Abstract

Software reliability models can provide quantitative measures of the reliability of software systems during development processes. Research activities in software reliability engineering are conducted over the past four decades, and many software reliability models are proposed. In this paper we will present our results in predicting the reliability of software and how that relates to the reliability of hardware.

1. Introduction

Today, almost everyone in the world is directly or indirectly affected by electronic systems [1]. They are used in diverse areas for various applications including air traffic control, nuclear reactors, aircraft, industrial process control, automotive mechanical and safety control, and hospital health care, affecting many millions of people. As the functionality of computer operations becomes more essential and yet more complicated and critical applications increase in size and complexity, there is a great need for looking at ways to quantify and predict the reliability of such systems in various complex operating environments [1]. A (complex) system is a set of interacting or interdependent components forming an integrated whole. This implicates that two components together already form a system. When the number of components and their interactions hugely increase, so-called large or complex systems are formed. The types of components, their quantities, their qualities and the manner in which they are arranged within the system have a direct effect on the system's reliability. The commonly used description for system reliability [2] is given as:

The probability that a system, including all hardware, firmware, software, and their interactions will satisfactorily perform the task for which it was designed or intended, for a specified time and in a specified environment.

From a system reliability point of view, the challenge is to master the reliability of all these components. Figure 1 shows two possible lighting applications where we see a large penetration of software enabled controls.

Figure 1: Lighting applications with software enabled controls.

Software failures are a primary cause of product reliability problems. Unlike hardware failures, software failures are not caused by faulty components, wear-out or physical environment stresses such as temperature and vibration. Software failures are caused by latent software defects that were introduced into the software as it was being developed, but were not detected and removed before the software was released to customers. The best approach to achieving higher software reliability is to reduce the likelihood that latent defects are in released software. Unfortunately, even with the most highly-skilled software engineers following industry best practices, the introduction of software defects is inevitable due to the inherent complexities of the software functionality and its execution environment. As a system is built out of sub-systems, which each consist out of modules, the system testing approach will focus on verification of the system and its interfaces to requirements. Often the V-model is used for product development. This model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing. The horizontal and vertical axes represent time or project completeness (left-to-right) and level of abstraction, as depicted in Figure 2. In this approach, the software can be considered as a component as well and, thus, reliability of this component needs to be considered. In this paper we will present our results in predicting the reliability of software and how that relates to the reliability of hardware. We will present a use case in which we will demonstrate the use of these so-called software reliability models.

2. Software versus Hardware Reliability

Software reliability or robustness is the probability of failure-free software operation for a specified period of
time in a specified environment. Software failures may be due to errors, ambiguities, oversights or misinterpretation of the specification that the software is supposed to satisfy, carelessness or incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software or other unforeseen problems. Software reliability is not a function of time. There is not something as software ‘wear-out’; software will not change in time. Software reliability relates to errors that are induced by circumstances or contexts that are unforeseen/not addressed in the design phase. Typical questions that need to be addressed are:

- How many errors are left?
- What is the probability of no failures in a given time period?
- What is the expected time until the next failure?
- What is the expected number of errors in a given time period?

The development of hardware reliability theory has a long history and got established to improve hardware reliability greatly [4, 5]. The history of reliability as we know it now goes back to the 1950s, when electronics played a major role for the first time. It may seem strange today but at that time there was considerable resistance to recognizing the stochastic nature of the time to failure, and hence reliability. Software systems do not degrade over time unless modified. There are many differences between the reliability and testing concepts and techniques of hardware and software. Therefore, a comparison of software and hardware reliability would be useful in developing software reliability modeling. Table 1 shows the differences and similarities between the two.

