Cycle Reliability Engineering*. Wiley. 2007.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	March 2022	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Robustness	FA8702-15-D-0002

6. AUTHOR(S)

Rick Kazman, Phil Bianco, Sebastián Echeverría, and James Ivers

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2022-TR-004

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report summarizes how to systematically analyze a software architecture with respect to a quality attribute requirement for robustness. The report introduces the quality attribute of robustness and common forms of robustness requirements for software architecture. It provides a set of definitions, foundational concepts, and a framework for reasoning about robustness and the satisfaction of robustness requirements by an architecture and by a system that realizes the architecture. It describes a set of architectural mechanisms—patterns and tactics—that are commonly used to satisfy robustness requirements. It also provides a set of steps that an analyst can use to determine whether an architecture documentation package provides enough information to support analysis and, if so, to determine whether the architectural decisions made contain serious risks relative to robustness requirements. An analyst can use these steps to determine whether those requirements, represented as a set of scenarios, have been sufficiently well specified to support the needs of analysis. The reasoning around this quality attribute should allow an analyst, armed with appropriate architectural documentation, to assess the robustness risks inherent in today's architectural decisions, in light of tomorrow's anticipated needs.

14. SUBJECT TERMS	15. NUMBER OF PAGES
architecture analysis, robustness, quality attributes, quality attribute requirements, software architecture	84

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102