A Comprehensive Exploration of Challenges in Architecture-Based Reliability Estimation

Ivo Krka, George Edwards, Leslie Cheung, Leana Golubchik, and Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA
{krka,gedwards,lccheung,leana,neno}@usc.edu

Abstract. Architecture-based reliability estimation is challenging: modern software is complex with numerous factors affecting a system's reliability. In this article, we address three core challenges for architecture-based estimation of a system's reliability: (1) defining an appropriate failure model based on characteristics of the system being analyzed, (2) dealing with uncertainties of the reliability-related parameters, due to the lack of system implementation, and (3) overcoming the barriers of complexity and scale inherent in modern software. For each challenge, we identify the essential elements of the problem space, outline promising solutions, and illustrate the problems and solutions using a robotics case study. First, we show how a failure model can be derived from the system requirements and architecture. Second, we suggest how information sources available during architectural design can be combined to mitigate model parameter uncertainties. Third, we foresee hierarchical techniques as a promising way of improving the computational tractability of reliability models.

1 Introduction

In modern software systems, ensuring the satisfaction of non-functional quality requirements, such as security, performance, reliability and safety, is often as critical as the satisfaction of functional requirements. Due to the continuing rise in the scale, complexity, and concurrency of software systems, creating systems that satisfy numerous quality requirements is difficult. Reasoning about system quality should not be delayed until implementation is underway because the principal design decisions that most heavily impact system quality — the system's software architecture [39] — are, by this stage, already deeply incorporated into every aspect of the system. Further, changing fundamental design decisions after implementation, testing, integration, or deployment is prohibitively expensive [37]. To be cost-effective, analysis of system quality attributes must be performed using the architectural artifacts available early in the development. In this paper, we focus on architecture-based estimation of an important quality attribute — reliability, which is defined as continuity of correct service [1]. Reliability is an important aspect of a software system, as unreliable systems may cause inconvenience, damage to business reputation and revenue, and even loss of human lives.

Software reliability has traditionally been addressed during testing and operation [22, 27]. Existing implementation-level approaches, such as software reliability growth models [9, 16, 21, 28], build models of failure behavior by collecting the run-time information about the system's operation. As discussed above, it is more desirable to start assessing software reliability early in order to reduce the costs of fault removal.

A high-level view of the process and the artifacts necessary for estimating system reliability at the architectural level is depicted in Figure 1 (the processes are depicted in blue, while the artifacts are in gray). The available design time artifacts, such as system requirements and architectural specifications (step 1), are leveraged to obtain a failure model capturing the characteristics of erroneous behavior — i.e., the specification of how system faults and errors occur, how they propagate, and how they result in service failures (step 2). It is important to note that the failure model of the system should ideally be constructed from the individual failure models of system elements. The system failure model is then combined with architectural specification and reliability parameter estimates (step 3). to finally produce the system reliability estimates (step 4).

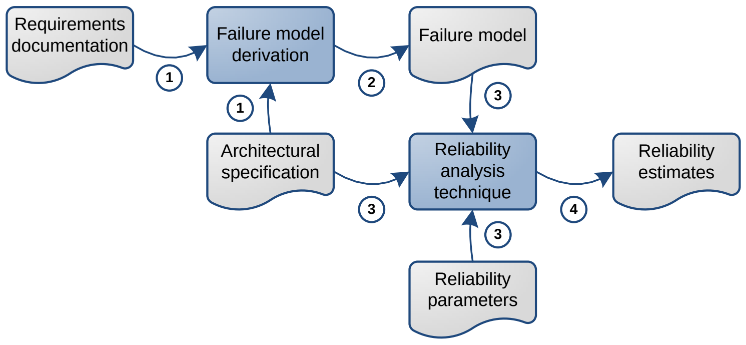

Fig. 1. Architecture-based reliability estimation process

Although generally useful, existing architecture-based approaches suffer from several shortcomings, which have been noted by several surveys of this area [10, 14, 15]. More specifically, existing approaches have tended to oversimplify or ignore one or more important challenges of the above process when producing reliability predictions:

1. Existing approaches often make particular assumptions about a system's failure model, which includes restrictions on supported failure instances, limited support for defining varying error combinations that cause failures, and so on. However, different types of failure models are suitable for different types of systems, and no single form of a failure model is ideal for all systems.
2. Existing approaches generally do not provide mechanisms for obtaining good estimates of architectural model and failure model parameters required for accurate reliability analysis. Estimating model parameters, such as the operational profile [26], is problematic due to the unavailability of the system implementation.
3. Existing approaches do not scale to large, complex software systems because they create computationally intractable analysis models or do not manage concurrency effectively.

To make architecture-based reliability estimation truly meaningful, useful, and usable, these challenges must be overcome and not ignored. In other words, a software architect will only consider the predicted reliability effects of prospective design decisions if reliability estimates realistically reflect the eventual characteristics of the implemented system. Thus, architecture-based reliability estimation techniques should offer:

1. the flexibility to choose or define an appropriate failure model, both at the level of constituent system elements and the system as a whole, based on the characteristics and goals of the system under evaluation,
2. processes for obtaining model parameters from the imperfect information sources available during early system development phases, and
3. analysis and simulation that are manageable and tractable even for highly complex software systems.

Our goal in this article is to encourage future improvement of existing techniques for architecture-based reliability estimation, and to inspire the creation of new techniques, by clearly scoping the central challenges of architecture-based reliability estimation and proposing solution approaches that have shown promise in a research setting. The work in this article is an extension of our previous study of the problem space of, the challenges in, and the strategies for architecture-based reliability estimation [18].

Every architecture-based reliability estimation technique relies on a concrete definition of a failure. A failure is an event that occurs when the delivered service deviates from the correct service [1]. Thus, different systems may have very different failure instances, depending on their corresponding requirements. As stated earlier however, existing architecture-based reliability estimation approaches do not allow flexibility in modeling the exact types of failure instances. For example, some reliability models assume that the failure rate is a function of the number of faults (bugs) present in the software [9, 16, 21, 28], while other reliability models define different failure instances as Boolean expressions on internal errors [32]. The repercussions of a failure may also be complex, with different severity, and duration characteristics in different contexts, even for a single system. Although reliability estimation is not directly concerned with consequences of a failure, this information is typically needed to determine which failures to include in the reliability analysis. Therefore, a goal of architecture-based reliability estimation should be to provide the architect with a multidimensional description of reliability for different types of failure instances, viewpoints, parameter combinations, and so on. We further elaborate on the challenge of defining failure models in Section 3. Additionally, we provide guidelines for derivation of the different facets of a failure model. In doing so, we intend to help both the developers and prospective users of new architecture-based reliability estimation techniques.

Evaluation of system quality when an implementation is unavailable is hampered by the many uncertainties related to system properties. In architecture-based reliability estimation, important reliability model parameters, such as the frequency of inputs, are not usually known [2]. Furthermore, many existing approaches often wrongly assume that the required information is available and precise. In Section 4, we provide an extensive discussion of various reliability parameters which, in general, cannot be precisely determined due to the implementation unavailability. Dealing with these uncertainties requires estimating or deriving unknown information from indirect or non-ideal sources. Also, it is not sufficient to simply provide a single reliability figure, as many existing approaches do; since parameter estimates are likely to be imprecise, ranges of values corresponding to different parameter estimates should be provided. Additionally, the quality of an information source may vary in different situations. In Section 4, we identify different classes of information sources that are potentially available during architectural design and elaborate on their applicability and usefulness in different development contexts.

Existing techniques for early reliability estimation have strong formal and theoretical foundations. For example, Reussner [30] models a software system as a set of interacting components whose connections represent flow of control, while Rodrigues [31] uses Message Sequence Charts to derive system reliability models, and Cheung [2] models the reliability of a component based on a representation of its internal behavior. However, the size and concurrency of the systems existing approaches are able to analyze are not comparable with those found in many modern systems, as discussed in Section 5. In order to apply an architecture-based reliability estimation technique to a real-world software system, the technique must be able to cope with the system's scale, complexity, and concurrency. Our experience suggests that hierarchical techniques are an effective way of dealing with the challenge of complexity in modern software. In Section 5, we discuss how the principles of hierarchy commonly used in software system modeling can be utilized in reliability analysis in order to improve computational tractability. We also discuss the possibility of additional state space reduction by applying well-known aggregation and truncation techniques [38].