<table>
<thead>
<tr>
<th>Hardware reliability</th>
<th>Software reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure rate has a bathtub curve</td>
<td>Without considering program evolution, failure rate is statistically non-increasing</td>
</tr>
<tr>
<td>Material deterioration can cause failures even though the system is not used</td>
<td>Failures never occur if the software is not used</td>
</tr>
<tr>
<td>Failure data are fitted to some distributions. The selection of the underlying distribution is based on the analysis of failure data and experiences. Emphasis is placed on analyzing failure data</td>
<td>Most models are analytically derived from assumptions. Emphasis is on developing the model, the interpretation of the model assumptions, and the physical meaning of the parameters</td>
</tr>
<tr>
<td>Failures are caused by material deterioration, random failures, design errors, misuse, and environment</td>
<td>Failures are caused by incorrect logic, incorrect statements, or incorrect input data. This is similar to design errors of a complex hardware system</td>
</tr>
<tr>
<td>Hardware reliability can be improved by better design, better material, applying redundancy and accelerated life testing</td>
<td>Software reliability can be improved by increasing the testing effort and by correcting detected faults</td>
</tr>
<tr>
<td>Hardware repairs restore the original condition</td>
<td>Software repairs establish a new piece of software</td>
</tr>
<tr>
<td>Hardware failures are usually preceded by warnings</td>
<td>Software failures are rarely preceded by warnings</td>
</tr>
</tbody>
</table>

### 3. Software Reliability Testing and Modeling

Software reliability engineering [6, 7] is a field of software development that relates to testing and modelling the software ability to function (or not), given environmental conditions, for a particular amount of time. No method of development can guarantee totally reliable software. A set of statistical modelling techniques are required that:

- Enable the achieved reliability to be assessed or predicted
- Is based on observation of system failures during system testing and operational use

It uses general reliability theory, but is much more than that:

- How reliable is the program/component now?
- Based on the current reliability of the software: can we accept it or should we reject it?
- Based on the current reliability of the hard-software system: can we stop testing and start shipping?
- How reliable will the system be, if we continue testing for some time?
- When will the reliability objective be achieved?
- How many failures will occur in the field (and when)?

Seven distinct steps can be marked in the software reliability engineering process (see also Figure 3):

1. **Define reliability objective:** express failure intensity as failures per natural unit (such as failures / Kpages printed, failures / Ktransactions, failures / Kcalls, etc). Per severity level, the objective may be established by contractual, warranty, or regulatory requirements.

2. **Expected system usage:** a complete set of operations with their probabilities of occurrences that represent ‘field conditions’:
   - Gives information on how users will employ the product we are building so that we can focus on our development and test resources.
   - Model how users will employ the software: environment, type of installation, distribution of inputs over input space.
   - Indicate the relative usage of program modules. You can construct this table from the operational profile and the operation-usage matrix. The operation-module matrix indicates which operations use which modules.
   - According to the usage model, test cases are selected randomly.

3. **Prepare test cases**:
   - Define test scheme, define what to test, for example:
     - Functionality test of each function
     - User interface tests
     - File input / output tests
     - Determine how many tests, for example
     - Create tests

4. **Execute test - collect failure data and severity levels**.
This form of testing has the advantage of testing more intensively the system functions that will be used the most.

Hence we differentiate testing that aims at finding defects (verification, α-tests) and testing whose purpose is reliability assessment (validation, β-test).

5. Reliability Growth Modeling, model-specific assumptions:
   - The system test reflects the intended usage of the system.
   - The failures are mutually independent.
   - The number of failures detected at any time is proportional to the current number of faults in the program.
   - Each time a failure occurs, the fault which caused it is immediately removed, and no new faults are introduced.
   - The (cumulative) number of failures by time follows a Poisson Process (NHPP) or General Order Statistics (GOS) model.

   Data Requirements
   - Actual times when failures occurred or failure time intervals (test execution time), and severity levels (critical / major / average / minor).

6. Projection to field life:
   - Once the model parameters are estimated, it can be used to predict failure intensity in the future, not only to estimate its current value. From this, we can plan how much additional testing is likely to be needed.
   - The model allows for the realistic situation where fault correction is not perfect (infinite number of failures at infinite time).
   - When faults stop being corrected, the model reduces to a homogeneous Poisson process.

7. Monitor field performance
   - Logging and monitoring schemes are required to track the field performance of the system / software.

