Software Reliability and Testing Time Allocation: An Architecture-Based Approach

Roberto Pietrantuono, *Member, IEEE,* Stefano Russo, *Member, IEEE,* and Kishor S. Trivedi, *Fellow, IEEE*

***Abstract*—With software systems increasingly being employed in critical contexts, assuring high reliability levels for large, complex systems can incur huge verification costs. Existing standards usually assign predefined risk levels to components in the design phase, to provide some guidelines for the verification. It is a rough-grained assignment that does not consider the costs and does not provide sufficient modelling basis to let engineers quantitatively optimize resources usage. Software reliability allocation models partially address such issues, but they usually make so many assumptions on the input parameters that their application is difficult in practice. In this paper we try to reduce this gap, proposing a reliability and testing resources allocation model that is able to provide solutions at various levels of detail, depending upon the information the engineer has about the system. The model aims to quantitatively identify the most critical components of software architecture in order to best assign the testing resources to them. A tool for the solution of the model is also developed. The model is applied to an empirical case study, a program developed for the European Space Agency, to verify model's prediction abilities and evaluate the impact of the parameter estimation errors on the prediction accuracy.**

***Index Terms*—Reliability, Software Architecture, Software Testing**

I. Introduction

COMPLEX software systems are increasingly employed in critical scenarios, such as air traffic control, railway transportation, and medical devices. The criticality of such scenarios poses new challenges to software engineers, who need to develop systems with assured high reliability levels while at the same time keeping the development time and costs low. Towards this aim, the development process of such systems is usually complemented by several analysis techniques (e.g., hazard analysis, FTA, FMECA) in the requirement specification, and in the design phase as well. Once the system has been implemented, the verification process has to provide the final assurance that the system meets the required reliability level. The verification phase is usually responsible for the major fraction of the overall costs, especially for critical systems. Efficacy of the verification phase strongly depends on the correct identification of the most critical components in the software architecture, as the available testing resources are usually allotted based on the components' risk levels. Identifying parts of a complex system that are the major contributors to its unreliability is not always an easy task, and consequently the testing resource allocation is most often based on engineering judgement and hence suboptimal. Sometimes, engineers are inclined to judge as "most critical" components that are the most complex ones, or those that are the most used ones and devote most of the testing efforts to them.

Some standards and methodologies[1] for critical systems [1], [2] suggest to assign predefined risk levels to components, based on the risk level of the services in which they are involved. This is clearly a rough-grained assignment that merely provides some guidelines for the verification phase. Such judgement-based approaches not only could lead to wrong assignment (e.g., a less reliable, but rarely used component could affect the total reliability less than a more reliable and frequently used one), but they do not even provide sufficient quantitative information in order to judiciously allocate testing resources and quantify the reliability of the final system. Allocation of testing resources based on quantitative reasoning is essential to answer questions such as:

- *How much* risk does a component pose to the system?
- Do components at the same risk level have the same impact on the system reliability?
- What is the impact of a *change* in the reliability of a component on the system reliability?

  Most importantly, non-quantitative approaches cannot answer the following question:
- What is the reliability that each component needs to achieve in order to assure a minimum system reliability level, and at what cost?

We believe that a system analysis aiming at assuring the required reliability while minimizing the testing costs needs to be quantitative. Several researchers have tried to quantify the required software components reliability that will assure a minimum total system reliability. This optimization problem has usually been addressed as *reliability allocation* problem. Most of the papers in the software field coped with the design phase and dealt with the redundancy reliability allocation [3], [4], [5]; some authors also dealt with the problem in the verification phase, where the issue is to allocate reliabilities to be achieved during testing [6], [7]. Typically, these problems are addressed by proposing some kind of model that allows engineers to carry out an optimal allocation. However, none of the models proposed so far meet the following requirements:

1) a model for reliability analysis of a software architecture should explicitly describe the relationships among its

R. Pietrantuono and S. Russo are with the Department of Computer and Systems Engineering, Federico II University of Naples, Via Claudio 21, 80125 Naples, Italy. E-mail: {roberto.pietrantuono, Stefano.Russo}@unina.it

K.S. Trivedi is with the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708, USA. E-mail: kst@ee.duke.edu.
Trivedi's research was funded in part by the US National Science Foundation Grant NSF-CNS-08-31325

[1]These standards are about the safety assessment. Safety is the probability that a system does not "catastrophically" fail in [0,t], while reliability is the probability that the system does not fail in [0.t]. This difference does not affect the model formulation and applicability; we will use reliability, for historical reasons (i.e., the model is close to the *reliability allocation models*).

components, in order to consider the effects of individual component reliabilities on the system reliability;
2) the reliability of a complex system does not only depend on the application components, but also on the operational environment; thus, a model should also take into account the reliability of the underlying software layers, such as the Operating System. Indeed, due to the intensive and continuous usage of OS services by the application components, the OS reliability level has a significant impact on the overall system reliability so that it should not be neglected;
3) since in a critical application fault tolerance mechanisms are increasingly adopted, a model should consider the presence of such means of failures mitigation;
4) for a model to be useful in practice, it needs to be flexible enough to give detailed answers when the user has a lot of information, and to continue giving useful indications, even though less accurately, when not much information is available.

In this paper we propose an approach to *quantitatively identify the most critical components of a software architecture in order to best assign the testing resources to them.* In particular, we present an optimization model for testing resources allocation that includes all of the above-mentioned aspects affecting the reliability of a complex software system. In order to represent the software architecture, we employ the so-called architecture-based reliability model; in particular a Discrete Time Markov Chain (DTMC) type state-based model is adopted. This allows us to explicitly consider the effects of such architectural features as loops and conditional branching on the overall reliability. Moreover, the architectural model encompasses the operating system to consider its reliability and its influence on the application.

The proposed optimization model also considers the most common fault tolerance mechanisms (such as restart a component, retry application as recovery mechanisms as also a failover to a standby) that critical systems typically employ. Furthermore, we try to impart the necessary flexibility to the model by: (i) providing different levels of solutions according to the information the user gives as input, and (ii) carrying out a sensitivity analysis in order to analyze the effect of the variation of some parameters on the solution. Information needed for model parameterization can be obtained by the user either considering design/code information (such as UML diagrams) and simulation before the testing of the system version under consideration or by dynamically profiling a real execution from system test cases of a previous version. Depending on the availability and the accuracy of information, the user may adopt one of the two approaches (or a combination of both).

Finally, the impact of performance testing time and the second-order architectural effects are also considered for a greater accuracy of the result.

The optimization model is implemented in a prototype tool, which receives the model parameters (e.g., the DTMC transition probabilities) and the user options as inputs and provides the solution by using an exact optimization technique (the *sequential quadratic programming* algorithm).

The approach is then applied to an empirical case study to verify the accuracy of the prediction abilities. The program chosen consists of almost 10,000 lines of C code, and was developed for the European Space Agency (ESA). Once we built the DTMC architectural model, we predicted the amount of testing needed for each component, in order to achieve a predefined level of reliability. Then the software was tested according to the predicted times and the actual achieved reliability was compared with the predicted reliability. We also evaluated the effect of the error in the parameter estimations on the prediction accuracy.

The rest of this paper is organized as follows: Section II provides an overview of related work, gives a basic background and introduces some terminology. Section III describes the adopted architectural model and Section IV illustrates the optimization model. Section V discusses the possible sources of information about the parameters of the architecture and its components, and how it can be retrieved. Finally, Section VI discusses the experiments, followed by the conclusions in Section VII.

II. Related Work and Background

Lot of work in the past considered the optimal allocation of the reliabilities to minimize a cost function, related to the design or the verification phase costs. Much initial research dealt with hardware systems (e.g., the series-parallel redundancy-allocation problem has been widely studied [8], [9], [10]); software systems received attention more recently. Most of work in the software area is concerned with the design phase, in which the goal is to select the right set of components with a known reliability and the amount of redundancy for each one of them, minimizing the total cost under a reliability constraint [11], or maximizing the total reliability under a cost constraint [3], [4], [5] (more specifically, this is a redundancy reliability allocation problem).

In some cases they also considered the redundancy strategies and the hardware. For instance, the work in [3] defines a model comparing different redundancy strategies at different level, giving as output the best redundancy strategy to achieve the required reliability.

When redundancy is not considered, the reliability allocation problem can still refer either to the design or to the verification phase. For instance, authors in [12] proposed an economic model to allocate reliabilities during the design phase, minimizing a cost function depending on fixed development costs and on a previously experienced failure decrease cost. The work in [13] also refers to the design phase and authors define a general-behavior cost function to relate the costs to the reliability of a component.