The paper is organized as follows. In Section 2, we describe the types of architecture and design models that may be leveraged in architecture-based reliability assessment, and introduce an example mobile robotics application that is used throughout the paper to illustrate important concepts. As outlined earlier, the main challenges in architecture-based reliability estimation, along with our proposed solutions to them, are described in Sections 3-5. In Section 6, we conclude the paper.

2 Background

2.1 Architectural Models for Reliability Estimation

To evaluate the reliability of a software system during its design, an architectural model of the system is required. In this section, we focus on the design-time architectural artifacts that may be required when reasoning about a system's reliability.

Most estimation techniques require at least a basic architectural model that captures how functionality is organized into components and how components interact with each other and their execution environment. System components should specify their provided and required interfaces in terms of offered services, the corresponding data inputs and outputs, and the logical connections between component interfaces over which data and/or control messages are exchanged. For example, Cheung [3] requires modelers to specify the data and control transfers that may take place among components as well as the probabilities of those transfers occurring.

Architecture-based reliability estimation techniques commonly require additional information beyond basic component structure and interaction to be captured in the architectural model. Different reliability estimation techniques require different types of information to be captured and, consequently, different modeling notations and architecture description languages (ADLs) are suited for different estimation techniques [25]. The types of additional information some estimation techniques require include (1) non-functional properties, such as system performance (e.g., [11, 43]) [1], (2) hardware-software interactions, such as component deployment (e.g., [5, 6, 40]), (3) internal component behavior, such as state-based models (e.g., [2, 14, 31]), and (4) platform characteristics, such as middleware services specifications (e.g., [5, 6, 43]). Frequently, the different aspects of a system are captured in separate views or perspectives. Modeling a system from multiple perspectives allows each one to focus on a narrow set of related concerns.

A specification of the system's hardware-software interactions is needed to incorporate the reliability effects of the underlying hardware platform into reliability estimates. The deployment architecture specifies the assignment of software components to hardware hosts. Related to the deployment architecture are detailed specifications of the individual hosts and network links. As an example, Cortellessa [5] utilizes UML deployment diagrams to incorporate component deployment information and the reliability of network links into analysis. Similarly, architecture-based reliability analysis can benefit from detailed models of the underlying middleware and operating system. For example, Das [6] utilizes a layered approach to specify properties of the underlying operating system or middleware. To reason about the influence of software connectors on reliability, the information about their types and (preferably) models of their behavior (i.e., their communication protocols) should be available. Since software connectors are often reused as part of the middleware platform on top of which the application is implemented, the aforementioned middleware specification may already contain such information. Finally, reliability analysis can be enhanced by documentation of component services' external and/or internal behavior, as in Reussner [30].

When a system is too complex to capture its complete software architecture, it can be modeled using a specification of its architectural style (the Internet is an example of such a system [8]). In these cases, the architectural style forms the basis for the reliability analysis (e.g., Wang [42] models reliabilities of different architectural styles). The obtained style-based results should then be refined for the specific system under consideration. Modern systems that employ mobility, adaptation, fault-tolerance, replication, caching, and other related mechanisms have highly dynamic architectures. Thus, it is not desirable, or perhaps even feasible, to generate every possible configuration and analyze its reliability properties. Instead, to facilitate reliability analysis at the architectural level, the behavior of these mechanisms should be captured in an architectural specification. The analysis can subsequently be performed by utilizing the information about these mechanisms. Lastly, when a system interacts closely with the physical environment (e.g., sensor networks, robotics), reliability analysis can be enhanced by including probable properties of the environment. For example, the reliability of a sensor node may be affected by the temperature (e.g., reliability is lower under extreme temperature). Similarly, knowing the physical environment in which a sensor network

[1] At the architectural level, estimating other non-functional properties, such as performance, can be as challenging as reliability estimation.

application is deployed (e.g., undersea or in a desert) can improve estimates of sensor node reliability.

2.2 Application Scenario

In this section, we introduce the functionality and software architecture of *RoboArch*, a mobile robotics application, which is leveraged in the rest of the paper to illustrate the most important concepts. Software applications in the domain of mobile robotics can be fairly complex, with numerous factors affecting application behavior, including the non-deterministic and rapidly changing physical environment. Additionally, reliability is a very important system quality attribute in mobile robotics. Thus, we found *RoboArch* appropriate for the purposes of discussion in this article.

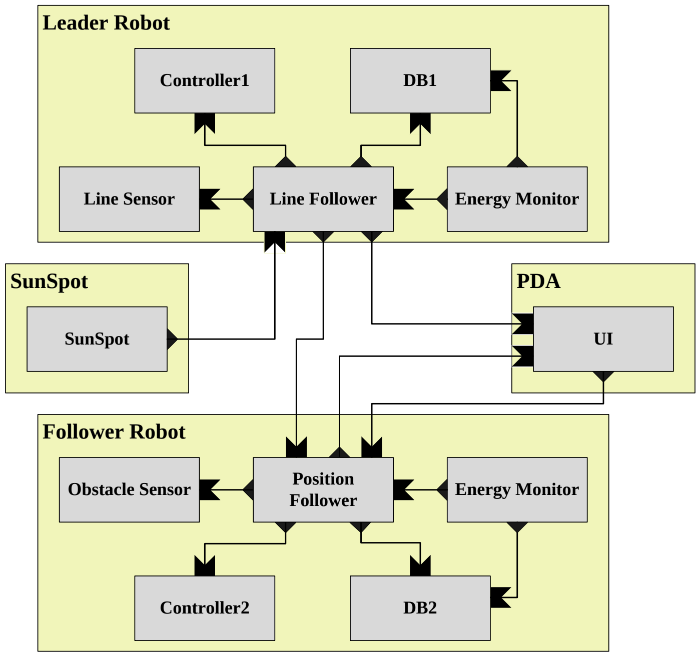

Fig. 2. Configuration and deployment of the *RoboArch* system

RoboArch is a system in which multiple robots create a convoy by following one another on a predefined path. A high-level configuration and deployment of a two robot *RoboArch* system is depicted in Figure 2. The *Leader Robot* is traveling along a predefined path, and is followed by the *Follower Robot*. The *Leader*'s main component is the *LineFollower* which is responsible for (1) reading the *Sensor* data, (2) controlling the robot's movement via the *Controller*, and (3) sending its movement data to the *PositionFollower* as well as to the human supervisor operating on a *PDA*. The *Position-Follower* component provides similar functionality and moves in accordance with the data received from the *LineFollower*. The *Sensors* provide a range of information about the tracking of the predefined path and the surrounding physical obstacles.

More advanced adaptive behavior in *RoboArch* is supported through additional components. Using the *SunSpot* component, the system supervisor can remotely direct a lost *Leader* towards the predefined path. The *PDA* is used as a display of the robots' positions, and a facility for sending the *Follower* its correct position in the event that this information becomes corrupted due to imprecise sensor measurements. The *EnergyMonitor* adapts the robots' behavior in accordance with their energy consumption and the remaining battery power. If a robot's battery power is low, or the robot has recharged, the *EnergyMonitor* informs the *LineFollower* (or *PositionFollower*) to increase (or decrease) the sensor reading and network packet sending rates.

Although *RoboArch*'s software connectors are uniformly depicted as lines in Figure 2, their respective types differ. For example, the connectors between different hosts are event-based, while the connectors between the *LineFollower* and the *LineSensor* are procedure-call connectors. Additionally, the connector between the *LineFollower* and the *DB* database component is a data access connector.