Software reliability growth models (SRGMs) enable the project leader / test manager to estimate the software reliability and the number of errors remaining in the software. This can help him to decide whether or not the code is suitable for customer use and how much more testing is required if it is not ready yet to release the software. It also provides an estimate of the number of faults that users will encounter when operating the software. These estimates also help to define the appropriate levels of support that will be required for fault correction after the software has been released.

During the testing phase of software development, faults are removed after they are detected. This reduces the number of total faults in the software and the fault-detection rate should decrease as more code is covered. In other words, the length of intervals between fault discoveries should increase. When the fault-detection rate reaches an acceptably low level, the software is deemed suitable to release to customers.

Software reliability growth models (SRGM) are mathematical functions that describe fault-detection and removal phenomenon. Some realistic issues such as imperfect debugging and learning phenomenon of software developers have been studied and incorporated in software reliability assessment. Among all SRGMs, a large class of stochastic reliability models is based on a Non-Homogeneous Poisson Process. These models are known as NHPP reliability models and have been widely used to track reliability improvement during software testing. Another popular class is the class of General Order Statistics, or GOS models. These two classes will be described in next sections.

**Background NHPP [8, 9]**

Software testing process has been widely modelled as a failure counting process. A counting process \{N(t), t ≥ 0\} is said to be a non-homogeneous Poisson process with intensity function \(\lambda(t)\), if \(N(t)\) follows a Poisson distribution with mean value function \(m(t)\), i.e.,

\[ P\{N(t) = k\} = \frac{m(t)^k}{k!} e^{-m(t)}, k = 0, 1, 2, \ldots \]

The mean value function \(m(t)\), which is the expected number of failures experienced up to a certain time \(t\), can be expressed in terms of failure rate of the program, i.e.,

\[ m(t) = \int_0^t \lambda(s)ds, \]

where \(\lambda(s)\) is the failure intensity function. Software reliability \(R(x|t)\) is defined as the probability that no software failure is detected in the time interval \([t, t+x]\), given that the last failure occurred at testing time \(t \geq 0\), \(x > 0\). That is, \(R(x|t) = e^{-(m(t+x)-m(t))}\)

For special cases, when \(t=0\) then \(R(x|0)e^{-mx(x)}\), and \(t=\infty\) then \(R(x|\infty)=1\).

**Background GOS [10]**

In this section we describe an important class of software reliability growth models known as General Order Statistics (GOS). The main assumption for this class of models is that the times between failures of a

---

2014 15th International Conference on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems, EuroSimE 2014
software system can be defined as the differences between two consecutive order statistics. It is assumed that the initial number of failures, denoted by \( N \), is unknown but fixed and finite. Thus, for any \( n \leq N \), we can interpret the first \( n \) failure times \( T_1 < T_2 < \ldots < T_n \) as the first \( n \) order statistics. The times between failures are defined as the difference of two order statistics, i.e., \( X_i = T_{i+1} - T_i \) for all \( i \geq 1 \). In general, the times between failures of GOS models are not independent nor identically distributed. It can be proven that the random variable \( N(t) \) follows a binomial distribution. For that reason the class of GOS models is often called the class of binomial distributions. Different GOS models arise when one considers different distributions for the failure times. The most well-known GOS model is based on the exponential distribution. Many existing software reliability models are variants or extensions of this basic model. Other popular GOS models consider the Weibull and Pareto distribution for the order statistics.

**Data analysis [11, 12]**

It is highly recommendable to carefully look at the data before starting any kind of statistical analysis. For example, it is possible to gain some understanding about the nature of the process being studied simply by plotting the data as a function of time. Figure 4 shows a plot of the failure times against the cumulative number of observed failures.

---

**The Military Handbook Test**

This test performs well at finding significance when the choice is between no trend and a NHPP Power Law (Duane) model. In other words, if the failure process follows the Power Law, this test will generally do better than any other test in terms of finding significance.