Not many papers considered the problem in the software verification phase, where the issue is to allocate reliabilities that components need to achieve during their testing. Among these papers, authors in [6], [14], proposed an optimization model with the cost function based on well-known reliability growth models. They also include the use of a coverage factor for each component, to take into account the possibility that a failure in a component could be tolerated (but fault tolerance

mechanisms are not explicitly taken into account, and the coverage factor is assumed to be known). The authors in [7] also try to allocate optimal testing times to the components in a software system (here the reliability-growth model is limited to the Hyper-Geometric (S-shaped) Model).
Some of the cited papers [3], [6], [14] also consider the solution for multiple applications; i.e., they aim to satisfy reliability requirements for a set of applications. However, none of the cited papers explicitly considers the architecture of the application. Work in [12] and [15], as well as [16] and [3], consider the software architecture implicitly, by taking into account the utilization of each component with a factor assumed to be known. Among these, only Everett [16] refers to the verification phase. Almost all of the cited papers about reliability allocation belong to the class of the so-called additive models [17]. However, there are other ways to describe a software application, which can explicitly consider the architecture and lend themselves to an easy integration with the other aspects described in the introduction, such as the Operating System, the fault tolerance mechanisms, the sensitivity analysis and the performance testing. They are the state-based models and the path-based models. Both the latter ones and additive models belong to the class of the so-called *Architecture-based* models.
This kind of models have gained importance since the advent of object-oriented and component-based systems, when the need to consider the internal structure of the software to properly characterize its reliability has become important (in the past, reliability analysis was conducted mainly considering the software as a black box). This led to an increasing interest in the architecture-based reliability and performance analysis [18], [19], [20], [21].
The main features of the three mentioned classes of architecture-based models are the following [17]:

- State-based models use the control flow graph to represent software architecture; they assume that the transfer of control among components has a Markov property, modelling the architecture as a Discrete Time Markov Chain (DTMC) a Continuous Time Markov Chain (CTMC) or semi Markov Process (SMP).
- Path-based models compute the system reliability considering the possible execution paths of the program.
- Additive-models, mentioned above, where the component reliabilities are modelled by non-homogeneous Poisson process (NHPP) and the system failure intensity is computed as the sum of the individual components failure intensities.

So far, state-based and path-based models have been mainly used to analyze system reliability starting from its component reliabilities, while the reliability allocation problem has been mainly based on additive models, as described above. In this paper we try to leverage state-based models ability of more accurate capture of the architecture (they can describe the architecture explicitly), by combining it with the NHPP models ability in relating the reliability and the testing time, as shown in Section IV. State-based models can be further categorized into composite and hierarchical models [22]. In the former, the software architecture and the failure behavior of the software are combined in the same model, while hierarchical approach separately solves the architectural model and then superimposes the failure behavior of the components on the solution. Although hierarchical models provide and approximation to the composite model solution, they are more flexible and computationally tractable. In the composite model, evaluating different architectural alternatives or the effect of changing an individual components behavior is computationally expensive. Unlike hierarchical models, they are also subject to the problem of stiffness [23]. To cope with the accuracy gap between hierarchical and composite models, Gokhale and Trivedi [24] included the second-order architectural effects in hierarchical models. In this paper, hierarchical modeling approach with the second-order architectural effects is used.

A. Terminology

In this section, we briefly introduce some terms used in the rest of the paper. Architecture-based models are conceived to relate the behavior of the system (expressed by some attribute, e.g., the reliability) to the behavior of its parts. In the literature, the *parts* of the application under study are often referred to as “components”. Although the notion of component is not well defined and universally accepted (except for applications based on component models such as CCM, EJB or DCOM)[2], in the context of architecture-based analysis it is intended as a logically independent unit performing a well-defined function [19]. The level of decomposition, which defines the granularity of components, is an analysis choice, addressed by a trade off between a large number of small units and a small number of large units.
In order to optimally allocate the testing resources to components while achieving a reliability goal, a relation between reliability and testing is needed. For this purpose, we use Software Reliability Growth Models (SRGM), i.e., models that describe how reliability grows as software is being improved (by faults detection and removal). There is an extensive body of literature on SRGMs and many different models are available.
Such models are usually calibrated using failure data collected during testing; they are then employed for predictions, in order to answer questions such as “how long to test a software”, or “how many faults likely remained in the software”, and so on. In the context of SRGMs, the term *failure intensity* refers to the number of failures encountered per unit time; its form determines a wide variety of SRGMs.
The time dimension over which reliability is assessed to grow, can be expressed as calendar time, clock time, CPU execution time, number of test-runs, or some similar measures. However, in general, the testing effort and its effectiveness do not vary linearly with time. The functions that describe how an effort is distributed over the exposure period, and how effective it is, are referred to as testing-effort functions (TEF) [27]. To address this issue, some SRGMs [27], [29], [30] also include

[2]Respectively, “The Corba Component Model” (http://www.omg.org), “Enterprise Java Beans” (http://java.sun.con/products/ejb) and “Component Object Model” (http://www.miscrosoft.com/com).

a TEF (Testing Effort Function) to describe this relation.
The model that we propose in the next sections uses SRGMs, one for each component, to describe the relation between the reliability of a component and the testing effort devoted to it. This relation can be represented as TE = f(λ), where TE stands for Testing Effort and λ is the desired failure intensity. Without loss of generality, we can consider "testing time" in place of "testing effort" (T = f(λ)): in fact, if the user of the model wants to consider the testing effort variation (for one or more components), s/he simply chooses an SRGM for that component that includes a Testing Effort Function (TEF) (like the cited ones) and put it in the model like any other SRGM (as described in the section IV). Hence, in the following, we refer to testing time and testing effort synonymously.

III. Architectural Model

We describe the software architecture by an absorbing DTMC, to represent terminating applications (as opposed to irreducible DTMCs, which are more suitable to represent continuously running applications.) A DTMC is characterized by its states and transition probabilities among the states. The one-step transition probability matrix P = [$p_{i,j}$] is a stochastic matrix so that all the elements in a row of P add up to 1 and each of the $p_{i,j}$ values lies in the range [0, 1]. The one-step transition probability matrix with *n* states and *m* absorbing states can be partitioned as:

$$P = \begin{pmatrix} Q & C \\ 0 & I \end{pmatrix} \tag{1}$$

where Q is an *(n-m)* by *(n-m)* sub-stochastic matrix (with at least one row sum <1), I is an *m* by *m* identity matrix, 0 is an *m* by *(n -m)* matrix of zeros and C an *(n-m)* by *m* matrix. If we denote with P^k the k-step transition probability matrix (where the entry *(i,j)* of the submatrix Q^k is the probability of arriving in the state *j* from the state *i* after *k* steps), it can be shown [25], [27] that the so-called fundamental matrix M is obtained as

$$M = (I-Q)^{-1} = I + Q + Q^2 + \cdots + Q^k = \sum_{k=0}^{\infty} Q^k \tag{2}$$

Denoting with $X_{i,j}$, the number of visits from the state *i* to the state *j* before absorption, it can be shown that the expected number of visits from *i* to *j*, i.e., $v_{i,j}$ = E[$X_{i,j}$], is the $m_{i,j}$ entry of the fundamental matrix. Thus, the expected number of visits starting from the initial state to the state *j* is:

$$v_{1,j} = m_{1,j} \tag{3}$$

These values are called expected visit counts; we denote them with V_j = $v_{1,j}$. They are particularly useful to describe the usage of each component in the application control flow. An alternative form to compute V_j values is described in [25].
We can also compute the variance of visit counts, using M [28]. Denote with $\sigma^2_{i,j}$ the variance of the number of visits to *j* starting from *i*. If we indicate with M_D the diagonal matrix with:

$$M_D = \begin{cases} m_{i,j} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

and define M_2 = [$m^2_{i,j}$], we have

$$\sigma^2 = M(2M_D - I) - M_2 \tag{5}$$

Hence

$$Var[X_{i,j}] = \sigma^2_{i,j} \tag{6}$$

To represent the application as a DTMC, we consider its control flow graph. Assuming that an application has *n* components, with the initial component indexed by 1 and the final component by *n*, DTMC states represent the components and the transition from state *i* to state *j* represents the transfer of control from component *i* to component *j*. Following the procedure explained above, we can compute the expected number of visits to each component and its variance.
The DTMC representation, along with the concept of visit counts, has been used to express the system reliability as a function of component reliabilities.
In particular, denoting with R_i the reliability of component *i*, the system reliability is the product of individual reliabilities raised to the power of the number of visits to each component, denoted by $X_{1,i}$ (i.e., each component reliability is multiplied by itself as many times as the number of times it is visited starting from the first component); i.e., $R \approx \prod_i^n R_i^{X_{1,i}}$. Since the number of visits to a component is a random variable (except for the last component), the so-computed system reliability is also a random variable. Thus, denoting with E[R] the total expected reliability of the system, we have:

$$E[R] \approx \prod_i^n E[R_i^{X_{1,i}}] \approx (\prod_i^{n-1} R_i^{E[X_{1,i}]}) R_n \tag{7}$$

where E[$X_{1,i}$] is the expected number of visits to component *i* (and $X_{1,i}$ is always 1 for the final component *n*). The adopted model is known to belong to the class of hierarchical approaches; this kind of models, though approximate, lead to quicker and more tractable solutions than composite models [18]. To also take into account the second-order architectural effects and obtain a more accurate result, we can expand the above equation according to the Taylor series, as shown in [28]:

$$E[R] = [\prod_i^{n-1} (R_i^{m_{1,i}} + \frac{1}{2}(R_i^{m_{1,i}})(\log R_i)^2 \sigma^2_{1,i})] R_n \tag{8}$$

Where $m_{1,i}$ = E[$X_{1,i}$] and $\sigma^2_{1,i}$ = Var[$X_{1,i}$] (since $X_{1,n}$ is always 1, $m_{1,n}$ = 1 and $\sigma^2_{1,n}$ = 0).