In principle, *RoboArch* might appear to be a functionally simple system, but it contains a number of components, allows different user inputs, depends on a non-deterministic physical environment, implements various communication mechanisms, adaptation policies, and so on. This all makes *RoboArch* fairly complex from an architectural perspective. Moreover, *RoboArch* is designed to be easily extensible. For example, the system can be expanded to more than two robots by connecting new *Followers* to the already existing *Followers*. The architecture depicted in Figure 2 can be further modified to capture an architectural style for systems with arbitrary numbers of robots. For clarity purposes, our description of the *RoboArch* system is focused on the main application functionality. In reality, *RoboArch* is further enhanced with capabilities for fault-tolerance, adaptation, dynamic redeployment, and equipped with additional hardware hosts such as docking stations.

3 Challenge 1: Defining a System's Failure Model

We start this section with a key insight into the characteristics of software failures. The multi-dimensional nature of failures makes determining an appropriate failure model very challenging for a software architect with limited reliability expertise (Section 3.1). In Section 3.2, we delineate our solution to this challenge, which leverages requirements and architectural specifications. In Section 3.3, we depict the outlined concepts in the context of the *RoboArch* system.

3.1 Multiple Dimensions of Failure Models

In modern software systems, failure and recovery behaviors tend to be complex, with a plethora of potential failure causes, and, in turn, a plethora of possible fault and error repercussions. Thus, it is necessary to explore the different facets constituting a system's failure model in order to estimate reliability in a satisfactory manner. Although researchers have classified important concepts related to erroneous behavior (e.g., Avizienis et al. [1]), architecture-based reliability approaches have generally failed to consider all the nuances of system faults and errors, and have consequently arrived at oversimplified failure models. In this section, we revisit and refine the characterization of failure and recovery related behavior widely used in dependability literature [1]. We approach this problem from a purely software architectural perspective, with a desire to assist software architects and designers of new architecture-based reliability estimation techniques. Consequently, we do not touch upon some aspects of erroneous and recovery behavior present in the existing dependability framework that are not as relevant from an architectural perspective (e.g., development faults, fault prevention, and so on).

The problem space of architecture-based reliability estimation is based on the specific definition of reliability being considered. Generally, system **reliability** is defined as the continuity of correct service [1]. Although correct, this definition is not refined enough from a software architect's perspective, as it implicitly assumes that concrete definitions of correct service, and, in turn, possible failure instances exist or are easily obtainable. In the following section, we illustrate the different ways of deriving the needed information.

As discussed in Section 2, a **failure** is defined as any deviation of the delivered service from the correct service [1]. Furthermore, the different concepts related to a failure have been detailed, and we distinguish between a **fault** (often referred to as defect in software architecture community), **error**, and **failure** [1]. A fault is the cause of an error (e.g., an incorrectly defined interaction protocol). When a fault is activated (e.g., the incorrect sequence of interactions takes place), it results in an error by causing the current state of the system to differ from the correct state (i.e., an internal system state which can result in a subsequent service failure). An error may cause a failure, which is when the delivered service is different from the correct service (e.g., processing of a user's request requires invocation of an erroneous component, and the service cannot be delivered).

It is important to note that a failure of a system element (e.g., component, connector, etc.) is, from the system's viewpoint, an error. Thus, the problem of defining a failure becomes the problem of determining which error combinations and their propagation sequences imply a failure. It is not clear, however, how a software architect with limited reliability/dependability expertise would define a failure in terms of errors of individual system elements and arrive at a complete failure model for an arbitrary system. Some existing approaches (e.g., [33]) can help an architect generate a comprehensive failure model, but these approaches start with a basic failure model and they do not detail how the basic model is obtained.

A system's failure model thus depends on a very precise definition of possible **failure instances** (we are referring to the different combinations of errors that cause failures). The existing architecture-based reliability estimation approaches offer little or no flexibility in defining particular failure instances, which negatively affects their applicability to arbitrary software systems. For example, Reussner [30] defines a failure instance as an error of any particular service of system components (i.e., they do not distinguish between external and internal services which is crucial in the reliability definition), while many others (e.g., [5, 31, 43]) define it as any error that occurs within any individual component. Roshandel [32], on the other hand, allows failure instances to be defined as Boolean combinations of individual component errors. Moreover, the definition of a failure instance should not be limited to application components, but extended to errors of any system element, such as software connectors, hardware hosts, and so on. Furthermore, it is important to reiterate that failures can be both content and timing failures [1], which is often overlooked by the existing approaches.

A system's failure model largely relies on the notion of **failure consequence**. Obviously, not all failures that can occur in a given system have the same weight, meaning that some failures are failures of a critical service, while the impact of other failures is almost negligible. Additionally, different stakeholders view the system from different perspectives, resulting in varying perspective-based failure models. None of the existing architecture-based approaches, except Goseva-Popstojanova [13], explicitly consider failure consequence in their analyses. Goseva-Popstojanova [13] illustrates how to compute the overall failure consequence distribution of the system's components, connectors, and use case scenarios.

The failure model of a system should be further refined in terms of the **error impact**. The error impact refers to the potential propagation of errors, which, in turn, causes new errors and may lead to a failure. Naturally, lower error impact implies better containment of errors. The approach proposed by Cortellessa [4] allows architects to specify the probability of propagation when an error has occurred, thus facilitating analysis of the error impact. Similarly, Wallace [41] has proposed a calculus for analysis of error propagation between a system's hardware and software components, intended for the domain of system safety.

Specification of the possible **extent of service outages** is another important aspect of a failure model, and is specified in terms of (1) the part of the system's services that fail due to some errors (e.g., all services failed vs. degraded mode) and (2) the durability of the resulting failures (e.g., permanent vs. restorable). Extent of service outages is determined by the degree to which a given set of errors influences the operation of provided system services. Reliability estimation techniques found in the research literature do not differentiate between different extents of service outages, and consider any error as a cause for failure of all the system services. As mentioned earlier, Roshandel [32] does accommodate one aspect of extents of outages by allowing different failures to be defined as Boolean combinations of component errors.

To complete a system's failure model, it is necessary to include a set of recovery-related parameters. Different **fault tolerance** mechanisms enable recovery in software systems, and should be consequently captured in a failure model. In general, fault tolerance is carried out via error detection and system recovery [1]. For example, checkpointing mechanisms are commonly used to enable restoration of a component's internal state when an error occurs. Furthermore, cold replication mechanisms mitigate the errors of a master component, as component replicas take over the master's functions when necessary.

Error handling mechanisms aim at eliminating errors from the system state [1]. In a system utilizing checkpointing, an erroneous component is restarted and the checkpointed internal state is restored. Similarly, a master replica error in a system employing cold replication is handled by assigning one of the backup replicas to become the master and initializing a fresh backup replica to maintain a constant number of replicas.

Extent of recovery refers to the fragment of the failed system services that can be restored to its full functionality. For example, certain types of failures cannot be completely managed because they are caused by faults outside the scope of the system software (e.g., hardware). Most existing approaches do not consider fault tolerance mechanisms, error handling mechanisms, nor extent of recovery. These approaches model systems at a higher-level of granularity that is not ideal for a detailed specification of these three facets of recovery, so while they can, in theory, incorporate these elements, it is difficult to do so in an accurate and informative way.

3.2 Promising Solution: Architectural Take on the Failure Model Definition

In this section, we provide guidance for both (1) the developers of architecture-based reliability estimation techniques and (2) software architects who want to assess reliability effects of their design decisions. We discuss how a failure model can be obtained by analyzing the system requirements documentation and software architecture specifications. The developers of new estimation techniques can use the following discussion as a set of features that a generally applicable technique should offer. Software architects can use the discussion as a set of guidelines on how to devise a failure model for a specific system. Parts of this discussion can, in some respects, be considered as a refinement of the discussion in [1] from a purely software architectural perspective.

To estimate a system's reliability we primarily need the definition of correct service. The definition of correct service should typically be extracted from system requirements that state the desired external functionalities (i.e., services of the system). To facilitate reliability analysis, these functionalities should then be mapped to particular externally visible system ports that realize the functionality. Alternatively, when a precise requirements documentation is unavailable, the assumed services of the system can be derived from the set of externally visible system ports. In such a case, a software architect or a domain expert would have to manually guide the process in order to derive a correct definition of system services.