Suppose we have \( r \) failure times \( T_1, T_2, T_3, \ldots, T_r \) with the observation period ending at time \( T_{end} \). Calculate

\[
\chi^2_{2r} = 2 \sum_{i=1}^{r} \ln \left( \frac{T_{end}}{T_i} \right)
\]

and compare this to percentiles of the chi-square distribution with \( 2r \) degrees of freedom. For a one-sided improvement test, reject no trend (or HPP) in favour of an improvement trend if the chi square value is beyond the 90 (or 95, or 99) percentile.

**The Laplace Test**

This test performs well at finding significance when the choice is between no trend and a NHPP exponential model. In other words, if the data come from a failure process following the Exponential Law, this test will generally do better than any test in terms of finding significance. As before, we have \( r \) failure times \( T_1, T_2, T_3, \ldots, T_r \) with the observation period ending at time \( T_{end} \).

Calculate

\[
z = \frac{\sqrt{2r} \sum_{i=1}^{r} \left( T_i - T_{end} \right)}{rT_{end}}
\]

and compare this to percentiles of the standard normal distribution. The interpretation of the test statistic is the following: for small values of the test statistic the null hypothesis of HPP is rejected in favour of reliability growth.

**Model type selection [12, 13]**

The main problem that we find when trying to select a suitable model for a specific problem is that there are no general rules to select a model. Although it is possible to find a large variety of lists of assumptions and data requirements for software reliability models in the literature, this is often far from facilitating model selection. For example, there is no universal agreement in the literature on the list of assumptions for certain well-known models. Therefore we propose the following procedure:

1. Select the assumptions which are relevant to the testing data at hand, and make a subset of applicable models.
2. Fit the data to several growth models, and take the one that best fits the data. There are two criteria that are often used for comparison of goodness-of-fit:

   - Mean Square Error (MSE) = \( \frac{1}{n-k} \sum_{i=k}^{a} (m(t_i) - y_i)^2 \),
   - \( m(t_i) \) is the estimated cumulative number of failures at time \( t_i \).

---

*Image: Figure 4: Failure times vs. cumulative number of observed failures*
t, for i=1,2,...,n, and k is the number parameters in the model.

- Akaike’s Information Criterion (AIC) = -2log(max. likelihood value) + 2k. The AIC measures the ability of a model to maximize the likelihood function that is directly related to the degrees of freedom during fitting, increasing the number of parameters will usually result in a better fit. AIC criterion takes the degree of freedom into consideration by assigning a model with more parameters a larger penalty.

Estimation of parameters [14 – 17]

Once the analytical solution for m(t) is known for a given model, the parameters need to be determined. Estimation is achieved by applying the technique of Maximum Likelihood Estimation (MLE). This is the most widely used estimation technique. In many cases, the maximum likelihood estimators are consistent and asymptotically normally distributed as the sample size increases. We refer to [1, 14 - 17] for further details.

Goel-Okumoto

This is the most well-known NHPP model. Due to the important role that this model has played on the software reliability modelling history, it is often called “the” NHPP model. It is based on the following assumptions:

- All failures in a program are mutually independent from the failure detection point of view.
- The number of failures detected at any time is proportional to the current number of failures in a program.
- The isolated failures are removed prior to future test occasions.
- Each time a software failure occurs, the software error which caused it is immediately removed, and no new errors are introduced.

The mean-value function is given by:

\[ m(t) = a(1 - e^{-bt}), \]

for all \( t \geq 0 \), where \( a > 0 \) and \( b > 0 \). The parameter \( a \) is the expected number of failures to be eventually detected while \( b \) is the rate at which each individual failure will be detected during testing.

Musa-Okamoto

Musa-Okamoto observed that the reduction in failure rate resulting from repair action following early failures are often greater because they tend to the most frequently occurring once. This property has been taken into account in the model. The mean value function is given by:

\[ m(t) = a \ln(1 + bt), a > 0, b > 0 \]

where \( a \) is the expected total number of failures to be detected, and \( b \) is the detection rate.