The second-order architectural effects are captured by the variance of the number of visits. The only source of approximation is the Taylor series cut-off. Note that the described model, as most of the architecture-based models, assumes independent failures among components. A more complex analysis would need to also consider failure dependencies. To complete the architectural model description, we add a state representing the Operating System (OS). The OS is considered as a component accessed through the system call interface.

Transitions to the Operating System represent the transfer of control from components to the OS, i.e., a service request carried out via a system call.

Assume for now that the OS reliability per request is known and denote it with K (further details about how to estimate this value are in Section V); we have that

$$K' = E[K^{X_{1,OS}}] = (K^{m_{1,OS}} + \frac{1}{2}(K^{m_{1,OS}})(\log K)^2\sigma^2_{1,OS}) \tag{9}$$

where $m_{1,OS}$ and $\sigma^2_{1,OS}$ respectively denote the mean and the variance in the number of visits to the OS. In the same manner, to simplify the notation, define:

$$R'_i = E[R_i^{X_{1,i}}] = (R_i^{m_{1,i}} + \frac{1}{2}(R_i^{m_{1,i}})(\log R_i)^2\sigma^2_{1,i}) \tag{10}$$

and the expression 8 becomes:

$$E[R] = [\prod_i^{n-1} R'_i]R_n * K' \tag{11}$$

In the optimization problem, E[R] is required to be greater than a predefined level, R_{MIN} (R_{MIN} is an input) and R_i values are the decision variables (i.e., they are the output), while K is a given constant.

IV. Optimization Model

In this Section we present the optimization model starting from a basic form and then enrich it by adding several features. The goal of the optimization model is to find the best combination of testing efforts to be devoted to each component so that they achieve a reliability level that can assure an overall reliability E[R] $\geq R_{MIN}$. Based on this model output, the tester will perform the verification activities focusing greater efforts on more critical components. If we assume that the reliability of each component grows with the testing time devoted to it, we can describe this relation by a software reliability growth model (SRGM), as in the cited additive models. This relation can be represented as $T = f(\lambda)$, where T is the Testing Time and λ is the failure intensity.
The general optimization model will then look like:
determine the optimal values of $T_1, T_2, \ldots, T_n$ so as to

$$\text{Minimize} \qquad T = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} f_i(\lambda_i) \tag{12.a}$$

subject to:

$$E[R] = [\prod_i^{n-1} R'_i]R_n * K' \geq R_{MIN} \tag{12.b}$$

with i = 1 ...*n-1*, *n* components and T indicating the total testing time for the application to get a total reliability E[R] $\geq R_{MIN}$. Here λ_i variables are the decision variables (which determine the T_i variables and that are of course present in the constraint factors, via $R_i = \exp[-\int_0^{t_i}\lambda_i(\theta)d\theta]$).
Note that after the application has been tested according to the output of the model and then released, the component failure intensities are assumed to be constant; this is reasonable if the application developer does not debug or change the component during the operational phase.[3] With this assumption, the reliability of the component *i* at the end of the testing will be:

$$R_i = \exp[-\int_0^{t_i}\lambda_i(\theta)d\theta] = \exp[-\lambda_i t_i] \tag{13}$$

with t_i is the expected execution time per visit to component i; this equation relates the failure intensity of component i to its reliability. Each component can be characterized by a different SRGM (among the plethora of proposed ones). For instance, if we assume the Goel-Okumoto model for all the components (for which $\lambda(T_i)$ = $a_i g_i e^{-g_i T_i}$), the objective function becomes T = $\sum_i (1/g_i ln(a_i g_i/\lambda_i))$, where a_i is the expected number of initial faults, λ_i is the desired failure intensity and g_i is the decay parameter. Other SRGMs can be used to represent the *reliability-testing time* relation for each component. This general model can be specialized according to the information the tester has, in order to get different accuracy levels in the solution in a flexible way. Assume that the tester has no knowledge about the components and their previous history. In other words, s/he does not have any historical data related to previous testing campaigns performed on the components (or on similar ones), and s/he is not able to obtain such information from the current version. Without such knowledge, s/he cannot build the SRGMs for these. In this case a basic, minimal solution can be obtained. In particular, the output will be the reliability that each component needs to achieve, assuming that the testing time to achieve a failure intensity λ is the same for all components, as though they were described by the same SRGM with identical parameters (i.e., $f_i(\lambda_i)$ = $f(\lambda_i)$). In this case, the model will then look like:

$$\text{Minimize} \qquad T = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} f_i(\lambda_i) = \sum_{i=1}^{n} f(\lambda_i) \tag{14.a}$$

subject to:

$$E[R] = [\prod_i^{n-1} R'_i]R_n * K' \geq R_{MIN} \tag{14.b}$$

In other words, in this case the model does not predict the testing times needed to achieve the required reliability. Therefore, the results have to be interpreted as an indication of the most critical components in the architecture or, equivalently, as the reliability values each component needs to achieve to satisfy E[R] = R_{MIN}. Measuring the reliability during the testing, (e.g. as in [31]), the engineers will know when the testing for each component can be stopped. We call this solution the ***basic solution***.
If some qualitative indications about the testing cost of components are available, it is possible to include them as weights in the objective function. For instance, information about process/product metrics, that is easily obtainable, can be used to estimate their fault content. Regression trees [32] are very

[3]This assumption, anyway, is not that important for us, because the problem of estimating the reliability variation during operation or maintenance is a different problem; for our purpose, i.e, satisfying the reliability constraint at the software release (i.e., at the end of testing phase) in minimal tesitng time, the stated assumption is not that relevant.

useful for this purpose, but also the simpler fault density approach [33] can be used. By including weights proportional to the estimated fault content, the model solution not only gives the reliabilities the components need to achieve, but also an indication about the *relative* testing efforts to make them achieve such reliabilities, according to the components complexity. The previous assumption is relaxed and becomes: each component needs an amount of testing time that is proportional to its fault content, and hence, indirectly to its complexity. A possible way to include the weights in the objective function is the following:

$$T_i = f(\frac{\lambda_i}{1 + rWeight_i}) \tag{15}$$

leading to the following model:

$$\text{Minimize} \qquad T = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} f(\frac{\lambda_i}{1 + rWeight_i}) \qquad (16.a)$$

subject to:

$$E[R] = [\prod_{i}^{n-1} R'_i] R_n * K' \geq R_{MIN} \qquad (16.b)$$

where $f(\lambda_i)$ is a default SRGM, the same for every component with the same parameter as in the basic solution, and $rWeight_i$ (i.e., reliability weights) is given by the proportion of the fault content in the component i with respect to the entire system's estimated fault content (i.e. $faultContent_i$ / $\sum_i faultContent_i$). We call this solution ***extended solution***.
Note that in order for the weights to really increase the needed testing time when they increase, the f(λ) function (the inverse of the default SRGM) is preferable to be a decreasing function, rather than an increasing/decreasing function (as for instance, the one derived by the Goel-Okumoto SRGM), and a simple function to be evaluated. The adoption of a non-decreasing function would require some expedients to be adopted for it to work correctly; however, since the only important requirement is that this function be the same for all the components, it makes no sense choosing a complex function. The most appropriate choice is a simple SRGM, like the Goel-Okumoto model.

The general model described by the equation 12 can be fully exploited when engineers also collect information about the failure behavior of the components. In this case the optimal testing times needed for each component can be predicted. In fact, data about the components failure behavior allows engineers to build a reliability growth model for each one of the component.
Engineers often collect this kind of information during the testing process for various purposes, like improving the process, assessing the achieved reliability after some testing time, building reliability growth models, and use it in the same or in successive projects for scheduling optimal release policies. In this case, failure data, (i.e., interfailure times or the fault density along with the coverage function [18]) are used to fit the best SRGM (the issue of how to fit an SRGM for the current project is described in Section V). By solving the model with the SRGMs, the *absolute* testing times to be devoted to each component are obtained. The accuracy of the result depends on how well the testing processes of the components are described by their corresponding SRGMs. The previous assumptions are replaced by the common SRGM assumptions (as summarized in [34] and many other related papers). We call this general solution, epressed by equation 12, ***complete solution***.
Finally, note that the model expression 12 is generalized to the case of multiple applications by adding a reliability expression, of the form of equation 11, for each application as a constraint. The solution for multiple-applications problems usually requires heuristic approaches (a genetic algorithm in our case).