As depicted in Figure 3, the definition of a system's failure instances can include various combinations of system elements. Specifically, failure instances are primarily defined in terms of errors of different architectural artifacts: (1) software components, (2) software connectors, and (3) architectural configurations. As noted earlier, a failure can also be defined in terms of (4) deployment features, such hardware hosts or network links. Additionally, failure instance definition can include (5) use case scenario specifications describing important execution sequences. Errors of these individual elements can be classified into three dimensions: (i) complete crashes (e.g., a network link is broken), (ii) abnormal functional behavior (e.g., component X is in an erroneous state), and (iii) abnormal non-functional behavior (e.g., connector S failed to send a message in less than 50 milliseconds). In complex and concurrent software systems, a failure is rarely caused by a single error of any one of the individual elements. Instead, a failure instance can be defined as a combination of (a) logical expressions (e.g., failure of X and failure of Y) and (b) numerical expressions (e.g., more than 15% of the last 1000 requests have not been returned in a timely manner), which can all be expressed together in predicate logic form.

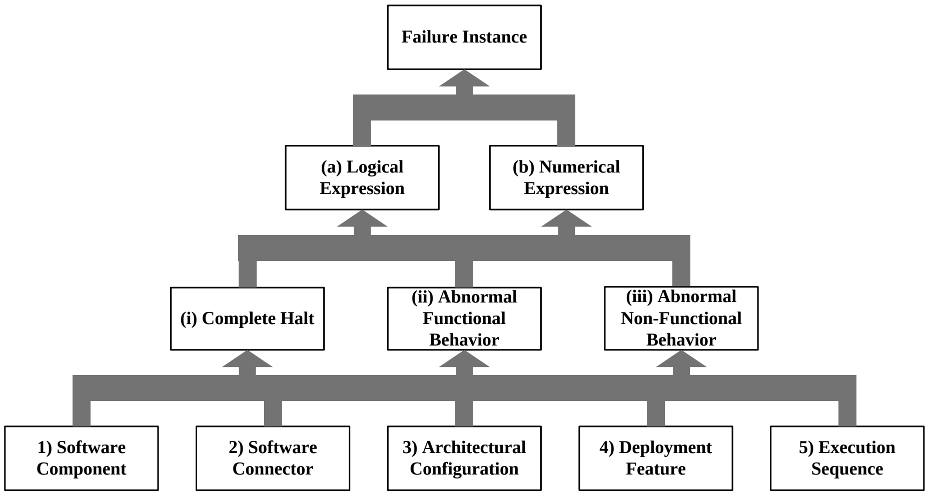

Fig. 3. Multiple dimensions in the definition of a failure instance

We leverage system requirements and architectural specifications to arrive at a failure model. The definition of failure consequence can be composed from specifications of the criticality of system-level services in requirements documents. It directly follows that the failure consequence of a critical service is higher than that of a non-critical service. Furthermore, stakeholder perspectives captured in the requirements must be considered when defining a failure model because different perspectives can produce different failure models even within a single system. System services with the highest criticality should be the first to be considered in reliability analysis. The definition of failure instances is thus targeted at the errors of system elements that realize a critical service (e.g., a set of components implementing a service).

Numerical expressions defining failure instances can also be extracted from system requirements. These requirements usually state that a particular subset of system elements must be operational at all times. Although this is a common way of defining failure instances in the reliability literature (e.g., k out of n replicas must be active), the definition has been limited to system components. In a software architecture, this definition applies to other design elements, such as software connectors and execution scenarios.

Moreover, a failure model must be enriched with information about potential error impacts. For example, a non-critical error of an internal service A can cause a failure of a critical system service B. Thus, the error of A should be included in the failure model. The error impact can be extracted from architectural models specifying component interfaces, behavior, and mutual interactions. Furthermore, the deployment of components and connectors should also be analyzed for error propagation. For example, if a component A fails, it might cause other components collocated on the same host to fail. In this context, deployment refers not only to assignment of components to hardware hosts, but also to operating system processes, virtual machines, and so on.

This information can be captured in both the system requirements and architectural specifications.

The details of fault tolerance and error handling mechanisms can be extracted either directly from requirements documents (if a particular mechanism is imposed) or from architectural models which capture the design decisions. In the latter case, redundant and replicated components suggest particular fault tolerance mechanisms. Additionally, fault-tolerant systems have separate components that realize these mechanisms. These components are further analyzed to extract their recovery-supporting behavior, which is ultimately incorporated into the failure model.

Rugina et al. [33] proposed an approach that extracts dependability models from a system's AADL specifications, while considering multiple facets of a system's erroneous and recovery related behavior. Thus, their approach can be considered as a framework which facilitates modeling and analysis of systems in terms of such behavior. However, Rugina's work is AADL specific, while the discussion we provided in this section is at a higher, language-independent level. Furthermore, we examined how an architect can extract details pertaining to the erroneous and recovery behavior of a system, while Rugina assumes that the architect would already know this.

3.3 Failure Model in the Application Scenario

The *RoboArch* system (recall Section 2.2) has multiple potential users and usage scenarios, which consequently results in different failure models for different perspectives. Figure 4 summarizes example failure instance definitions from different perspectives. Obviously, the definition of a failure instance directly depends on the system service criticality from the particular perspective, which is extracted from the system requirements. A software architect should explore system reliability from each perspective in order to satisfy stakeholder requirements.

The first type of a failure instance from Figure 4, in which any error causes a failure, is the most restrictive one. For example, according to that definition, an error of the *Follower*'s *DB* component causes a failure. The perspective of the system administrator is different, and the most important system services are the user interface services. In a multi-robot application without advanced functionality, a *LineFollower* error implies a failure of all services, since *LineFollower* is the only irreplaceable element in the system. The use-case scenario definition of a failure instance focuses on the importance of the robot following behavior (captured in the communication between the *LineFollower* and the subsequent *PositionFollowers*). This means that a *PositionFollower* error while communicating with a *Sensor* does not cause a failure if the *PositionFollower* recovers in a timely manner for the robot following scenario.

The example from Figure 4 further illustrates how failure instances can be defined in terms of non-functional properties (e.g., timing properties). The connector-centric failure instance identifies the critical connectors without which the system services cannot be provided. The next failure instance example is derived from a requirement of complete system supervision service. To satisfy this requirement, the architectural configuration must allow the administrator to communicate with every robot, which is subsequently captured in the definition of the failure instance. Finally, in a more advanced variant of the *RoboArch* system, the system services failed if there are too many

Highly critical system

A failure will occur if any component or connector error occurs.

System administrator

A failure will occur if either UI or SunSpot error occurs.

Multi-robot application

A failure will occur if a LineFollower error occurs.

Crucial execution sequence

A failure will occur if the robot following scenario fails.

Non-functional properties

A failure will occur if a robot does not get the movement data of its leader for more than 20 seconds.

Connector centric

A failure will occur if errors occur in both the connectors connecting the LineFollower and the PositionFollower with the UI.

Complete control

A failure will occur if the UI cannot communicate with components on the other hosts either directly or via multiple hops.

Fault-tolerant and adaptive multi-robot system

A failure will occur if more than 40% robot hosts have failed or if any component in the fault-tolerance infrastructure is in an erroneous state.

Fig. 4. The example failure instance definitions in *RoboArch*

host errors, which is an example of a numerical formula-based failure definition. Additionally, from that particular perspective, important facets of the system are its adaption and fault-tolerance capabilities, which is reflected in the definition.

Analyzing the system from any of the above perspectives without further insight may be very misleading. System element errors which are not included in definitions of failure instances for critical system services can still cause a critical service to fail. For example, a *SunSpot* error may propagate to the *LineFollower* during their interactions. Furthermore, if there is a documented requirement or a design decision stating that all components on a host have to be running inside a single virtual machine, such as JVM, a *Sensor* error may cause a *LineFollower* error because they are running in the same process. Similarly, if the *EnergyMonitor* is in an erroneous state, it will have an impact on the whole system, as the robot it is running on will drain all the battery power and will not recharge when close to a docking station.

