An improved framework for evaluating the reliability of component-based software from UML specifications

Rania Elnaggar
Lane Department of Computer Science & Electrical Engineering
West Virginia University
Morgantown, WV 26505
May 2003

1 Abstract

Early assessment of software quality attributes, such as reliability and performance is receiving wide attention in the research community. Specifying software in UML notations gives a boost to these efforts since UML artifacts are often representative of software in the very early stages of the development life cycle, and they continue to be used throughout the development process. In this paper we compare two approaches aimed at estimating reliability of component-based software systems specified in UML. We identify common problems in the discussed approaches and we introduce a framework to overcome these problems by incorporating software performance attributes. We hence serve an overall framework to facilitate both reliability and performance analysis.

2 Introduction

2.1 Overview

The objective of this paper is to address improvements to software reliability estimation techniques that are used early in the software life cycle, namely in the design phase. We focus on component-based software artifacts that are specified in UML notations. We mainly discuss and compare two approaches to component-based software reliability assessment, and hence identify common problems in these approaches. Accordingly, we introduce an improvement to these techniques that is based on incorporating software performance attributes. Both reliability and performance are classified as software non-functional requirements, and are among the most important software quality attributes.

We also evaluate software reliability within the context of the overall system reliability, and we suggest an improvement to computing overall system reliability by incorporating the reliability of the operating system.

This work is organized as follows. In section 2.2 we discuss and classify software non-functional requirements. Section 2.3, we give an overview of the unified modeling language (UML). In section 2.4, we briefly highlight component-based software development. In Section 3 we investigate software reliability within the context of overall system reliability. In section 4 we discuss and compare two approaches to early software reliability assessment. In section 5 we outline common problems in the two approaches...
and we propose a solution to overcome these problems. Finally, we conclude the paper in section 6.

2.2 Software Non-Functional Requirements

While Non-functional Requirements (NFRs) have great impact on software quality and can cause the failure of the software after it is deployed, they are always set aside in the software development process that focuses mainly on the functional requirements.

It has come to recognition the need to be able to systematically deal with NFRs and incorporate them in the development process. In [CHUNG95], the authors introduce a framework to incorporate dealing with NFRs in the software development process. That approach is classified as process oriented as opposed to product-oriented, where the quality judgment is done after a product is at hand. In [CYSN02], the authors extend that framework to specifically focus on tackling NFRs within the context of UML models. NFRs are elicited and incorporated into UML use cases and scenarios during the software specification phase.

NFRs can be classified into three main categories: Dependability, Timeliness and others. This classification is illustrated in Figure 1.

![Figure 1: Classification of NFRs](image)

2.2.1 Software Dependability

In [LAPR93] and [AVIZ01] a concept of software dependability is formulated and introduced. According to the authors, dependability consists of three parts: the threats to the system, the attributes of the system, and the means by which dependability is achieved. Dependability of a system is the ability to deliver service that can justifiably be trusted. We here list and define dependability attributes according to [KNIGHT02] and [AVIZ01]:

- **Availability**: readiness for correct service.
- **Reliability**: continuity of correct service
- **Safety**: absence of catastrophic consequences on the user(s) and the environment;
- **Confidentiality**: absence of unauthorized disclosure of information.
- **Integrity**: absence of improper system state alterations
- **Maintainability**: ability to undergo repairs and modifications.

Security is the concurrent existence of a) availability for authorized users only, b) confidentiality, and c) integrity with ‘improper’ meaning ‘unauthorized’.

It is important to note that the threat to system dependability results from the existence of faults. The activation of a faulty part results in an error, which may cause system failure. A system consists of a set of interacting components; therefore the system state is the set of its component states. A fault originally causes an error within the state of one (or more) components, but system failure will not occur as long as the error does not reach the service interface of the system.

Since this work mainly focuses on software reliability, we hence introduce a more elaborate definition for that term; software reliability which can be defined as *the measurement and probability that the software will perform its intended function (according to specifications) without error for a given period of time* [SHOOM02].

### 2.2.2 Software Timeliness

Timeliness NFRs are mainly concerned with the ability of the system to “perform” within its desired time limits. Performance involves two parameters: *responsiveness* and *scalability*. Responsiveness is usually defined in terms of either response time or throughput; Response time is the time required by the system to respond to a request, while throughput is the number of requests that the system can process in a specified time interval. Scalability is the ability of the system to continue to meet its timeliness objectives as the demand (requests) for service increases [SMITH01].

Performance is an essential quality attribute of every software system. Experience shows that performance problems are often introduced early in the software life cycle, and hence, early measures to alleviate these problems should be taken into consideration.