A. Performance Testing Time Contribution

During the testing process of a critical application, part of testing resources could be reserved to tune the application performance in order to fulfill performance requirements. This is especially true for real-time systems. To also consider the additional testing resources each component can require for performance testing we should know the relation between the performance improvement and the performance testing time (i.e., a sort of *performance growth model*). Even if this could be a topic of future research, more realistically it is difficult to infer such relations, differently from reliability growth models. For this reason, it is easier to include the performance testing time contributions as weights in the objective function. In the extended solution case, where no SRGM was available, we saw that the *rWeight* values represent the assumed proportionality between the testing time and the estimated fault content (and indirectly to process/product metrics). In a similar way, the performance weights will represent the proportionality between the performance testing time and some performance metric, e.g., the expected total execution time for a component (ETET). Denoting with t_i the expected execution time per visit for the component i, the $ETET_i$ is given by $ETET_i$ = $V_i t_i$. The $argmax_i\{V_i t_i\}$ is the performance bottleneck of the application [18]. Thus, assuming that performance testing time will be proportionally devoted to components according to their $ETET_i$ value, we can add a weight, named *pWeight*, in the objective function computed as ($ETET_i$ /$\sum_i ETET_i$). This weight represents the *performance testing time contribution*. In this way, performance bottlenecks will receive more performance testing time. A possible objective function for the extended solution (and the basic one, if $rWeight_i$ = 0) can be the following:

$$\text{Minimize} \quad T = \sum_{i=1}^{n} (f(\frac{\lambda_i}{1 + rWeight_i}) * [1 + PF * pWeight_i]) \tag{17}$$

where the first term in the sum has the same meaning as in equation 16 and PF is a factor between 0 and 1 representing the percentage of time the tester wants to devote to performance tests. When more information is available (like maximum tolerable execution times for each component, as in the case of real time systems) different weights are possible (e.g., ranking the component based on the difference between the total expected execution time and the imposed maximum execution time; the greater is the difference, the more performance testing time they need). For the complete solution, when the objective function is the sum of the testing time functions $f_i(\lambda_i)$ (i.e. the inverse of SRGMs), the performance weights, *pWeight*, can be added in the same way, as a penalty, due to the "performance testing" time to be devoted to each component. We thus have:

$$\text{Minimize} \quad T = \sum_{i=1}^{n} (f_i(\lambda_i) * [1 + PF * \textit{pWeight}_i]) \qquad (18)$$

In both cases, accounting for performance testing time will be optional (if the tester does not want to account for it, PF will be 0). In the future, we plan to explore the relation among performance testing times and performance improvement, in order to formulate a testing resource allocation problem, with both minimum reliability and minimum performance levels as constraints.

B. Fault Tolerance Mechanisms

The model considers the potential means of failure mitigation that one or more components could employ. Main mechanisms that are adopted in critical systems, namely, the restart-component, retry-application and failover to a standby are considered here. Denote as "subsystem C" a component along with its standby version (in the case of no standby version, C denotes a single component). The fault tolerance mechanisms are considered in the following order (Figure 1): if a failure occurs, and the failure is detected, the first recovery attempt is to restart-component operation; the second recovery attempt, if the first one fails, is to retry-application and the final operation is a failover to a standby version, once the other actions have failed. The expression we derive below for the reliability of such a subsystem can describe components implementing one or more of these fault tolerance mechanisms. Other mechanisms can easily be added to the model, by deriving the corresponding reliability expression.

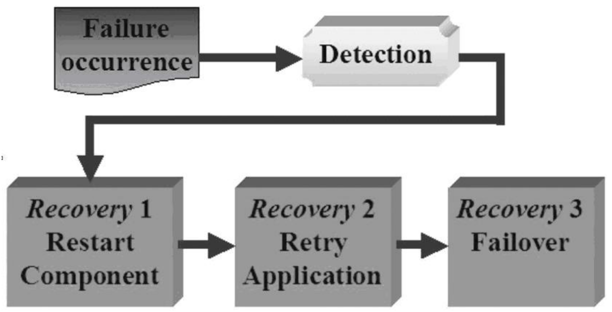

Fig. 1. Fault Tolerance Mechanisms

In this case, we need to consider the following events in order to derive an expression for the reliability of the subsystem C:

- DETFailed = detection failed given a failure occurred
- RESFailed = the restart failed given a failure and detection occurred
- RETFailed = the retry failed, given a failure occurred, detection occurred and the restart failed
- FOFailed = the failover failed given a failure occurred, detection occurred, the restart failed and the retry also failed.

Describing the component reliabilities as in the equation 13, the failure probability of a component will be F_i = 1 - R_i = 1 - exp[-λt] , i.e. the time-to-failure (TTF) has an exponential distribution. When a failure occurs in the primary component of a subsystem C, the following group of events (we denote them as E1, E2, E3) can take place:

1) The detection fails, or the detection succeeds but the restart, retry and failover operations subsequently fail (i.e. P_{E_1} = $P_{DET_{Failed}}$ + (1 - $P_{DET_{Failed}}$)($P_{RES_{Failed}} P_{RET_{Failed}} P_{FO_{Failed}}$): in this case the failure has not been covered and the conditional TTF distribution of the subsystem C is given by the TTF of the primary version:

$$EXP(\lambda) = 1 - e^{-\lambda t} \qquad (19)$$

2) The detection succeeds and the restart also succeeds, or the retry succeeds after the restart failed (i.e. P_{E_2} = (1 - $P_{DET_{Failed}}$)[(1 - $P_{RES_{Failed}}$)+$P_{RES_{Failed}}$(1 - $P_{RET_{Failed}}$)]). In this case, the failure has been covered and the same component starts operating from scratch. Thus, the conditional TTF distribution is given by the sequence of two identical exponentially distribution representing the same component rerun, i.e., by a 2-stage *Erlang*:

$$ERLANG(\lambda, 2) = 1 - (e^{-\lambda t}(1 + \lambda t))) \qquad (20)$$

3) The detection succeeds, the restart/retry operations fail and the failover to a standby version succeeds (i.e.P_{E_3} = (1 - $P_{DET_{Failed}}$)[$P_{RES_{Failed}} P_{RET_{Failed}}$(1 - $P_{FO_{Failed}}$)]). Thus, after a failure occurrence, the standby version will be activated. When this event occurs, if the standby is an identical copy of the primary version, the conditional TTF distribution is given again by the equation 20. If a different version is used, the distribution of the conditional TTF is given by the sequence of two independent exponential distributions (describing the primary and the standby TTF), which is known to be a Hypoexponential distribution. Given λ and λ_1, the failure intensities respectively of the primary version and its standby, the TTF distribution will be

$$HYPO(\lambda, \lambda_1) = 1 - (\frac{\lambda_1}{\lambda_1 - \lambda})e^{-\lambda t} + (\frac{\lambda}{\lambda_1 - \lambda})e^{-\lambda_1 t} \qquad (21)$$

The most general reliability expression, for a component that owns all of the considered mitigation means, will be:

$$\begin{gathered} R_C = 1 - [P_{E_1} * EXP(\lambda) + P_{E_2} * ERLANG(\lambda, 2) + \\ + P_{E_3} * (\textit{sameVersion} * ERLANG(\lambda, 2) + \\ + \textit{!sameVersion} * HYPO(\lambda, \lambda_1))] \end{gathered} \qquad (22)$$

where *sameVersion* is 1 if the standby is identical to the primary copy, 0 otherwise. If a component does not have any mitigation means, the probabilities $P_{RES_{Failed}}$, $P_{RET_{Failed}}$ $P_{FO_{Failed}}$ will be 1 and P_{E_1} will also be 1, while P_{E_2} and P_{E_3} will be 0:

$$R_C = 1 - EXP(\lambda) \tag{23}$$

If a component has only the restart/retry mechanism and not a standby version, the $P_{FO_{Failed}} = 1$, $P_{E_3} = 0$ and the expression becomes:

$$R_C = 1 - [EXP(\lambda)P_{E_1} + ERLANG(\lambda, 2)P_{E_2}] \tag{24}$$

Finally, if a component has a stand-by version, but not the restart/retry mechanism, $P_{RES_{Failed}}$ and $P_{RET_{Failed}}$ are 1, $P_{E_2} = 0$ and the expression is:

$$\begin{gathered} R_C = 1 - [P_{E_1} * EXP(\lambda) + \\ + P_{E_3} * (\mathit{sameVersion} * ERLANG(\lambda, 2) + \\ + \mathit{sameVersion} * HYPO(\lambda, \lambda_1))] \end{gathered} \tag{25}$$

By replacing R_i by the R_{C_i} expression in the equation 10, the model will be described by the following:

$$\text{Minimize} \qquad T = \sum_{i=1}^{n} (f_i(\lambda_i)) \tag{26}$$

subject to:

$$E[R] = [\prod_{i}^{n-1} R'_{C_i}] R_{C_n} * K' \geq R_{MIN}$$

This general form allows adding any other failure mitigation means for the component *i*, by finding the corresponding expression for R_{C_i}. Similarly, if one wants to adopt more complex expressions to describe the reliability of a subsystem C, it is sufficent to replace the discussed R_{C_i} expressions with the new expression (for instance, in the discussed expression for the failover to a standby version, the statistical independence is assumed; the R_{C_i} expression can be replaced by more complex expressions accounting for any form of dependence). The parameters estimation ($P_{DET_{Failed}}$, $P_{RES_{Failed}}$, $P_{RET_{Failed}}$, $P_{FO_{Failed}}$) is briefly discussed in the next Section.

