Abstract

Reliability is one of the most critical extra-functional properties of a software system, which needs to be evaluated early in the development process when formal methods and tools can be applied. Though many approaches for reliability prediction exist, not much work has been done in combining different types of failures and system views that influence the reliability. This paper presents an integrated approach to reliability prediction, reflecting failures triggered by both software faults and physical-resource breakdowns, and incorporating detailed information about system control flow governed by user inputs.

1. Introduction and Related Work

Software systems are steadily growing in size and complexity. At the same time, the loss of user acceptance as well as economical risks connected to malfunctioning systems are becoming critical factors for the competitiveness of software products. Measures for achieving reliability have to be embedded into all phases of the software engineering process. This is particularly true for system design time, when the outline of the later product is constituted.

A large number of approaches exist to predict software-system reliability. However, their application is typically limited to certain failure classes or exclusively concentrates on software reliability, neglecting the influence of hardware resources and system deployment. The examined models use to be built on a high level of abstraction, assuming the reliability of individual software components as given, and only considering the system level [4, 13]. The approaches that go deeper in the abstraction, and compute the reliability of individual components [10], do not take into account important system characteristics, like the usage profile that determines the actual execution of the system. These and other issues have already been discussed also by other authors [3, 6].

In this paper, we propose an approach that takes into account the reliability on different levels of system architecture, starting with services and their internal computation, and propagating it through the reliability of individual components to the system level. We consider the influence of system usage and explicitly model the propagation of a usage profile throughout the architecture. We take into account not only software faults, but also system deployment and the effect of physical-resource breakdowns. Thus, we strive for a comprehensive view and a highly qualified answer to the question how reliable a system will be. As the basis for our work, we use the Palladiom Component Model (PCM) [2], which is a formal prediction framework, incorporating different aspects influencing operation of the examined system. Since PCM is designed for performance prediction, we extend its capabilities to include the support for reliability prediction.

The paper is organized as follows. Section 2 introduces the main terms and concepts. Section 3 describes the modeling and prediction of system reliability, followed by an illustrating example in Section 4. After that, Section 5 provides some notes about the implementation. A summary and outlook are given in Section 6.

2. Terminology and Scope

This section explains the basic concepts of our approach. Section 2.1 introduces our view of a software system, and Section 2.2 describes the sources of failure that are important to consider when predicting system reliability.

2.1. Software Systems and Reliability

Our approach is concerned with the reliability of a software system. We follow a holistic view in which not only the end-user application belongs to the system, but also supporting software layers like application servers and operating systems, as well as the physical resources like CPU
and memory, which the system needs for execution. We assume that the end-user application is built of software components in the sense of Szyperski’s definition [12]: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

A component-based software system provides services to its users. The term service is used here in a very general sense as a functionality provided by the system, described in terms of an interface with one or more operations. Each operation may have input parameters and a return value. The system usage is characterized by the frequency of service operation invocations, the input parameter values used for each invocation, and the sequences of calls performed by the users.

From the user’s point of view, each invocation of a service operation can be seen as a random experiment with two main possible outcomes: either the operation is executed successfully, or it fails. Thereby, failure means any departure of the system’s behavior from the user's expectation. A failure may be detected by the system itself and indicated to the user, or it may be completely unnoticed by the system. In both cases, we consider the operation as failed. We associate the probability of failure of a service with the system. In both cases, we consider the operation as failed. The system usage is characterized by the frequency of service operation invocations, the input parameter values used for each invocation, and the sequences of calls performed by the users.

From the user’s point of view, each invocation of a service operation can be seen as a random experiment with two main possible outcomes: either the operation is executed successfully, or it fails. Therefore, failure means any departure of the system’s behavior from the user’s expectation. A failure may be detected by the system itself and indicated to the user, or it may be completely unnoticed by the system. In both cases, we consider the operation as failed. We associate the probability of failure of a service with the well-known Probability of Failure on Demand (PFD). Our goal is to determine the probability of success, which can be expressed as $PS := P(SUCCESS) = 1 - PFD$. This probability constitutes an important metric for the assessment of the software system’s reliability.

### 2.2. Sources of Failure

Considering potential reasons for failure of a service operation, we identify two main sources of failure, which are software faults on the one hand, and physical-resource breakdowns on the other hand.