Yamada S-shaped

The so-called S-shaped model was presented in Yamada and Osaki (1984). It receives the name S-shaped because the curve of the mean-value function is often S-shaped (in comparison with the exponential-shaped mean-value function of the Goel-Okumoto model). The mean-value function is given by:

\[ m(t) = a(1 - (1 + bt)e^{-bt}), \]

for all \( t \geq 0 \), where \( a > 0 \) and \( b > 0 \). The parameters of the model have the same interpretation as in the Goel-Okumoto model.

NHPP Imperfect Debugging models

Many existing models describe perfect debugging in previous section, that is, \( a(t) = a \) and where the error detection rate \( b(t) \) function is time-dependent. In this section, we discuss several software reliability models with imperfect debugging processes. The NHPP imperfect debugging model is based on the following assumptions:

- When detected errors are removed, it is possible to introduce new errors.
- The probability of finding an error in a program is proportional to the number of remaining errors in the program.

There are 2 types of the Yamada imperfect debugging model:

1. \[ m(t) = \frac{ab}{b + a}(e^{at} - e^{-bt}), \]

   for all \( t \geq 0 \), where \( a > 0, b > 0, \) and \( \alpha \geq 0 \).

2. \[ m(t) = a\left(1 - e^{-bt}\right)\left(1 - \frac{\alpha}{b}\right) + a\alpha t, \]

   for all \( t \geq 0 \), where \( a > 0, b > 0, \) and \( \alpha \geq 0 \).

In the same analogy the Pham-Nordmann-Zhang (PNZ) model is defined:

\[ m(t) = \frac{a}{1 + \beta e^{-bt}}\left(\left(1 - e^{-bt}\right)\left(1 - \alpha / b\right) + a\alpha t\right), \]

for all \( t \geq 0 \), where \( a > 0, b > 0, \alpha > 0, \) and \( \beta > 0 \). PNZ is a so-called imperfect debugging S-shaped model, where \( a \) is the expected total number of failures, \( \alpha \) is the constant failure introduction rate, \( b \) and \( \beta \) are constants.

GOS model: Jelinski-Moranda

The assumptions in this model include the following:

- The program contains \( N \) initial failures which is an unknown but fixed.
- Each failure in the program is independent and equally likely to cause a failure during a test.
- Time intervals between occurrences of failure are independent of each other.
- Whenever a failure occurs, a corresponding failure is removed with certainty.
- The failure that causes a failure is assumed to be instantaneously removed, and no new failures are inserted during the removal of the detected failure.
- The software failure rate during a failure interval is constant and is proportional to the number of failures remaining in the program.
The program failure rate at the \(i\)th failure interval is given by:

\[
\lambda(t_i) = \phi(N - (i - 1) \theta), \quad i = 1, 2, \ldots, N,
\]

where \(\phi\) is a proportional constant, i.e., the contribution of any one failure makes to the overall software. Hence, the software reliability function equals:

\[
R(t_i) = \exp(-\phi(N - i \theta) t_i)
\]

### 4. Use Case: Visual Basic Software

A visual basic application developed to evaluate product designs is taken as our use case. The features of the software are:

- About 13500 codes lines
- Divided over 29 modules
- Dedicate GUI that interacts with the users

As a rule of thumb, research study has shown that professional programmers implement on average six software defects for every 1000 lines of code (LOC) written [1]. Following this number, then the case software would encompass \(6 \times \frac{13500}{1000} = 81\) defects or failures.

In our case, the software itself is developed according to the flowchart given in Figure 3. In total, 3 \(\alpha\)-test phases are executed and one \(\beta\)-test phase. Alpha testing is performed by experienced users at the developers’ site. Alpha testing is a form of internal acceptance testing, before the software goes to beta testing. Beta testing comes after alpha testing and can be considered as a form of external user acceptance testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. After the \(\beta\)-test the software was released to groups of people so that further testing can ensure the product has few failures.