V. Information Extraction

The described approach is designed to provide different levels of solution according to the available information. To obtain a minimal solution, the basic information to be provided is related to the architecture (components identification, transition probabilities), and each component (expected execution time per visit (*t*) or expected total execution time (ETET) and the OS reliability). Additional information about the component fault density and some process/products metrics allows for computing the extended solution. Further information about the interfailure times, or the *coverage testing function / faults contents* for the components allows to obtain the SRGMs, and thus the complete solution. The ETET (or equivalently *t*) for the components also allows the performance testing time to be included in the solution. Finally, to include one of the described fault tolerance mechanisms, the corresponding parameters need to be estimated.

Since the model results have to be applied in the testing phase, there are basically two different ways to obtain such information: by design/code information and simulation before the testing or by dynamically profiling a real execution from system test cases of a previous version. The former refers to design documents (such as UML diagrams), and to static code analysis tools (mainly static profiling techniques and simulation tools). The latter refers to the execution of the system test cases, which emulate the system functionalities (thus, in this case a previous version of the software has to be profiled, since for the current version the testing still has not started). Dynamic profiling solution assumes that the executions will represent the real operational usage; this is generally accomplished in two ways: by assessing the operational profile and assigning an execution probability to each functionality or by shuffling and re-executing the system functional test cases, averaging the results.

Neither the design/code nor the dynamic profiling approach is the best one for all the parameters. An approach can be better for some parameters and worse for others.

The advantage of design/code-based estimations is that it relies on the current system version and not on a previous one. On the other hand, for some information (such as the ETET or transition probabilities) the dynamic profiling approach can be easier to use. As for the accuracy, if the system does not change much between the profiled version and the current version, execution traces would be more accurate than design-based approaches; but if the current version introduced significant changes in the code, a design-based approach would be better. If possible, a combination of both is probably the best solution: values obtained from past execution traces can be refined by reflecting the changes in the new version (which could have altered some values).

A third important way can be useful for both the approaches: expert judgments and historical data from similar systems. Basic information could be obtained as follows:

- **Architecture**: components (at each granularity level) are normally identifiable from design documents. When documents are not available, the architecture can be extracted by using some source code[4] as well as object code extraction tools [35]. It can also be derived by traces resulting from a dynamic profiling tool, such as gprof[5]. Transition probabilities can also be derived by both approaches. As for the design phase, the estimation can be accomplished by a scenario-based approach [36], by simulation, by static profiling of the execution or by interviewing the program users, as in [31]. As for the dynamic profiling, transition probabilities can be estimated by counting the number of times the control passes from a component to others (or to itself): point estimate of the transition probability from component *i* to component *j* will be given by (*NumberOfTransfers*$_{i,j}$ / $\sum_j NumberOfTransfer_{i,j}$).

  In particular, from the output of a profiler tool, the flat

[4]for instance, SWAGkit tool. Available from: http://www.swag.uwaterloo.ca/swag-1354kit/index.html.

[5]GNU gprof. Available from: www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html.

profile and the call graph can be obtained: information provided by the flat profile (i.e., how much time a program spent in each function and how many times that function was called) and by the call graph (i.e. information regarding the other functions calling a particular function and the functions called by it) allows us to construct the DTMC model and also get the relevant transition probabilities. A possible way to get these parameters is outlined in [19]. Finally, since the OS is usually not subject to changes, its reliability can be estimated by dynamic profiling the previous version or also other similar systems using the same OS. The point estimate is $K = 1 - lim_n F/N$, where F is the number of observed OS failures and N is the number of test cases.

- **Components**: The ETET of a component (or equivalently *t*) in the design phase can be obtained as in the case of the transition probabilities. Simulation could be more suitable in this case. However, this parameter is more easily obtainable experimentally, by the profiling approach, since the time spent in a component is an output of many profiler tools. Note that the accuracy of the expected time per visit, *t*, also depends on the granularity level chosen for a visit: for instance, in [18] a visit is intended as the execution of a basic block[6] of instruction (that in average was 2 lines of code) and, as a result, the expected time per visit did not differ significantly among visits. This consideration also stands for the transition probabilities accuracy. For the extended solution, we need an estimation of the component fault contents, to compute *rWeight* values. Following the fault density approach [33], we can estimate the fault density (FD) from past experience of the same system (previous versions) or similar systems, or by assuming common values from the literature or also by expert's judgment. The fault content of component *i* will be $\phi_i = FD * LOC_i$, where LOC is the number of code lines. Following the regression trees approach [32], we still need the estimated fault content of the system and some complexity metrics of the components to derive a more accurate estimation than the FD approach of the components' fault contents. Finally, for the complete solution, the data necessary to build the SRGMs can be the interfailure times or alternatively the testing coverage function along with the estimated component fault content [37]. Both of these are obtainable by the dynamic profiling approach (of the previous as well as the current version) or by historical data from the same or similar components.
In particular, when a component does not undergo significant changes from version to version, the SRGM parameters estimation based on the collected failure data can be used; when a component is significantly changed in the new version, some parameters can still be built using the collected data and refined through the current version information, while others are completely to be estimated from the current version information (e.g., the initial fault content, a common parameter, can be estimated considering the complexity metrics of the new version, with the fault density approach or regression trees). Some recent studies [34], estimating the number of faults during the current version testing, reported that after about a 25% of the total testing time, several SRGMs (both finite and infinite NHPP models) prediction accuracy deviated by only 20%. For a testing resource allocation problem, this means that initially the computed resources allocation will be affected by SRGM errors, but dynamically re-computing the optimal allocation at some time intervals will give more and more accurate results. It is however worth to point out that SRGMs are known to give good results even when data partly violates the model's assumptions [34], [38] they are based on, and their usage is therefore encouraged.

Similar approaches can be used to estimate the parameters for the fault tolerance mechanisms. Depending upon the actual mechanisms used, the failure probability of the failure detection, restart/retry or failover operations need to be estimated. In particular, some potential solutions are:

- Estimating the values from historical data form previous similar systems or common values assumed for the fault tolerance manager components in other critical systems or from the literature.
- Doing a fault injection campaign and obtain:
  - *#Failed detections / #faults injected*
  - *#Failed restarts / #attempted restarts*
  - *#Failed retries / #attempted retries*
  - *#Failed failover / #attempted failover*

It is finally worth to point out that for all the cited parameters expert judgment should not be neglected and should always be used to refine the other estimation method results.
In general, it is difficult to have a common method to assess the quality of the extracted information and to evaluate how much it impacts on results. However, we believe that the most effective way to do this, is to evaluate the impact of the information quality on the specific model where it is used, by performing a sensitivity analysis on all the estimated parameters (as we did in the case study). This method is at once practical and accurate. It allows us to understand and balance possible estimation errors of input parameters (caused by the low quality of extracted information) and their effects on the solution.

VI. Experiments and results

The case study chosen for illustrative purpose is an application, written in C, developed for the European Space Agency. It is a program to provide user interface for the configuration of an array of antennas. The program consists of about 10000 lines of code. Its purpose is to prepare a data file according to a predefined format and characteristics from a user, given the array antenna configuration described using an appropriate Array Definition Language. The program is divided into three main subsystems: the Parser module, the

[6]A basic block, or simply a block, is a sequence of instructions that, except for the last instruction, is free of branches and function calls. The instructions in any basic block are either executed all together, or not at all.

Computational module and the Formatting module.
This case study has been used in other studies about reliability analysis and evaluation (e.g., in [31],[19]),
Figure 2 shows the architectural model of the system.