To complete the failure model, we extract recovery-related parameters. *RoboArch* contains components that support fault-tolerance. These components monitor the application components for errors and perform planning which results in redeployment in case an error occurs. Fault-tolerance components are defined in the architectural specification as ordinary components, but their behavior is specifically intended for error detection and system recovery. The failure model must include the specification of fault tolerance and error handling mechanisms because the system reliability is dependent on them. For example, system reliability is different depending on whether the component in charge of recovery just restarts an erroneous component or remotely fetches and deploys a different component stored on the *PDA*. Both techniques influence system reliability: the first one may be faster, but the second one can provide more guarantees that the component will not fail under the same circumstances. Thus, if a failure model considers these recovery-related parameters, a software architect is able to weigh the costs of each potential mechanism with its actual reliability gains and select with more confidence the most appropriate one for the system under design.

4 Challenge 2: Parameter Uncertainties

Defining a failure model is only the first step in estimating a system's reliability; a number of detailed architectural model and failure model parameters are still required for a rigorous analysis. In this section, we characterize the types of information that are commonly required for architecture-based reliability estimation; we also pinpoint the subset of this information that is commonly defined as part of the architectural development process and that can be reasonably assumed to be available (Section 4.1). We then discuss how information that is not readily available can be approximated or derived from additional sources (Section 4.2). Finally, we explain how we derived reliability parameters for the *RoboArch* system (Section 4.3).

4.1 Reliability Parameters

When all of the parameters of the architectural model and the failure model are known with some certainty, reliability analysis of architectural models can be pursued with confidence. However, architecture-based reliability estimation techniques must be applicable to large-scale, complex systems, for which software architects are often lacking a complete understanding of (1) the erroneous behavior of application logic, (2) a definitive usage profile, and (3) the behavior and effectiveness of recovery. As a result, most of these parameters, which can in principle be captured in an architectural model, rarely are available in practice.

Erroneous Behavior. As described in Section 3, a failure instance is generally defined as some function or combination of errors in the system components or environment. Therefore, in order to calculate the probability of failure, and, by extension, the system reliability, the **probabilities of errors** must be specified (i.e., probabilities that a particular fault is activated). However, the probability of errors cannot, in most cases, be directly extracted from an architectural model. The probability of a given error may depend on a number of factors which cannot be fully determined at architecture design time, such as hardware and network design or the way in which application logic is implemented.

All architecture-based reliability estimation techniques naturally have to use component error probabilities to analyze the reliability, but only a couple of approaches [13, 30] explore the derivation of those probabilities. Goseva-Popstojanova [13] relates the probability of a component error to the complexity of components' statechart models. Reussner [30] derives the error probability of component services through a combination of the reliabilities of method bodies, method calls and returns, and the environment, but does not specify how these input values are obtained. Because reliability estimation techniques require error probabilities as inputs, methods for deriving these probabilities from alternative information sources described in Section 4.2 complement these techniques, and should be developed more fully. Error probabilities are also related to operational profiles, which we describe next.

Usage Profile. A system's usage in terms of its operational profile is a key ingredient of system reliability. Defining an operational profile requires specifying the **frequency of execution** of different system services and operations, the frequency and probability of possible **user inputs**, and the **operational contexts** in which these processes and inputs occur [26]. The operational profile is necessary in order to estimate the effects of defects in the system. For example, a buggy component whose services are frequently invoked will affect system reliability much more than a buggy component whose services are rarely invoked.

The frequency of execution of system services is usually not captured in an architectural model. The set of user inputs to the system are normally specified in an architectural model, but the frequencies and probabilities of each possible input are not. This information can, in most cases, be only approximated. The operational contexts of different system processes can be determined from a compositional evaluation of component behaviors, concurrency mechanisms, computational resources, and so on. For example, an architectural model should specify which services may execute in parallel; this provides information about which other processes may be executing (and consuming computational resources) when a given service is invoked. The availability of computational resources may, in turn, affect a service's reliability.

The frequency of execution of system services and operations is essential for reliability estimation techniques. Cheung [2] requires the probabilities of transitions between internal component states, while system-level approaches need the probabilities of transfer of control between components and services [3, 12, 19, 35, 36, 20, 30], or the probabilities of execution of particular execution paths [5, 17, 31, 34, 43]. Rodrigues [31] also requires the probabilities of transfer of control between execution paths. However, only Cheung [2] proposes a way of obtaining these parameters, by deriving them from a combination of expert knowledge, functionally similar components, and system simulations.

Cortellessa [5] presents one of the rare approaches that explicitly provide a modeling notation that accounts for user inputs in reliability estimation, using annotations of UML use case diagrams. However, this approach does not specify how the required quantitative data can be obtained. All of the architecture-based reliability estimation approaches could benefit from a technique that provides a description of how users interact with the system by leveraging it to derive operational profiles. As Cheung [2] argues, the connection between user inputs and individual service invocations is not as explicit at the component-level as it is at the system-level, so component-level reliability estimation techniques have more difficulty incorporating this reliability ingredient in analysis.

Many reliability estimation techniques pay little or no attention to operational contexts and assume sequential systems. The exceptions are [7, 31, 42], which do consider concurrency. The challenges related to concurrency are discussed in more detail in Section 5. Additionally, none of the existing approaches takes the contention over computational resources into account. More specifically, a component may spend additional time waiting for the resources to be available, which makes the component take longer to service a request. If the software requirements specify performance constraints (e.g., a request has to be completed in X seconds), contention over resources may lead to a higher probability of not satisfying such requirements.

Recovery Information. If a system employs fault-tolerance mechanisms, it is necessary to know two recovery-related parameters to calculate system reliability: the **likelihood of recovery** and **time to recovery**. An error which is successfully handled by fault-tolerance mechanisms will likely not cause a permanent failure; thus, the probability of the recovery must be known. Steady-state definitions of reliability require the time to recovery to be specified because they calculate the proportion of time the system spends in a failure mode.

The likelihood of recovery cannot be solely determined from an architectural model. For example, an architectural model may specify a mechanism to recover corrupted data (e.g. checksum). However, such a mechanism can likely only tolerate certain errors, and an architectural model usually does not specify the probability of one error type versus another as this depends on user inputs. In general, the time to recover from an error also cannot be ascertained from an architectural model. Firstly, recovering from an error may require manual intervention, whose duration can be highly variable. Secondly, even when error handling mechanisms are fully automated, the time required to execute may depend on other unknown parameters, such as the services requested by users and the state and availability of computational resources.

The approach from Cheung [2] is the only architecture-based reliability estimation technique that explicitly models the characteristics of component error recovery. The likelihood of recovery and time to recover from a given error is accounted for in a limited form, as a recovery transition probability is assigned to each erroneous state. The remaining approaches do not explicitly account for recovery, but some approaches [30, 31, 43] can be extended to provide an analysis that includes likelihood of recovery. Similarly, these approaches can be enhanced to include time to recovery. However, obtaining accurate time to recovery data is a harder problem than incorporating it into the analysis techniques.

4.2 Promising Solution: Additional Information Sources

As described above, to be able to predict the reliability of a software-intensive system, a number of parameters must be known. However, this information is usually not contained in an architectural model. Furthermore, other required information may not be available if the architectural model is incomplete or vaguely defined. This section describes how missing information required for reliability analysis can be derived from various indirect (and possibly imprecise) sources.

Key system scenarios. The selection and decomposition of key system scenarios – a common step in the requirements elicitation process – provides information relevant to the operational profile by defining important system execution paths. This, in turn, helps an architect to parameterize a reliability model with the probability of different user inputs and processing sequences. Mapping user interactions to the underlying component execution sequences may indicate the relative frequency that the various component services are invoked. For example, a service that executes as part of numerous system use cases is likely to be executed frequently in practice.