Performance analysis is used to verify that the software system will meet its performance objective under the foreseen workload. To conduct performance analysis, we have to specify the system’s deployment environment and accordingly identify an array of resources. Resource demands are then identified and attached to processing steps. For details about performance analysis, please refer to [SMITH01].

### 2.2.3 Other NFRs

Under this category, we classify all other NFR attributes that do not fall under the previous two categories. These could be very few, and could possibly grow as new quality attributes are introduced. This category includes Cost of the system, and Usability of the system.

### 2.3 UML

The Unified Modeling Language (UML) is a general purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system.
UML is rapidly becoming the standard for software development. UML provides a standard object-oriented modeling notation that represents the software through a set of graphical diagrams. Each type of diagrams provides a specific “view” of the software being modeled. It is completely up to the developer to decide upon which diagrams to consider, and hence, UML provides flexibility that is geared towards achieving different objectives. This flexibility, graphical notation, and the concept of providing “views”, are the main reasons that UML is becoming so popular. For information about UML specifications, refer to [BOOCH99] [RAMB99].

UML provides means to model both system architecture and system behavior in an integrated fashion. Moreover, the same set of diagrams supplied by UML can represent the system throughout the stages of the development cycle. That is in contrast to other modeling formalisms where clear distinction is made between different development stages, and accordingly different sets of diagrams are used.

2.4 Component-based Software Development

Component-based software development is an approach to assemble software components in a software architecture that satisfies its respective specifications. Components need to interact with each other to perform the required overall functionality. Commercial-Of-The-Shelf components are often used in this development approach.

As the emphasis of software reuse continues to grow, component-based software development is gaining more popularity. However, this trend poses challenges to traditional software reliability assessment method; and new approaches need to address the specific nature of component-based software development. These new approaches aim at assessing the reliability of the software system from the individual reliabilities of its components [LYU96]. That is as opposed to treating the entire system as a single entity.

3 Evaluating Overall System Reliability

The reliability of a computerized system is dependant on the individual reliabilities of all its components. In the literature, a system is often defined to have three main elements: hardware, software and human operator. If the failure in each of these components is independent on the others, then the reliability of the system can be expressed as [SHOOM02]:

\[ R_{sys} = R_H \times R_S \times R_O \]

Where,
\[ R_{sys} \] is the system reliability
\[ R_H \] is the hardware reliability
\[ R_S \] is the software reliability
and \[ R_O \] is the reliability of the human operator

There is a clear distinction between the methods adopted to estimate the reliabilities of each of the systems elements. That is due to the different nature of failure for each of these elements. The dominant mode of hardware failures is due to equipment wear out. While software code doesn’t wear out, software failures are often due to latent discovery
of specification, design and coding errors. However, software faults can be eliminated by exhaustive testing, which is impractical (if not impossible) even for very small programs.

In this context, we distinguish a system component that is often ignored in the literature; that is the operating system. Software components are almost never deployed directly on the system hardware; instead there is always this intermediate layer, the operating system, which separates software applications from hardware. This 3-tier architecture is illustrated in Figure 2.

![Three-tier approach](image)

Figure 2: Three-tier approach

We recognize the importance of incorporating the reliability of the operating system in evaluating the overall system reliability. In fact, operating systems are constantly increasing in complexity, and hence become increasingly fault-prone. In [SPIN99], the author introduces a rough measure of operating systems complexity by measuring the number of supported system calls. Table 1 (which is adopted from [SPIN99]) illustrates the increasing number of system calls for a number of well operating systems over a period of 27 years.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Year</th>
<th>Number of System calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Edition Unix</td>
<td>1971</td>
<td>33</td>
</tr>
<tr>
<td>7th Edition Unix</td>
<td>1979</td>
<td>47</td>
</tr>
<tr>
<td>SunOS 4.1</td>
<td>1989</td>
<td>171</td>
</tr>
<tr>
<td>4.3 BSD Net 2</td>
<td>1991</td>
<td>136</td>
</tr>
<tr>
<td>HP-UX 9.05</td>
<td>1992</td>
<td>219</td>
</tr>
<tr>
<td>SunOS 5.4</td>
<td>1994</td>
<td>163</td>
</tr>
<tr>
<td>Linux 1.2</td>
<td>1996</td>
<td>211</td>
</tr>
<tr>
<td>SunOS 5.6</td>
<td>1997</td>
<td>190</td>
</tr>
<tr>
<td>Linux 2.0</td>
<td>1998</td>
<td>229</td>
</tr>
<tr>
<td>Windows platform SDK</td>
<td>1998</td>
<td>3433</td>
</tr>
</tbody>
</table>

Supported by the above discussion, we hence propose incorporating the reliability of the operating system $R_{OS}$ into the system reliability equation. Assuming all system elements are independent, then system reliability $R_{SYS}$ can be evaluated as follows:
\[ R_{sys} = R_H \times R_{OS} \times R_S \times R_O \]

In this paper we mainly focus on evaluating software reliability \( R_S \) in the early stages of the software development process.