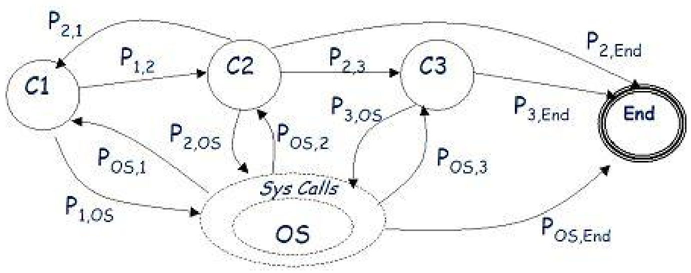

Fig. 2. Software Architecture

The granularity of component (i.e., the level of decomposition), is an analysis choice that depends on the needs. In general, few large components results in easier computational analyses and in a greater amount of available data that allow us to build an accurate SRGM (particularly useful in the case of *complete solution*). The choice of component granularity also depends on how much they are decoupled. High decoupling indicates that components can be more independently tested than with coupled components: this enables the possibility to better schedule testing activities and organize the testing team to work on different components, by using results obtained with the allocation model.
The granularity of component, in this case, is chosen to be a subsystem. No fault tolerance mechanism, as those described above, is present in this system. All the described kinds of solution (i.e., basic, extended and complete, with or without performance, with or without fault tolerance means, single or multiple applications) can be obtained with the implemented tool, depending on the features of the application under study. In the conducted experiments we computed a single-application complete solution, with performance and without fault tolerance means. To estimate the parameters we used a hybrid approach, exploiting dynamic profiling and design/code information. In particular, the experimental procedure is outlined in the following basic steps:

1) Creation of a faulty version of the program, by reinserting faults belonging to real fault set discovered during integration testing and operational usage (Table I). This faulty version emulates the previous version of the application. Note that it is likely that the version of the application we used contains very few faults (except the ones inserted by us), since it has been extensively used without having failures for a long time.
2) Testing execution for the faulty version for a certain amount of total testing time. Only a fraction of the injected faults are removed. During the tests of this "previous version", the application is profiled. From execution traces the DTMC model and the transition probabilities are obtained. The OS is included among the components. Failure data and execution times are also collected during this phase.

TABLE I
TYPES OF INJECTED FAULTS

Fault Types and Subtypes	#Injected Faults	Type Number
Logic omitted or incorrect		
Forgotten cases or steps	4	*1*
Unnecessary Functions	2	*2*
Missing Condition Test	10	*3*
Checking Wrong Variable	4	*4*
Computational Problems		
Equation insufficient or incorrect	20	*5*
Interface incorrect or incomplete		
Module mismatch	6	*6*
Data Handling Problems		
Data initialized incorrectly	4	*7*
Data accessed or stored incorrectly	20	*8*
Total	**70**	

3) Based on the collected failure data, an SRGM for each component is determined. We used SREPT [39] functionalities to obtain the SRGMs.
4) Applying the optimization model, testing times for each component is predicted for the current version. Our tool uses the sequential quadratic programming algorithm to solve the non-linear constrained optimization problem [40]. The current version is then tested according to the computed test times allocation.
5) At the end of the testing, the reliability predicted by the model is compared with the actual achieved reliability, computed as by $(1 - lim_n Nf/N)$, where Nf is the number of observed failures and N is the number of executions of input cases. The prediction error is then analyzed against the possible prediction errors that could occur in the (i) transition probabilities estimation, (ii) in the SRGMs and (iii) in the OS reliability estimation, by carrying out a sensitivity analysis.

A. Results and Analysis

Step 1

According to the described steps, we injected 70 faults in the software (31, 28 and 11 respectively in the component 1, 2 and 3) based on the fault categories of Table I. An excerpt of the injected faults with the corresponding detection testing time and test case number is reported in Table II.

TABLE II
AN EXCERPT OF THE INJECTED FAULTS

Fault Number	Function	Line	Type	Component	Detect. Time	Test Case
2	*Nodedef*	62	8	1	0.124	2
3	*Hexdef*	100	8	1	0.949	7
4	*Nodecoor*	49	8	1	1.074	8
...	...	...	...	...	...	...
37	*Fixsgrel*	99	4	2	2.121	17
38	*Fixsgrpha*	29	4	2	2.414	20
39	*Fixsgrid*	64	4	2	2.987	24
...	...	...	...	...	...	...
68	*Gwrite*	110	7	3	14.412	120

Step 2

Test execution for this faulty version was carried out by randomly generating test cases based on the operational profile

(in this phase, 366 test cases were generated). After the testing phase, 46 faults were removed (respectively 20, 19 and 7), leaving 24 faults in the software. The reliability of this first version of the software was measured executing further 3600 test cases (picked up from the operational profile) and recording the number of failures, without removing the corresponding faults. It amounted to $R = 1 - lim_n Nf/N$ = 0.93583, with $Nf = 231$ and $N = 3600$. We assumed a reliability goal for the next release of R_{MIN} = 0.99; thus the testing resources for the current version are to be allocated according to this goal. We profiled the previous test executions by gprof[7] (for the user functions) and by straceNT[8] (for the system calls), obtaining the execution counts among the components, the corresponding transition probabilities (as described in the previous section) and then the visit counts.
In particular, a visit to an application component is a flow of the control to a user function coming either from a caller user function or from a return by a called function (user function or system call). Thus, a user function F calling another function will have two visits: one from the caller function and another from the return of the called function. The average time per visit to application components during an execution can be computed using the following formula:

$$TV = totalTime/(2 * \#calls + systemCalls - \#termination * averageDepth) \quad (27)$$

where *totalTime* is the average total user function time per execution (an output of *gprof*, averaged over test case executions), and the denominator is the average number of visits per execution to all the application components: *#calls* is the number of user functions called per execution (doubled to consider the return), *systemCalls* is the number of calls to the OS (an output of *sTrace*), not doubled because we are counting the visits to the application components, not also to the OS; *#termination* is the number of terminations (either normal or abnormal) and *averageDepth* is the average depth of the call graph, included in order to subtract the "returns" from a function that are lost due to the termination (this value has been computed by analyzing the output of *gprof*, and was equal to 2.6). Figure 3 clarifies this computation with an example (with a normal terminating single execution and *averageDepth* = 1). Execution counts for a component are computed similarly to the denominator of equation 27. The difference is that only the calls from a component to itself are doubled, in order to consider the return of the control flow to itself. Calls to the other components (and to the OS) are not doubled, because in that case just the "return" from the called component has to be accounted as a visit to the calling one.
However, for a better accuracy, we actually estimated the time per visit for each one of the components, considering:

$$TV_i = totalTime_i/visitCounts_i \quad (28)$$

where *totalTime*$_i$ is the proportion of user time spent in the component *i* (computed as *totalTime* * *#calls*$_i$ / *#calls*) and *visitCounts*$_i$ are computed from transition probabilities (in turn derived from execution counts) with the procedure described in the Section III. Results are summarized in Table III. As for the OS, the visit granularity is slightly different.

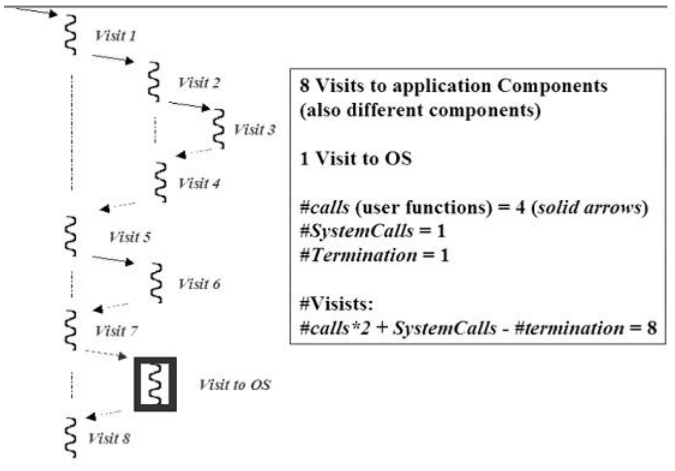
Visit 1
Visit 2
Visit 3
Visit 4
Visit 5
Visit 6
Visit 7
Visit to OS
Visit 8
8 Visits to application Components (also different components)
1 Visit to OS
#calls (user functions) = 4 (solid arrows)
#SystemCalls = 1
#Termination = 1
#Visists:
#calls*2 + SystemCalls - #termination = 8

Fig. 3. Granularity of Visit

Since OS was proprietary, it was not possible to trace the internal function calls (and this was the reason why we did not consider the block as visit granularity); thus the visit in this case is the flow of the control coming from a user function to the OS via a system call; it does not consider the control flow from the OS to itself through internal kernel function calls. Correspondingly, the time per visit will also have a rougher granularity: it is the average time spent in an entire system call execution, computed as the average OS time (obtained by timeit[9]) divided by the average number of system calls (i.e., the average number of visits in this case). Their product is always the total time spent in the OS.
The return of the system call is accounted as a visit for the calling component from the OS, as explained above.
Their values (execution counts/number of visits and times per visit), however, are not of interest for the OS reliability itself, because the OS reliability (i) is given in this case for the entire execution (not as reliability per visit) and (ii) it is assumed constant with time. But the execution counts to and from the OS are important to determine the visit counts for other components.
Also the variance of the visit counts is considered, for the second-order architectural effects. Finally the performance factor is set to 0.1, i.e., only the 10% of testing efforts will be employed for performance testing. Results of this step are summarized in Table III.

Note that since the visit granularity is so fine, the expected visit counts during an execution are very high and the transition probabilities toward the end state are very low (because only a minimal part of code leads to the end). Moreover, also note that the Parser subsystem (component 1) and the Formatting subsystem (component 3) make a great

[7]MinGW (Minimalist GNU for windows) has been used to provide *gprof* and other GNU tools under windows. See http://www.mingw.org/.

[8]straceNT is a system call tracer for Windows See http://www.intellectualheaven.com/.