Functionally-similar and predecessor systems. It may be possible to estimate reliability parameters by examining existing systems that are functionally similar. A previous version of a software system could be used to approximate error and recovery probabilities. For example, information related to recovery parameters of a new system with fault-tolerance support may be extracted from a system with a similar fault-tolerance infrastructure. One problem with this approach is that the new system will likely be implemented differently, so that its failure model parameters may not be directly related to those of any existing system. Moreover, if new functionality is added, the reliability of new components and services implementing that functionality could be distinctly different than the reliability of the previous version. Existing systems that provide similar functionality may also be used to gather operational profile information from the system's execution traces. These execution traces, however, are not necessarily specified at the level of detail that is required by a reliability analysis technique. In such cases, we suggest utilization of techniques that can approximate the operational profile from incomplete information, such as Hidden Markov Models [29] which are used in Cheung's approach [2].

User information. By gathering information from the intended users of a software system, some reliability parameters related to the operational profile can be estimated. A description of the users' behavior can indicate not only the types and frequencies of user inputs, but also what types of workloads are put on the system simultaneously, which helps in determining operational context. For example, novice users tend to work slower and use the provided services in an unexpected order, while expert users will work faster and use the system services appropriately. Alternatively, an architect could "rapidly prototype the system" to gather usage statistics. Finally, the requirements documents will often capture the ways users intend to use the system.

Software systems expert knowledge. An architect's previous experience, technical literature, accepted community practices, and industry trends and standards can be invaluable in estimating reliability parameters. For example, within a certain family of applications, some types of software components may be known as common sources of errors, while other types of components are known to be very reliable. Additionally, an experienced software architect can predict operational profile of a new system based on the operational profile of a system that was previously designed. For example, if execution traces of a grid system are not available, a software architect who previously designed grids may be able to estimate the operational profile with satisfactory accuracy. The reliability of off-the-shelf hardware is usually provided by the device manufacturer, and can be leveraged within reliability models that consider hardware faults. Estimation of network reliability, by itself, comprises a large body of research, but network reliability models can, in some cases, be combined with architecture-based reliability estimation.

Domain expertise. Software systems whose primary functions involve interaction with the physical environment (e.g., sensor networks and robotics systems), largely depend on the often unpredictable properties of that environment. Therefore, a person with primary expertise in software design might not be able to provide meaningful approximations of the necessary parameters. Rather, a domain expert who has an in-depth understanding of the processes occurring in the physical environment is perhaps the best available information source of the operational profile and failure-related information. For example, a weather expert will know the likelihood of extreme weather conditions that may cause a sensor network deployed in a forest to fail.

Other models. Automatic simulation of architectural models [11] can be leveraged to improve the potential analysis of operational profile for more complex components or systems. However, simulation techniques still require information related to a components or the system's operational profile, which would have to come from other sources. Models other than software architecture can be an important source of reliability parameters. For example, detailed hardware models can provide key information about the likelihood of some errors, such as the mean-time-to-failure of a persistent storage drive. In systems that are deployed in large organizations, high-level specification of business processes and organizational structure can be used to extract the probable usage characteristics.

4.3 Addressing Parameter Uncertainty in the Application Scenario

Although the *RoboArch* system was constructed faithfully according to a detailed architectural model, the model still did not contain a number of parameters necessary for a meaningful architecture-based reliability analysis. Thus, we drew on various available external sources of information to approximate these parameters. Figure 5 depicts the information sources we used for particular types of reliability model parameters.

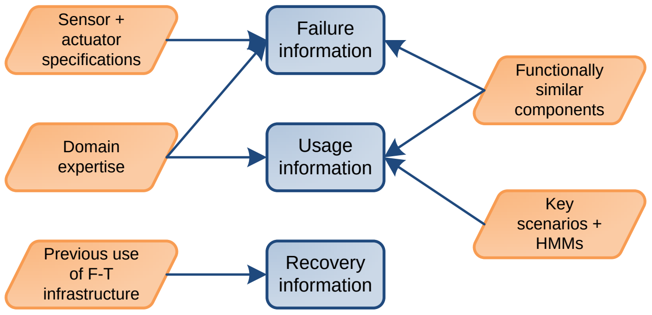
Sensor + actuator specifications
Domain expertise
Previous use of F-T infrastructure
Failure information
Usage information
Recovery information
Functionally similar components
Key scenarios + HMMs

Fig. 5. Sources of information for *RoboArch* system

Specifications of the used hardware provided data about the frequency of hardware faults. In particular, the various sensors and actuators installed on the robot were relatively unreliable, and the occurrence of sensor and actuator errors needed to be considered for an accurate reliability estimation. We were able to determine both the mean-time-to-failure and the precision of readings for all sensors in the *RoboArch* system from manufacturer documentation. These values were then plugged into our reliability models.

An unrelated system that employed the same fault tolerance mechanisms as *RoboArch* indicated the time necessary to recover from software errors. Both *RoboArch* and the MIDAS system [24], which is a family of sensor network applications, use the fault-tolerance mechanisms provided by Prism-MW [23], a lightweight middleware for mobile and embedded systems. Based on our experience injecting faults into MIDAS, we knew that the Prism-MW fault-tolerance service could transparently replace an erroneous component in less than 2 seconds.

Interviews with domain experts guided our approximation of environment variables. One important environment variable that has a significant impact on the reliability of the *RoboArch* system is the prevalence and shape of obstacles. When an obstacle is in the robots path, it must temporarily leave the path to go around the obstacle. When numerous obstacles are present, the robot may be unable to relocate the path, which causes failure of most system services. The robot is more successful at navigating around circular obstacles than irregular obstacles with sharp corners. A domain expert helped us to estimate the number of obstacles which would be commonly found in an outdoor environment and the shape of those obstacles.

Based on previous experience developing robotics software components, we assumed that certain software components had a low error probability. These reliable components included the *LineFollower* and *PositionFollower*. Because these components do not interact directly with robot hardware, they were less likely to experience external faults. Moreover, the algorithms used within these components were reused off-the-shelf, and had previously functioned with high reliability.

Finally, we used the specifications of key system scenarios accompanied with the expected rates of their occurrence. Although defined at the level of components' service invocations, this information helped us deduce a more detailed operational profile at the level of internal component operations. For this task, we leveraged aforementioned Hidden Markov Models which approximated operational profile based on the information about external service invocations. The actual operational profile of the implemented system has proven to be inside the ranges approximated using HMMs.

5 Challenge 3: Complexity and Scalability

In this section, we consider the challenges of complexity, scale, and concurrency in architecture-based reliability estimation. In Section 5.1, we outline the issues of complexity, scale, and concurrency in modern software and highlight the shortcomings of many existing approaches when analyzing such systems. As we show in Section 5.2, a promising solution for these challenges is application of hierarchical principles, as well as aggregation and truncation methods. In Section 5.3, we illustrate the rapid state space growth for the *RoboArch* system when simple parallel composition is applied and compare it with the tractability of a hierarchical model.

5.1 Complexity in Architecture-Based Reliability Estimation

Modern software systems are growing rapidly in their scale, complexity, and the concurrency of their constituent elements. Consequently, early design models are becoming more complex, which, in turn, makes them harder to analyze. A simple way to reduce the complexity of models is to simply abstract away some information and perform the analysis on a simplified model. For example, complex component behaviors can be modeled stochastically. However, such an approach can suffer from significant information loss in the abstraction process, which makes the analysis results less useful. Thus, it is beneficial to explore the largest tractable reliability model in a controlled manner to obtain the most useful estimates.

A number of approaches estimate system reliability as a function of the reliabilities of individual components, without going into sufficient detail regarding the component interactions (e.g., [3, 30, 35, 36, 42, 43]). Although such techniques can be scalable, they are not entirely satisfactory because they assume that reliabilities of individual components are known. Furthermore, most architecture-based reliability estimation approaches, with some notable exceptions [7, 31, 42], assume that the system under evaluation is sequential, and that the Markov property holds (i.e., transition probability to the next state is determined only by the current state). These assumptions, in turn, allow, the above approaches to account for internal component structure with only a linear $O(n)$ rise in the number of the states that have to be analyzed. Specifically, this is accomplished by replacing a reliability model state representing a component with n new states representing that component's internal structure. Overall, this results in an aggregate model with $O(m \times n)$ states.