The reliability objective of the software was set at 1 critical or major failure per 100 user sessions. The expected usage of the software was derived by determining both the users and their typical behavior with the software. Based on this operational profile, we developed more than 100 functionality tests for the first \(\alpha\) phase. This test plan comprises the following consecutive parts:

1. Functionality & load tests
2. Security tests
3. User tests

The way of working for both the \(\alpha\)- and \(\beta\)-tests is:

- Run the software
- Record the starting time
- Report any failure or event that occurs, including its level of severity
- Record the test time that was needed to detect the failure or event
- Close the software
- Repeat the above

For the testing itself, a scoring form is used. It is very important that this form is filled in a good manner and all details are very precisely logged. Screen shots of the failures need to be included. The ranking of the failures is done a priori by the tester and evaluated afterwards. In this evaluation, both the \(\alpha\) & \(\beta\) testers and the software developer need to agree on the ranking. As given in the test release plan, four different ranks are used:

1. Critical
2. Major
3. Average
4. Minor

The explanation of these codes is given in Table 2. A typical response from a tester is shown in Figure 5. An overview of all the results is given in Table 3.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Description</th>
<th>Example Event</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>The defect causes the software to stop functioning</td>
<td>Incorrect output when using a specific input</td>
<td>C1</td>
</tr>
<tr>
<td>Major</td>
<td>The defect does not result in the failure of the entire system, but it does cause the software to stop functioning</td>
<td>Incorrect output when using a specific input</td>
<td>C2</td>
</tr>
<tr>
<td>Average</td>
<td>The defect does not result in the failure of the entire system, but it does cause the software to stop functioning</td>
<td>Incorrect output when using a specific input</td>
<td>C3</td>
</tr>
<tr>
<td>Minor</td>
<td>The defect does not cause a failure, does not impair usability, and the associated concerning features are rarely related to wanting around the defect</td>
<td>Incorrect output when using a specific input</td>
<td>C4</td>
</tr>
</tbody>
</table>

Table 2: Severity coding for testing the software.

![Screenshot example of the test results.](image)

**Table 3: Found failures and corrections during the different test phases of the software development process.**

<table>
<thead>
<tr>
<th>Test Phase</th>
<th>Number of test cases</th>
<th>Critical</th>
<th>Major</th>
<th>Average</th>
<th>Minor</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha)-2</td>
<td>100</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>(\alpha)-3</td>
<td>100</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>(\beta)</td>
<td>200</td>
<td>0</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Release</td>
<td>300</td>
<td>0</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

New features were added after the \(\alpha\)-2 and \(\alpha\)-3 phases.

Based on the time series of identified failures during the \(\alpha\)-2, \(\alpha\)-3 and \(\beta\) tests, see Table 3, we fitted software models as described in the previous sections. At the \(\alpha\)-1 phase, however, the software was not considered to be
sufficiently mature for testing. Also, all found failures as shown in Table 3 are newly discovered. Software reliability can be estimated once the mean value function is determined. Based on Akaike’s Information Criterion (AIC), the Mean Square Error (MSE) values and the significance of parameters, it appeared that the simple Goel-Okumoto model fits the failure rates best for three test series, see Table 4. Table 5 shows the estimated number remaining failures and their upper bounds of the 95% confidence intervals. After the β release we found another 3 failures in approximately 300 cases. This number is smaller than the 95% upper bound as determined at the end of test phase β. Figure 6 shows the failure intensity function $\hat{\lambda}(t)/\hat{t} = b(a - m(t))$ of each phase. Note the (large) increases of the failure intensities by adding new features (with additional failures).

In the discussed α- and β-phases we discovered 57 failures (= 36 critical/majors + 21 average/minors). This number does not deviate hugely from the number of failures as predicted by the rule of thumb (81). We expect that users will discover some additional failures in the (near) future.

Table 4: Fitting results for the several software reliability models during the consecutive testing phases.