[9]*timeit* is a command-line tool, provided with Microsoft Windows Resource Kit Tools, that records the time a specified command takes to run.

TABLE III
ESTIMATED PARAMTER VALUES

Minimum Required Reliability					0.99
Reliability of the Previous Version					0.93583
Mean User Process Execution Time					0.01389
Mean OS Execution Time					0.10589
Mean # User Calls					407.4
Mean # System Calls					5442
Transition Probabilities					
TO / FROM	**1**	**2**	**3**	**OS**	**End**
1	0.2307	2.71E-4	0	0.7689	7.79E-5
2	0	0.65253	0.0298	0.3439	4.96E-4
3	0	0	7.72E-4	0.99902	2.10E-4
OS	0.4049	0.0141	0.5810	0	0
END	0	0	0	0	1
Visit Counts	2865.8	222.8	3165.0	5442.3	-
Exec. Time	0.01128	0.00248	1.251E-4	0.10589	-
SRGM: Exponential SRGM. $\lambda(t) = age^{-gt}$					
1		**2**		**3**	
a	*g*	*a*	*g*	*a*	*g*
13	5.46E-2	11	9.46E-2	5	5.34E-2

use of the OS, and the corresponding transition probabilities are significantly higher than the Computational subsystem.
The high values for visit counts could also be due to the first-order DTMC: a first-order DTMC does not allow one to consider the dependence of transition probability from a component i to j on the current as well as the previous components from which the control arrived at component i. However, we also considered the second-order DTMC, detecting no significant changes in the transition probabilities. Thus to keep the treatment simple, the first-order results are considered in the following. Finally, we did not observe failures due to the OS; thus we estimate an OS reliability value equals to 1. This value is clearly an overestimation due to the low number of test cases used to estimate it; thus we start with this value, but it will be varied (*step 5*) between 0.995 and 1.0 to take into account the possible estimation error (this will also show how the overall reliability estimate is affected by the OS reliability value).

Step 3
Based on the interfailure times (Table II) of the previous version testing, an SRGM for each component was built, using SREPT to fit the best model to the data. The fault content parameters for the current version were derived from the estimated remaining fault contents, while the rate parameters were set at the same value of the previous testing process (Table III).
For all of the components, the same kind of SRGM was found to be the best fitting one (i.e., the Goel-Okumoto model): this is mainly due to the strong similarities among the testing processes followed for the three components.
Step 4
With the visit count values, the SRGMs and the OS reliability, the model was built and solved by our tool, giving as output the optimal testing times for each component. Figure 4 and 5 show two stages of the tool, describing a screenshot of the initial architecture configuration and of the parameters configuration of one of the components, respectively.

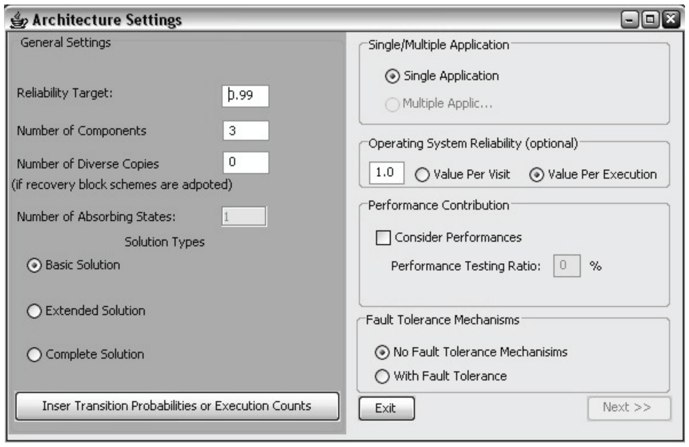
Architecture Settings
General Settings
Reliability Target: 0.99
Number of Components: 3
Number of Diverse Copies: 0
(if recovery block schemes are adpoted)
Number of Absorbing States: 1
Solution Types
[✓ Basic Solution] Extended Solution Complete Solution
Single/Multiple Application: [✓ Single Application] Multiple Applic...
Operating System Reliability (optional): 1.0 Value Per Visit [✓ Value Per Execution]
Performance Contribution: Consider Performances
Performance Testing Ratio: 0 %
Fault Tolerance Mechanisms: [✓ No Fault Tolerance Mechanisims] With Fault Tolerance

Fig. 4. The initial architecture configuration

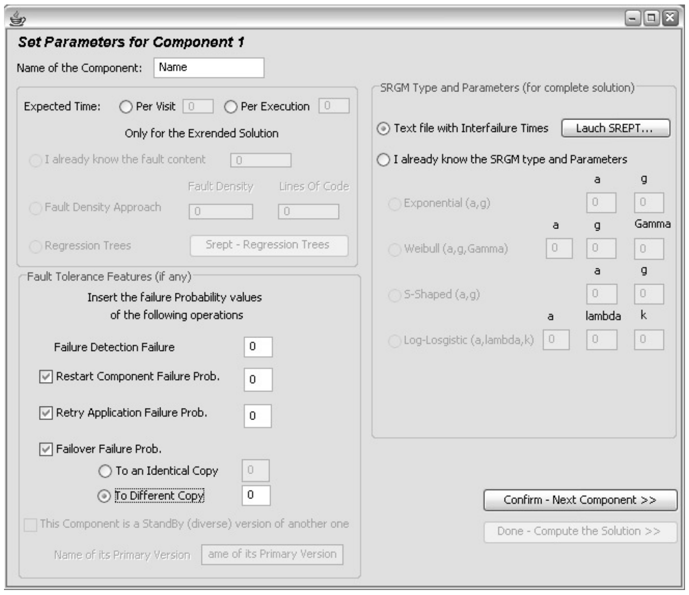
Set Parameters for Component 1
Name of the Component: Name
Expected Time: Per Visit 0 Per Execution 0
Only for the Exrended Solution
I already know the fault content 0
Fault Density Approach — Fault Density 0, Lines Of Code 0
Regression Trees
Fault Tolerance Features (if any)
Insert the failure Probability values of the following operations
Failure Detection Failure: 0
✓ Restart Component Failure Prob.: 0
✓ Retry Application Failure Prob.: 0
✓ Failover Failure Prob.: To an Identical Copy 0 [✓ To Different Copy] 0
This Component is a StandBy (diverse) version of another one
Name of its Primary Version: ame of its Primary Version
SRGM Type and Parameters (for complete solution)
[✓ Text file with Interfailure Times] I already know the SRGM type and Parameters
Exponential (a,g): a 0, g 0
Weibull (a,g,Gamma): a 0, g 0, Gamma 0
S-Shaped (a,g): a 0, g 0
Log-Losgistic (a,lambda,k): a 0, lambda 0, k 0

Fig. 5. The configuration of the parameters of one of the components

After executing the tests (always by generating test cases from the operational profile) according to the optimal testing times, 19 faults of 24 were removed (respectively 11 of 11, 5 of 9 and 3 of 4). Table IV shows the testing times devoted to each component, the corresponding number of executed test cases, the **detection time** and the detecting test case number for each fault. (Clearly the actual devoted time will slightly exceed the allotted time, because the latter is not a perfect multiple of the execution time of a test case).
Step 5
According to the model, the final reliability should be 0.990289. Measuring the actual reliability using the same procedure used for the previous version, it turned out to be 0.989722, with 37 observed failures over 3600 executions. The relative error is about 5.7289E-4 resulting in an overestimation of 0.057289 %.
The main sources of error in the prediction are i) the DTMC transition probabilities and consequently the visit counts, ii)

TABLE IV
TESTING OF THE SYSTEM ACCORDING TO THE MODEL RESULTS

Component	#Fault Removed	Optimal Testing Time	#Test Cases
1	11	40.638	348
2	5	6.315	58
3	3	24.682	215
Fault Number		Detection Time	Test Case Number
Component 1			
21		1.213	10
22		3.112	25
23		6.452	53
24		6.992	57
25		8.346	69
26		12.240	102
27		13.332	111
28		15.341	128
29		22.021	183
30		30.041	250
31		38.098 of 40.638	318 of 348
Component 2			
51		0.933	7
52		2.729	22
53		2.981	24
54		4.065	33
55		6.209 of 6.315	51 of 58
Component 3			
69		5.034	42
70		7.355	61
71		20.442 of 24.682	170 of 215