Modern software systems are vastly concurrent, and a simplistic approach that assumes sequentiality clearly cannot provide realistic reliability estimates. Sequential approaches may, however, be modified to account for concurrency by applying parallel composition to the system elements' reliability models. In general, states of the system model represent all combinations of the internal states of all components. Specifically, a state in the system reliability model consists of multiple variables, where the i^{th} variable corresponds to the state of Component C_i. We refer to this approach as the **brute force** approach in the rest of this section. Unfortunately, the brute force approach suffers from significant scalability problems due to the exponential growth of states: a system with n components, each with m internal states, will have a system reliability model with up to m^n states. Such huge models are prohibitively costly to generate and solve even for systems with a modest number of components. In the next subsection, we discuss promising directions for reducing the analyzed state space with the goal of little information loss.

5.2 Promising Solution: Hierarchy and Truncation Techniques

We believe that principles of **hierarchy**, **aggregation**, and **truncation** are the most promising directions for improving analyzability of modern software. As one of the main software engineering principles, hierarchy is already incorporated to some extent into software architectural models (e.g., a system is decomposed into components and connectors which are, in turn, decomposed into internal objects). Hierarchical approaches for reliability analysis consider each subsystem and combine the results of the subsystems to obtain a system-level reliability estimate. The motivation for a hierarchical approach is the following: solving many smaller models is more efficient than solving one large model, resulting in savings in computational costs. Hierarchical approaches retain details about the system while managing the complexity of the reliability model. Their accuracy is largely a function of how the submodels are composed to compute the final reliability estimates.

One natural approach is to construct the submodels based on the component structure. Another possible approach is to consider use case scenarios. As we mentioned earlier in the article, use case scenarios are a common way to focus design activities on important execution sequences. Execution sequences captured in use case scenarios tend to have the biggest impact on system's reliability. This fact has been leveraged by a number of architecture-based reliability estimation approaches. For example, the approaches in [5, 13, 31, 43] leverage use cases and scenarios to estimate reliability by decomposing systems into smaller subsystems. These scenarios, usually described as UML sequence diagrams, can be transformed into Markov-based reliability models using, for example, the approach described in [13]. The scenario reliability is determined by solving these fine-grained, scenario-based submodels using standard techniques such as the one proposed by Stewart [38].

The results of scenario-based models can be combined to obtain an overall estimate of system reliability as a weighted sum of scenario reliabilities (weighted by the probability that a scenario occurs). However, most current approaches [5, 13, 43] assume that the probability of each scenario occurring is known. Rodrigues [31] combines scenario estimates according to a higher-level scenario diagram which describes allowed sequences of scenarios themselves. These scenario-based approaches still make an assumption of sequentiality at the level of the system scenarios. However, we expect that a technique that accounts for scenario concurrency in a brute force manner will often have a significantly smaller state space than techniques that model all component states.

A component-level reliability estimation technique such as Cheung's [2] can form the basis for a hierarchical reliability estimation approach. Hypothetically, the estimates obtained from a component-level approach can be directly plugged-in into any one of the existing system-level reliability analysis approaches. As we noted earlier however, the existing system-level approaches assume sequential systems and do not consider the details of component interactions. Consequently, their analysis capabilities are hampered by these facts. One possible way to reduce the complexity of the brute force reliability model is to represent each component in aggregate states, e.g., either as operational or non-operational, and then plug-in the reliability estimates of individual components (e.g., [7]). Such enhancements will constitute a significant focus of our future research in the field of architecture-based reliability estimation.

To further improve computational tractability of reliability models, we also propose using techniques for model truncation and aggregation [38]. For instance, details of a scenario that rarely occurs or whose execution is believed to be highly reliable can be truncated. Alternatively, multiple instances of the same component may result in symmetries in the reliability model that are amenable to aggregation or lumping techniques [38]. These techniques, along with the hierarchical approaches outlined above, can be used in combination to further reduce the size of a system's reliability model and consequently allow system modelers to build tractable reliability models of complex systems. Furthermore, in models of concurrent systems, the truncation techniques are likely to find symmetries when the internal states of a small part of the system change.

Generally speaking, the hierarchical approaches, as well as truncation and aggregation techniques [38], are only approximations. Therefore, it is important to consider the trade-off between the precision in reliability estimates and the scalability of the proposed approach. While scalability can be quantified as the number of operations needed to generate and solve the models, precision in reliability estimates is not trivial to compute. Imprecision in reliability estimates may result in unnecessary costs in addressing a defect when the estimates are too pessimistic or in underestimating the effect of a defect when the estimates are too optimistic. Quantifying the cost of estimate imprecision depends on factors such as the type of system in consideration (e.g., safety-critical system vs. a system for demonstrations), the system's context (e.g., military software vs. online shopping system), and the technique used to compute the system reliability. Furthermore, estimating this cost at architecture design time is not trivial because of the numerous uncertainties about the system, as discussed in Section 4.

5.3 Addressing Complexity in the Application Scenario

In this section, we explore how the complexity of the *RoboArch* system can be handled using the hierarchical approaches, as well as the aggregation and truncation techniques (recall Section 5.2). The main aspects of *RoboArch*'s behavior can be captured with five distinctive scenarios given in Table 1. For example, the scenario-based approach from Goseva-Popstojanova [13] would generate reliability models of the five scenarios separately, and solve the models for scenario reliabilities. System reliability can then be computed as the weighted sum of scenario reliabilities, weighted by the probability that a scenario occurs.

Table 1. Use-case scenarios of *RoboArch*

Num	Scenario	Description
1	*Leader Robot Movement*	Controls how a leader robot should move
2	*Follower Robot Movement*	Controls how a follower robot should move
3	*SunSpot Data*	Describes the interactions between the leader robot and *SunSpot* component
4	*Energy Management*	Describes a robot's interactions with *EnergyMonitor*
5	*User Request*	Describes the system's interactions with users through *PDA*

As discussed in Section 5.2, the existing hierarchical approaches do not capture system concurrency in their models. Therefore, we explored the effects of modeling *RoboArch* using the envisioned hierarchical technique that captures concurrent nature of the system, as discussed above. We compared size of the model generated by the hierarchical approach with the size of the model generated by the brute force approach, both accounting for concurrency (recall Sections 5.1 and 5.2). The results of the analysis are depicted in Figure 6, which shows how the size of the model grows with the number of *Followers* in the system. Note that the y-axis in Figure 6 is plotted in log-scale. The size of model of the "brute-force" approach grows quickly, and the model is expensive to solve. On the other hand, the growth in the size of the model is slower in a hierarchical approach. Additionally, solving many smaller submodels is more cost efficient than solving one huge model.

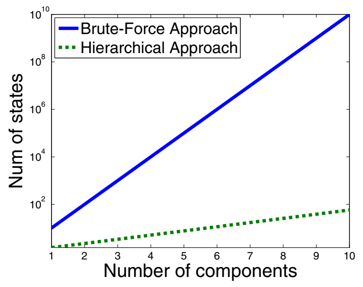

Fig. 6. Comparison of the complexity of "brute-force" and hierarchical approaches

Let us now illustrate how aggregation techniques can be applied to the concurrent system reliability estimation approach in [7] to model a *Follower Robot*. A simplified model of a *Follower* is depicted in Figure 7, which contains only the *PositionFollower* component, and two *Sensor* components. A state (LF, S_1, S_2) indicates the state of each component, where 1 indicates that the component is operational, and 0 indicates that the component has failed. As an example, we consider the follower robot to be reliable if at least one *Sensor* is operational. If it does not matter which *Sensor* is operational, we can aggregate states (1,0,1) and (1,1,0), as well as states (0,1,0) and (0,0,1), thus significantly reducing the state space.