## 4 Comparing two reliability assessment approaches

In this section we describe two approaches to reliability estimation of component-based software specified in UML terms. We then compare and critique the two approaches outlining some weaknesses that need to be alleviated.

### 4.1 Yacoub et al.

In [YAC99], the authors propose a methodology for evaluating the reliability of component-based software specifications specified in UML, and they apply that technique on a self-serve car wash case study. The proposed technique is called Scenario-Based Reliability analysis (SARBA) and is applicable at the high-level design stage.

This technique makes extensive use of component interactions specified in terms of UML sequence diagrams. Sequence diagrams are used to then construct a probabilistic model of the interaction scenarios called Component Dependency Graph (CDG). We here outline the parameters used in the SARBA technique as follows:

- Software specifications are given in terms of UML. Specifically, sequence diagrams.
- Each scenario (sequence diagram) has a probability of execution \( P_{S_k} \) (for the \( k \)th scenario), such that \( \sum_{k=1}^{n} P_{S_k} = 1 \) for all the scenarios of the application.
- Components Reliability \( R_{Ci} \) (for the \( i \)th component) is assumed to be known. The authors assume the reliability of COTS components can be readily available. They introduce a technique to measure the reliability of the components based on their dynamic complexity metrics in further work [YAC00].
- Reliabilities of transitions between components during the course of the execution is assumed to be known. \( R_{T_{ij}} \) is defined as the reliability of the transition from the source component \( i \) to the destination component \( j \). Transition reliability is defined to be a product of interface reliability and connector (or link) reliability.
- Average execution time of the application is calculated by summing up (over all scenarios) the average execution time per scenario (which is assumed to be known) multiplied by the probability of the scenario.
- Average execution time of a component is calculated by summing up (over all scenarios) component execution time per scenario (which is estimated from active time) multiplied by the probability of the scenario.
- Transition probability between two components is estimated using the number of interactions between the two components in each scenario.
- The above parameters are used to construct the CDG which is then consumed by an algorithm that estimates the system reliability.
4.2 Cortellessa et al.

In [SINGH01] and [CORT02] respectively, the authors introduce another approach to the reliability assessment of component-based UML-specified software models. The technique first accounted only for component reliabilities in [SINGH01] and then incorporated connector reliabilities in [CORT02]. This approach estimates a Bayesian reliability that is then extended after test data becomes available. However, we restrict our analysis here to the early estimation of the software reliability. Similar to SARBA, both component and connector reliabilities are assumed to be known.

This technique makes use of three types UML diagrams: Sequence diagrams that depict interactions between components and are annotated with probabilities of execution of each scenario, Deployment diagrams that map components to connected sites and are annotated with connector reliabilities\(^1\), and use case diagrams that depict utilization of the system and are annotated with frequency of execution of use cases.

We outline the steps for the application of this methodology as follows:

- The software specification is given in UML terms, specifically in term of: use case diagram (annotated with probability of execution of each use case), sequence diagrams representing each scenario, and deployment diagram (annotated with connector reliability).
- Although the methodology makes use of component reliability and scenario probability (both are assumed given), this information is not annotated on the UML diagrams.
- The probability of execution of a given scenario is the product of the probability of its use case multiplied by the probability of executing that particular scenario within the use case.
- The number of busy periods for a component in a given scenario is used to estimate the failure probability of that component within the given scenario.
- The system reliability is obtained in a combinatorial manner from the: probability of failure of each component within a scenario, the connector reliability of the component interaction within the scenario, and the probability of execution of the scenario.
- The resulting equation (which is generally intractable) is solved by using simulation.

4.3 Comparing the two approaches

- Both the two approaches assume that an estimation of component and connector reliabilities is available.
- Both of the two approaches assume the independence of component failures. Moreover, they both assume that component reliability is the same in each interaction in every scenario.

\(^1\) Components on the same site are assumed to have perfect connection (connector reliability = 1), and hence their respective connector reliabilities are ignored.
In [SINGH01] and [CORT01], the proposed methodology separates the frequency of execution of use cases from that of the scenarios under each use case. That structure is better suited to UML constructs than the SARBA in [YAC99], where no distinction is made between the frequency of execution of a use case and frequency of execution of its child scenarios. Making the distinction between the two frequencies is important to better utilize operational profile data.

In [YAC99] the methodology measures the component execution time in a given scenario by measuring the sum of its active time along the life line during the scenario (length of vertical rectangles along the lifeline). In [SINGH01] and [CORT01], they estimate the component active time by counting the “number” of its active intervals along the lifeline during a given scenario (the count of the vertical rectangles along the lifeline).