Software faults are defects included in the system’s implementation. Even a high-quality development process generally cannot eliminate every fault in the system. Some faults are left and may be activated at run-time to produce failures. However, the faults that cause failures in an easily reproducible way are mostly eliminated during system test. Those faults are also referred to as Bohrbugs. The remaining faults are hard to localize and have complex activation patterns; these are referred to as Mandelbugs [5]. In many cases, Mandelbugs are hidden in supporting software layers or exhibit long time lags between fault activation and failure occurrence, and are therefore hard to find. Assuming a high-quality development process, failures occurring at system run-time will be mainly caused by Mandelbugs.

Physical-resource breakdowns mainly happen due to hardware wear out and represent an inevitability in software systems. Typically, a broken-down resource, which might be a CPU, memory, or a network link, is eventually repaired or replaced by a functionally equivalent new resource. The reliability of such resources is expressed in terms of Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR). We assume that any service operation that uses a certain resource $r$ at some point in time $t$ fails if $r$ is broken-down at $t$. Still, reasoning about the probability that the control flow of an operation uses a certain resource is non-trivial, since the control flow depends on the implementation of involved services and user inputs that propagate through them. This is especially true for highly distributed software systems, which exhibit complex deployment settings.

The sources of failures described here are by no means complete. They rather point out two main inherently different types of failures that need to be taken into account when reasoning about system reliability. A more complete classification can be found elsewhere [1].

### 3. Reliability Modeling

This section describes our approach to modeling a software system in a way that enables reliability prediction. As pointed out in Section 2.2, software faults and resource breakdowns do not need to automatically lead to a failure of a service operation. Rather, it is the interplay among all the elements of the system’s architecture that determines the occurrence of a failure. Therefore, we build our approach to reliability-driven system modeling on a framework that already incorporates different aspects determining the system’s operation. The framework is based on the Palladio Component Model (PCM), which is highly expressive. It includes not only software components and their composition, but also an abstraction of component behavior, system usage and deployment. The framework is designed for performance prediction and reaches very good accuracy thanks to the concept of parametric dependencies — propagation of system usage information through the whole component architecture.

In its current state, the PCM allows only for evaluation of system performance. For prediction of the success probabilities of service operations, it needs to be extended with new concepts that reflect the different sources of failures identified in Section 2.2. As each of these requires its own modeling constructs, Sections 3.1 and 3.2 describe them separately. Finally, Section 3.3 consolidates the results into a single comprehensive modeling method.

#### 3.1. Software Faults

The occurrence of failures due to software faults depends on the activation of faults during execution of a service operation, which in turn depends on the control and data flow through the system. The control and data flow is determined
by the implemented component behavior and the input parameter values used for service operation invocation.

The PCM captures an abstraction of component behavior through the concept of a Service Effect Specification (SEFF), which has a graphical representation similar to a UML activity diagram (illustrated in Figure 1). It captures the control flow through a component including branches, loops, and external calls to the component’s required interfaces. Branches and loops can be guarded by probability distribution, or conditions based on passed parameters. Aggregation of all SEFF information related to a certain service operation yields the complete execution graph of the operation. For a detailed description of PCM SEFF and related concepts, see [2].

An essential part of PCM models is the PCM Usage Profile specifying (in a probabilistic fashion) input parameter values that then propagate throughout the whole architecture via parametric dependencies included in SEFFs.

From the point of view of reliability modeling, the most relevant parts of the model are the PCM Internal Actions, which abstract the component-internal processing, and hence represent the places where faults are expected to reside and get activated during execution. An Internal Action can generally reflect arbitrarily complex algorithms and processing, as long as it constitutes an internal part of a single software component.

As pointed out in Section 2.2, the faults remaining in a system at run-time are mainly Mandelbugs, which are very difficult to trace back to their grounds and have complex activation patterns, possibly involving supporting software layers. While explicit modeling of these faults is hardly possible due to their nature (and missing information about the circumstances), expert knowledge can be used to associate Internal Actions with a failure probability\(^1\) based on the type of operation it executes. For example, a component developer who knows that an Internal Action \(i\) involves an average of 100 I/O operations and that I/O operations generally fail with probability \(10^{-6}\), may estimate the Internal Action’s failure probability \(f_{p_i} \approx 1 - (1 - 10^{-6})^{100}\).