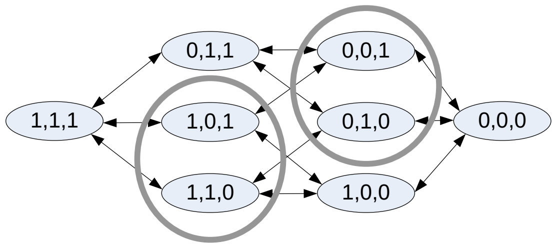

Fig. 7. Reliability model of the *Follower Robot*

The results obtained for the *RoboArch* system clearly showcase the different trade-offs when selecting a reliability estimation technique. For example, *RoboArch* with a single *Follower* can be analyzed with the brute force approach in order to obtain the most precise reliability estimates. However, with the growth of the system size, the reliability models become computationally costly, but still tractable. In such a case, it is desirable to speed up the analysis by applying the aggregation and truncation techniques. Finally, when the brute force generated model grows over the computational tractability threshold, some hierarchical technique has to be applied (possibly enhanced with the aggregation and truncation techniques) to obtain the reliability estimates. Furthermore, the designers of hierarchical techniques should clearly specify the assumptions, as well as the approximations and abstractions they are performing on the reliability models, in order to allow a software architect to choose an appropriate technique for a specific development context.

6 Conclusions

This article defined three core challenges in architecture-based reliability estimation: (1) defining a failure model, (2) obtaining reliability-related parameter estimates, and (3) dealing with the scale and complexity of modern software. We outlined each of these challenges, described promising solutions to them, and illustrated the most important concepts on the *RoboArch* system. Our goal in this article was to provide guidance for enhancing the existing architecture-based reliability estimation approaches and to motivate development of new techniques that are able to deal with the described challenges more efficiently. First, we envision the development of more formal methods for devising a system's failure model from architectural specifications and requirements documents. Second, we encourage the development of new techniques for derivation of reliability related parameters from available information sources. Third, further development of hierarchical reliability analysis techniques seems a particularly promising solution for the challenges caused by the scale and complexity of modern software. The proposed directions will frame our future research, but we also hope that they will spur the broader software engineering community into developing techniques for addressing the described challenges more effectively.

References

1. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.
2. Leslie Cheung, Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. Early prediction of software component reliability. In *Proceedings of the 30th international conference on Software engineering*, pages 111–120, 2008.
3. Roger C. Cheung. A user-oriented software reliability model. In *IEEE Transactions on Software Engineering*, volume 6, pages 118–125, 1980.
4. Vittorio Cortellessa and Vincenzo Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In *Proceedings of the 10th*

International ACM SIGSOFT Symposium on Component-Based Software Engineering, pages 140–156, 2007.

5. Vittorio Cortellessa, Harshinder Singh, and Bojan Cukic. Early reliability assessment of UML based software models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 302–309, 2002.
6. Olivia Das and C. Murray Woodside. Layered dependability modeling of an air traffic control system. In *Workshop on Software Architectures for Dependable Systems*, May 2003.
7. Rehab El-Kharboutly, Reda A. Ammar, and Swapna S. Gokhale. UML-based methodology for reliability analysis of concurrent software applications. *International Journal of Computers and Their Applications*, 14(4):250–259, 2007.
8. Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.
9. Amrit L. Goel and Kazuhira Okumoto. Time-dependent error-detection rate models for software reliability and other performance measures. *IEEE Trans. on Reliability*, 28(3), 1979.
10. Swapna S. Gokhale. Architecture-Based software reliability analysis: Overview and limitations. *IEEE Transactions on Dependable and Secure Computing*, 4(1):32–40, 2007.
11. Swapna S. Gokhale, Michael R. Lyu, and Kishor S. Trivedi. Reliability simulation of component-based software systems. In *Proceedings of the 9th International Symposium on Software Reliability Engineering*, pages 192–201, 1998.
12. Swapna S. Gokhale and Kishor S. Trivedi. Reliability prediction and sensitivity analysis based on software architecture. In *ISSRE 2002*.
13. Katerina Goseva-Popstojanova, Ahmed Hassan, Walid Abdelmoez, Diaa Eldin M. Nassar, Hany Ammar, and Ali Mili. Architectural-level risk analysis using UML. *IEEE Transactions on Software Engineering*, 29(3), 2003.
14. Katerina Goseva-Popstojanova and Kishor S. Trivedi. Architecture-based approaches to software reliability prediction. *Computers and Mathematics with Applications*, 46(7):1023–1036, 2003.
15. Anne Immonen and Eila Niemel. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008.
16. Z. Jelinski and P.B. Moranda. Software reliability research. *Statistical Computer Performance Evaluation*, 1972.
17. Saileshwar Krishnamurthy and Aditya P. Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *In Proceedings of ISSRE 1997*.
18. Ivo Krka, Leslie Cheung, George Edwards, Leana Golubchik, and Nenad Medvidovic. Architecture-based software reliability estimation: Problem space, challenges, and strategies. In *DSN '08 Companion: Proceedings of DSN 2008 Workshop on Architecting Dependable Systems*, 2008.
19. P. Kubat. Assessing reliability of modular software. *Operations Research Letters*, 8, 1989.
20. Bev Littlewood. A reliability model for Markov structured software. In *Proceedings of the international conference on Reliable software*, pages 204–207, 1975.
21. Bev Littlewood and J.L. Verrall. A bayesian reliability growth model for computer software. *Applied Statistics*, 22:332–346, 1973.
22. Michael R. Lyu. *Handbook of Software Reliability*. Princeton University Press, 1996.
23. Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, 2005.
24. Sam Malek, Chiyoung Seo, Sharmila Ravula, Brad Petrus, and Nenad Medvidovic. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support.

In *Proceedings of the 29th International Conference on Software Engineering*, pages 591–601, 2007.

25. Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
26. John D. Musa. Operational profiles in software-reliability engineering. *Software, IEEE*, 10(2):14–32, 1993.
27. John D. Musa. *Software Reliability Engineering*. McGraw-Hill, 1999.
28. John D. Musa and Kazuhira Okumoto. Logarithmic poisson execution time model for software reliability measurement. In *Proceedings of. Compsac 1984*.
29. Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.
30. Ralf R. Reussner, Heinz W. Schmidt, and Iman H. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66(3), 2003.
31. Genaina Rodrigues, David S. Rosenblum, and Sebastian Uchitel. Using scenarios to predict the reliability of concurreny component-based software systems. In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering*, 2005.
32. Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. A Bayesian model for predicting reliability of software systems at the architectural level. In *Proceedings of the 3rd International Conference on the Quality of Software Architectures*, 2007.
33. Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaniche. A system dependability modeling framework using aadl and gspns. In *Architecting Dependable Systems IV*, volume 4615 of *Lecture Notes in Computer Science*, pages 14–38, 2007.
34. Martin L. Shooman. Structural models for software reliability prediction. In *In Proceedings of ICSE 1976*.
35. Kyle Siegrist. Reliability of systems with Markov transfer of control. *IEEE TSE*, 13(7), July 1988.
36. Kyle Siegrist. Reliability of systems with markov transfer of control, II. *IEEE Trans. Softw. Eng.*, 14(10), 1988.
37. Ian Sommerville. *Software Engineering*. Addison Wesley, 2004.
38. William Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.
39. Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. John Wiley and Sons, 2009.
40. S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, Y. Xie, and W-L. Hung. Reliability-centric hardware/software co-design. In *ISQED '05: Proceedings of the 6th International Symposium on Quality of Electronic Design*, pages 375–380, Washington, DC, USA, 2005. IEEE Computer Society.
41. Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Elec. Notes in Theoretical Computer Science*, 18(3):351–356, 2002.
42. Wen-Li Wang, Dai Pan, and Mei-Hwa Chen. Architecture-based software reliability modeling. *Journal of Systems and Software*, 79(1), 2006.
43. Sherif M. Yacoub, Bojan Cukic, and Hany H. Ammar. A scenario-based reliability analysis approach for component-based software. *IEEE Transactions on Reliability*, 53(4):465–480, 2004.