<table>
<thead>
<tr>
<th>Model</th>
<th>MSE</th>
<th>AIC</th>
<th>α, β, remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goel-Okumoto</td>
<td>0.187</td>
<td>30.290</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Masa-Okumoto</td>
<td>0.108</td>
<td>30.049</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada S-shaped</td>
<td>1.157</td>
<td>31.651</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada Imperfect 1</td>
<td>0.059</td>
<td>31.128</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada Imperfect 2</td>
<td>1.784</td>
<td>31.911</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada Imperfect 2 No convergence MLE</td>
<td>0.546</td>
<td>20.492</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Jeneliki-Moranda</td>
<td>0.449</td>
<td>59.895</td>
<td>parameter estimates are not significant</td>
</tr>
</tbody>
</table>

The PHZ model could not be estimated due to lack of convergence.

<table>
<thead>
<tr>
<th>Model</th>
<th>MSE</th>
<th>AIC</th>
<th>α, β, remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goel-Okumoto</td>
<td>0.150</td>
<td>19.956</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Masa-Okumoto</td>
<td>0.133</td>
<td>20.031</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada S-shaped</td>
<td>0.591</td>
<td>21.297</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada Imperfect 1</td>
<td>0.546</td>
<td>20.492</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada Imperfect 2</td>
<td>1.511</td>
<td>34.089</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada Imperfect 2 No convergence MLE</td>
<td>0.083</td>
<td>97.904</td>
<td>parameter estimates are not significant</td>
</tr>
</tbody>
</table>

The PHZ model could not be estimated due to lack of convergence.

<table>
<thead>
<tr>
<th>Model</th>
<th>MSE</th>
<th>AIC</th>
<th>α, β, remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goel-Okumoto</td>
<td>0.058</td>
<td>32.181</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Masa-Okumoto</td>
<td>0.277</td>
<td>32.835</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada S-shaped</td>
<td>0.990</td>
<td>34.921</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada Imperfect 1</td>
<td>0.074</td>
<td>34.091</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada Imperfect 2</td>
<td>14.511</td>
<td>34.089</td>
<td>parameter estimates are not significant</td>
</tr>
<tr>
<td>Yamada Imperfect 2 No convergence MLE</td>
<td>0.083</td>
<td>97.904</td>
<td>parameter estimates are not significant</td>
</tr>
</tbody>
</table>

Table 5: Remaining failures and 95% upper bound

<table>
<thead>
<tr>
<th>End of test phase</th>
<th>Expected remaining Critical/Major failures</th>
<th>95% upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>α-2 phase</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>α-3 phase</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>β-phase</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 6: Failure intensity function

5. Interaction with Hardware: System Reliability

Using the V-model, one is able to consider the reliability of both the hardware and software components. By doing so, and we demonstrated in this paper that this is possible, one still needs to combine these two models in order to determine the reliability of the system. In section 2, we already described the huge differences between these two components. To make it even more complex, consider that on a system level, there is a distinct difference between the reliability and the availability of the system. As such, the definition of availability is:

The degree to which a system is operational and accessible, when required for use. Availability requirements for consumers are lower as for healthcare applications.

A system can have a low reliability but still a very high availability. But in general, a high reliability will always lead to a high availability. Both items belong to what is called the system dependability [20]. Dependability (see also Figure 6) is the ability of a system to avoid failures that are more frequent and more severe than acceptable. A dependable system is:

• having all its required properties,
• and does not show failures.

Much literature is available on the prediction of software reliability [1] and even more is available on the prediction of system reliability based on its hardware [2]. Little literature is available on the interaction of the above. In a consecutive study we will devote our investigation to this subject.
Figure 6: The dependability tree [19].

6. Conclusions
In this paper we will present our results in predicting the reliability of software and present a use case to demonstrate how software reliability models can be obtained. These software reliability models enable the developer to determine the optimal time to stop software testing and decide to release the software. Several other criteria, such as the number of remaining errors, failure rate, reliability requirements, or total system cost, may be used to determine optimal testing time.

Acknowledgments
This research work was funded by the European Union (EU) Artemis project Design, Monitoring, and Operation of Adaptive Networked Embedded Systems (DEMANES).

References