Assuming that Internal Actions fail independently, and that every failure of an Internal Action causes the currently executed operation to fail, the success probability of an overall service operation is the product of the success probabilities of the Internal Actions passed on the way. That is, if the execution involves a sequence of \(n\) Internal Actions with failure probabilities \(f_{p_1}, f_{p_2}, \ldots, f_{p_n}\), we have:

\[
PS_{sf} = \prod_{i=1}^{n} (1 - f_{p_i})
\]

where \(PS_{sf}\) denotes the probability of success with respect to software faults.

In general, a service operation may have a great number of possible execution paths, depending on the input parameter values used for service invocation, which are probabilistically specified in the PCM Usage Profile. The set of all possible execution paths and their probabilities forms a potentially complex execution graph including control flow constructs like branches and loops. This graph can be transformed into a Discrete-Time Markov Chain (DTMC), which is evaluated to determine the probability of success.

Calculation of the success probability of a service operation as a function of individual failure probabilities along the execution path enables system architects to reason in an assume-guarantee style: If we assume the specified upper bounds on failure probabilities of individual internal actions, we can guarantee the computed lower bound on the overall success probability.

### 3.2. Physical-Resource Breakdowns

Including physical-resource breakdowns into reliability prediction requires capturing resource failure and recovery, as well as the mapping of software components and execution control flow to resources. The PCM already includes the notion of resources and a deployment view that provides the necessary mapping. We extend the model with the notion of resource failure and recovery by associating an MTTF and an MTTR value with each physical resource in the system.

Let \(R = \{r_1, r_2, \ldots, r_n\}\) be the set of physical resources in the system. Each resource \(r_i\) is characterized by its \(MTTF_i\) and \(MTTR_i\) and has two possible states \(OK\) and \(NA\) (not available). Let \(s(r_i)\) be the current state of resource \(r_i\). Then, we have the following probabilities:

---

\(^1\)With the failure probability we mean the probability that a software fault gets activated during execution of the Internal Action and causes the current operation to fail.
Let \( S \) be the set of possible resource-state combinations of the system, that is, \( S = \{s_1, s_2, \ldots, s_m\} \), where each \( s_j \) is a combination of states of all \( n \) resources:

\[
s_j = \{s^j_1, s^j_2, \ldots, s^j_n\} \in \{OK, NA\}^n
\]

As each resource has 2 possible states, there are \( 2^n \) possible state combinations, that is, \( m = 2^n \). Assuming independent resource failures, the probability of the combination of resource states is the product of the individual resource-state probabilities:

\[
P(s_j) = \prod_{i=1}^{n} P(s(r_i) = s^j_i), \ j \in \{1, \ldots, m\}
\]

Thus, for each possible resource-state combination we get a probability that the resources are in this state at an arbitrary point in time. We use this information for combined consideration of resource breakdowns and software faults, as described in the following section.

### 3.3. Combined Failure Modeling

Combining the results of the previous sections, we consider a service execution as failed if any of the involved Internal Actions fails, which can be caused by a software fault or by breakdown of a physical resource used by the Internal Action.

When MTTF and MTTR on the resource level were transformed to the probabilities of being in OK/NA states, we lost the information about the frequency of resource state changes. This shift is, however, necessary to avoid incorporation of time into the reliability model, and keep the model tractable with analytical methods. Assuming that resource failure and repair times are orders of magnitude longer than the duration of a single service execution, we accept the assumption that resource state changes may not take place within a service execution, but only between service executions. Thanks to this assumption, subsequent Internal Actions inside one service are executed under the same resource setting, which is more realistic than considering the possibility of resource-state changes after each action.

Let \( a \) be an Internal Action, and let \( fp_a \) be the failure probability of \( a \). Let \( R_a \subseteq R \) be the set of resources used by \( a \). For each possible resource-state combination \( s_j \), where \( j \in \{1, \ldots, m\} \), \( fp^j_a \) denotes the failure probability of \( a \) while the system is in state \( s_j \). Then, we have:

\[
fp^j_a = \left\{ \begin{array}{ll}
fp_a & \text{iff } \forall r_i \in R_a: \ s^j_i = OK \\
1 & \text{iff } \exists r_i \in R_a: \ s^j_i = NA
\end{array} \right.
\]

Having determined the state-dependent failure probabilities, we calculate \( PS_j := P(SUCCESS|s_j) \) for each \( j \in \{1, \ldots, m\} \) by construction and evaluation of the execution path DTMC, as mentioned in Section 3.1. The overall probability of success is then calculated as a weighted sum over all individual results:

\[
PS = \sum_{j=1}^{m} (PS_j \times P(s_j))
\]

### 4. Example

This section describes a simple example. Though not covering the full capabilities of the approach, it illustrates the main steps involved. The system described here contains two software components \( C_1 \) and \( C_2 \), which provide two interfaces \( I_1 \) and \( I_2 \) with operations \( O_1 \) and \( O_2 \), respectively. Component \( C_1 \) requires \( I_2 \) from \( C_2 \). Figure 2 shows the set up.