In [YAC99], it is necessary to build a Component Dependency Graph (CDG) that is then consumed by a simple algorithm to get the reliability estimation. In [SINGH01] and [CORT01] building the reliability equation is simple, however, the resulting equation is generally intractable and they do solve it by simulation using Mathematica. No conclusion can be made here about which approach is better in terms of time or effort needed to obtain the solution. Neither can we compare the reliability results since they are both applied on two different case studies.

5 Improvements to Reliability Assessment Techniques

From the discussion in section 4 above, we mainly outline these common weaknesses in the two approaches:

- We consider the assumption that the component execution times can be estimated from the length of the active periods in the sequence diagram [YAC99] to be very inaccurate. The length of the active periods (represented as rectangles) is a graphical representation that doesn’t necessarily correspond to the actual length of execution.

- Also, estimating the component execution time from the number of the busy periods (number of rectangles) in a component lifeline, is an imprecise approximation. Busy periods can have very big variations in the actual execution time they correspond to.

- Both approaches assumed that “all” the possible scenarios are depicted as sequence diagrams. In reality, sequence diagrams fall short in effectively describing scenarios because of the following characteristics:
  - Hierarchical scenarios can not be represented in sequence diagrams; they have to be specified as separate individual scenarios. That renders representing all possible scenarios as sequence diagrams very impractical. Instead, only “key” descriptive scenarios are often represented as sequence diagrams.
  - Concurrent forks and joins can not be represented in sequence diagrams since sequence diagrams are sequential by definition.
  - It is very tedious to represent loops in sequence diagrams since all the loop iterations must be individually specified in the diagram.
Following, is our proposed solutions to improve these two approaches.

5.1 Estimating Component Execution Times

If a component becomes active during the course of a scenario, then it is executing a certain function/method/algorithm. We call each active interval in a component lifeline a processing step. Since the deployment environment of the software is assumed to be known at this stage of the development process, then we can identify an array of resources (typically hardware) that the software uses. Resource demand estimates can be made at each processing step. By multiplying the resource demand for each resource by its respective service time, then the execution time for each processing step can be accurately estimated (and so is the components execution time). Let’s assume we have the following resources deployed: CPU, Disk, and Network. At each processing step we need to estimate the demand for each of the resources respectively. It is important to note that the choice of the demand “units” is important to simplify the resource demand assignment.

If we annotate UML diagrams with resource demand estimations at each processing step, then we are serving two important purposes:

- Facilitating the prediction of an accurate system reliability.
- Facilitating the performance analysis of the software specification.

This contributes to an integrated framework to quantitatively assess software quality attributes that are often expressed as non-functional requirements.

We however make note that sequence diagrams are not the best available UML notation to incorporate such annotation. Scenarios in UML can be specified in term of Activity Graphs which are more suitable to reliability and performance analyses. In activity graphs, the focus is on processing steps, rather than on messages exchanged between components.

Figure 3 shows an example for an annotated activity graph that represents a scenario interaction between three components. Each component has a “swim lane” instead of a life line. Busy periods are represented as activities (or processing steps). Each activity can have sub-activities (be a composite activity) or a state diagram, hence, activities can be as granular as we want them to be. Component interactions are shown by arrows transferring the activity execution from one swim lane to another. In the example shown in Figure 3, we can see resource demands associated with each activity.
5.2 Effectively Representing Scenarios

As we discussed earlier in this section, sequence diagrams have three major drawbacks when it comes to effectively representing the entire set of scenarios of an application. On the other hand, Activity diagrams have all the muscle to represent different execution paths, loops, and hierarchical scenarios. Moreover, these different execution path can be annotated with their respective probabilities of execution. Also, loops can be annotated with their average number of executions. Examples of these annotations are shown in Figure 3. Note that the entire scenario presented in the diagram is annotated with its execution probability before the start of the first activity.

Activity diagrams are as standard UML notation as sequence diagrams are; unfortunately, they are less known and used despite their capabilities.
6 Conclusion
In this paper we presented a suggested improvement to calculating the overall system reliability by incorporating the reliability of the operating systems. We showed that operating system reliabilities are becoming more crucial to consider as they continue to grow in complexity.

We compared two approaches that aim at evaluating software reliability for component-based software specified in UML terms. Both the two approaches provide means to estimate the software reliability at an early stage of the development life cycle, namely the design stage. Based on our comparison we identified two common weaknesses in both approaches: estimating component execution times and representation of scenarios. We then proposed an improvement to the formerly introduced methodologies that is based on incorporating performance aspects of the software into the reliability model. This serves the objective of having a single framework to evaluate software quality attributes as early as possible in the software life cycle.

7 References