![Figure 2. Sample system set up](image)

In the example, we omit consideration of probabilistic control flow with parametric dependencies and just assume that \( O_1 \) includes an Internal Action \( A_1 \) with failure probability \( fp_1 \), as well as a call to \( O_2 \), which in turn includes an Internal Action \( A_2 \) with \( fp_2 \). We suppose that the call to \( O_2 \) as part of \( O_1 \) only happens in 30% of all cases. This probability might be the result of analysing a certain usage profile and its propagation from \( C_1 \) to \( C_2 \) in PCM. We are interested in the probability of success of \( O_1 \).

Construction and evaluation of the execution path DTMC (discussed in Section 3.1) leads to the following probability:

\[
PS_{sf} = 0.7(1 - fp_1) + 0.3(1 - fp_1)(1 - fp_2)
\]

We set the CPUs of the two computing nodes shown in Figure 2 as the resources to consider: \( R = \{CPU_1, CPU_2\} \). Deployment of \( C_1 \) and \( C_2 \) on Node 1 and Node 2 implies that \( A_1 \) uses \( CPU_1 \) and \( A_2 \) uses \( CPU_2 \). While a breakdown of \( CPU_1 \) definitely leads to failure of
$O_1$, a breakdown of $CPU_2$ only leads to failure when $O_2$ is called. For the 4 possible resource-state combinations, Table 1 lists the success probabilities.

<table>
<thead>
<tr>
<th>resource-state combination</th>
<th>probability of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1 = (OK, OK)$</td>
<td>$PS_1 = PS_{sf}$</td>
</tr>
<tr>
<td>$s_2 = (OK, NA)$</td>
<td>$PS_2 = 0.7(1 - f_{p1})$</td>
</tr>
<tr>
<td>$s_3 = (NA, OK)$</td>
<td>$PS_3 = 0$</td>
</tr>
<tr>
<td>$s_4 = (NA, NA)$</td>
<td>$PS_4 = 0$</td>
</tr>
</tbody>
</table>

Table 1. Sample system success probabilities

Finally, the overall probability of success can be computed as the weighted sum over all resource-state combinations:

$$PS = PS_{sf} 	imes P(s_1) + 0.7(1 - f_{p1}) 	imes P(s_2)$$

5. Implementation

The implementation of our approach is based on the existing PCM meta-model and tool chain. The meta-model needed to be extended to care for Internal Action failure probabilities, as well as physical-resource MTTF and MTTR values. A new tool for reliability analysis that involves a transformation into and evaluation of the execution path DTMC is added to the tool chain. In addition, we implemented a reliability simulation, which is a variant of the already existing performance simulation. We use the reliability simulation for validation of our approach.

As the PCM implementation is open source based on the Eclipse platform [7] and the Eclipse Modeling Framework (EMF) [11], new functionality can be added as plug-ins into the existing tool chain. Performance simulation is based on DESMO-J [8] and openArchitectureWare code generation [9]. The reliability simulation is realized using the same technologies.

Currently, the implemented reliability extension of the tool is in a prototype version. Hence it is not yet part of the official release of the tool, but is planned to become part of it in the near future.

6. Conclusions

In this paper, we present an approach to design-time reliability prediction for component-based software systems. We consider software faults and physical-resource breakdowns as possible sources of failure. Our method evaluates the probability that execution of a service operation succeeds.

We utilize the Palladio Component Model (PCM), which captures software component composition, component behavior, system usage and deployment. The system usage profile is propagated throughout the architecture. While so far, the PCM allows for system performance evaluation, we extend the model and the corresponding tools with capabilities for reliability modeling and prediction.

As the next step, we will complete the implementation of the approach, including a reliability simulation for validation purposes, and make it publicly available. Beyond that, further steps will comprise reliability calculations for complete call sequences, as well as consideration of error handling mechanisms. These measures are intended to further improve the generality of the approach, in order to satisfy the increasing demand for authentic software system reliability prediction.

References