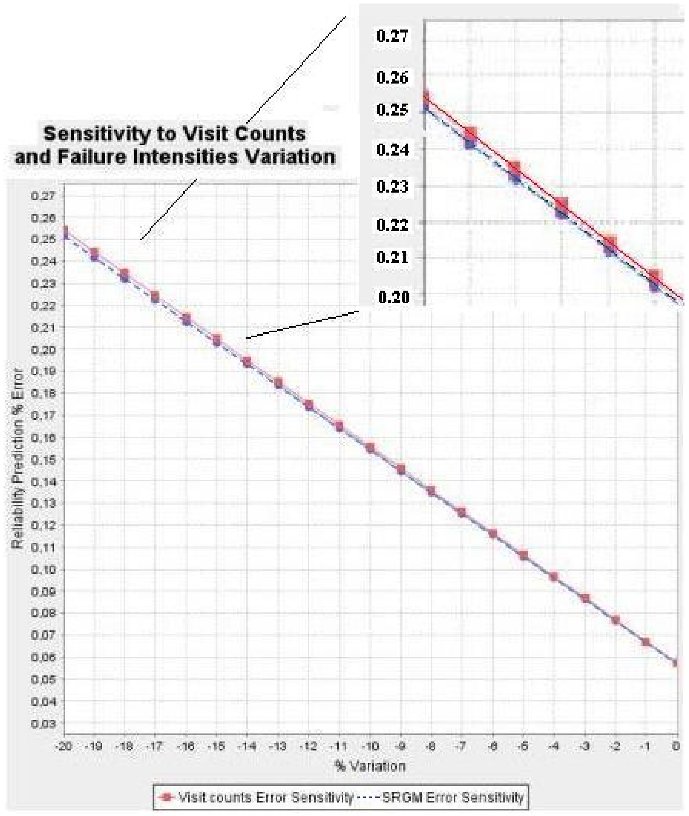

Fig. 6. Sensitivity to visit counts and failure intensities variation

the SRGMs prediction ability and (iii) the OS reliability. In our case the architecture of the application and the components themselves have not significantly changed as well as the testing process between the two versions. Such a good prediction is also due to this. However, we can see how in the presence of significant changes in such values, the prediction is still good. First, figure 6 shows the variation of the percentage relative error in the reliability prediction against the variation of the percentage relative error in all the visit counts (solid line). For a maximum of 20% in the visit counts estimation errors (underestimation) we have a reliability prediction error of about 0.257% , i.e., a prediction of about 0.99226.

Second, in the same figure (figure 6, dashed line) the same variation in the prediction against the percentage relative error in the failure intensities is shown: if an SRGM does not accurately describe the testing process, the value of the achieved failure intensities at the end of the testing time for each component will be affected. We evaluated such effect by varying the failure intensities. Results in figure 6 show that the reliability prediction will be affected by an error of 0.256% , i.e. 0.99225, for a percentage variation of 20% in the failure intensities (underestimation). Failure intensities plot shows almost the same behavior as the visit counts plot, because in the reliability computation their product appear in the exponents (the slight difference is due to the second-order part in the formula).

Third, since no OS failures have been observed, we estimated the OS reliability to be equal to 1; figure 7 shows the effect of an estimation error in the OS reliability on the overall reliability prediction. In this case the overall prediction is more sensitive to OS reliability prediction errors than the previous parameters.

However, the previous percentage errors simply do not make sense in this case; it is very unlikely to mistake the OS reliability prediction with an error of 20%. For instance, suppose that in our case an OS failure over 3600 execution tests has been observed and we estimate the reliability as (1-1/3600) = 0.9997; an error of 20% would mean making an estimation of 0.79976, that in our case would correspond to about 720 failures: i.e., in the current version the OS has experiences 719 failures more than the previous version over only 3600 test cases. It is a huge prediction mistake. Moreover, OS has usually much more historical failure data than the other application components; this makes OS reliability estimation easier and more accurate. Figure 7 shows therefore the percentage error variation of the overall reliability depending on the absolute variation of OS reliability in a reasonable range.

In this case, for a maximum error of 0.005 (i.e., the 0.5%), the prediction error shifts from the original overestimation of 0.057289% with respect to the actual reliability to an underestimation of about 0.443% , i.e., a reliability of 0.98534. The solution is therefore much more sensitive to the OS reliability estimation than the other parameters.

Finally, all the results have been obtained with the estimated operational profile; changing the operational profile will yield different results. The effect of the operational profile variation on reliability measurements (particularly on SRGMs) has been studied by Pasquini et al. in [31], which however pointed out that the "predictive accuracy of the models is not heavily

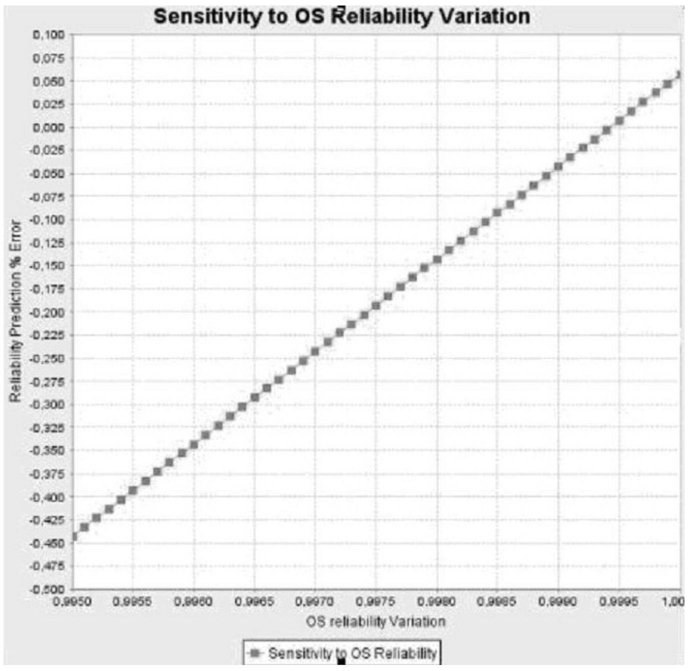

Fig. 7. Sensitivity to OS Reliability variation

affected by errors in the OP-estimate. In general, if a model-estimate fits the actual reliability for a correct OP-estimate then it fits well even if the OP-estimate is erroneous".

VII. Conclusion and future work

In this paper we proposed an optimization model to allocate the testing resources to different system components in order for the system to achieve a required reliability level at minimum verification costs. The purpose of the model, through the tool implementing it, is therefore to drive engineers in the verification phase. An architecture-based model (in particular a DTMC) was used to describe the software architecture. The optimization model was used for providing flexible solutions, at different levels of detail, according to the information provided by the user. The model includes the possibility to consider the OS as a component and the potential fault tolerance mechanisms a component could employ. Moreover, performance testing times and second order architectural effects were considered. Experiments showed the prediction ability of the model and its sensitivity to the variation of potential sources of errors. The implemented tool used the SQP algorithm for the exact solution and a genetic algorithm for the heuristic solution, useful for solving multiple-applications problems.

Even though the presented model aims to allocate testing times for each component, its output contributes, as a side effect, to a better integration testing (i.e., identifying the most critical components, an integration strategy by critical modules is eased) and to the system test cases generation (i.e., from the DTMC describing the application execution a markov-chain based testing can be adopted, without additional efforts). Besides, the adopted architecture-based solution allows us to obtain information about the sensitivity of some components and their impact on the final solution that are useful to successive designs for future versions.

As future work, we plan to extend the approach to concurrent systems by describing the architecture through a SPN (stochastic Petri Net) or a DAG (Direct Acyclic Graph) [25],[26]. Moreover, we plan to include other fault tolerance mechanisms (e.g., N-version programming) to be able to describe more systems. A better investigation on the performance-testing time relations and on the OS influence is also desirable. Finally, considering the testing strategies influence on the testing process of a component, and hence on its SRGM, we plan to include, as output, a rough indication of the testing strategies to be used for each component in addition to the testing times.

References

[1] DO-178B/ED12B, "Software consideration in airborne systems and equipment certification," RTCA and EUROCAE, Dec. 1992.

[2] SAF.ET1.ST03.1000-MAN-01, "Air Navigation System Safety Assessment Methodology (v2-0)," EUROCONTROL EATMP Safety Management, Apr. 2004.

[3] N. Wattanapongsakorn and S. P. Levitan, "Reliability optimization models for embedded systems with multiple applications," IEEE Trans. on Reliability, vol. 53, no. 3, Sep 2004.

[4] J. Onishi, S. Kimura, R.J.W. James, and Y. Nakagawa, "Solving the Redundancy Allocation Problem With a Mix of Components Using the Improved Surrogate Constraint Method," IEEE Trans. on Reliability, vol. 56, no. 1, Mar 2007.

[5] F. W. Rice, C.R. Cassady and R.T. Wise, "Simplifying the solution of redundancy allocation problems," Proc. of the Annual Reliability & Maintainability Symposium (RAMS '99), pp. 190-194, 1999.

[6] M.R. Lyu, S. Rangarajan and A.P.A. van Moorsel, "Optimal Allocation of Test Resources for Software Reliability Growth Modeling in Software Development," IEEE Trans. on Reliability, vol. 51, no. 2, June 2002.

[7] R.H. Hou, S.Y. Kuo, and Y.P. Chang, "Efficient allocation of testing resources for software module testing based on the hyper-geometric distribution software reliability growth model," Proc. of the 7th International Symposium on Software Reliability Engineering (ISSRE '96), pp. 289-298, Oct./Nov. 1996.

[8] Y. Nakagawa and S. Miyazaki, "Surrogate constraints algorithm for reliability optimization problems with two constraints," IEEE Trans. Reliability, vol. R-30, pp. 175-181, 1981.

[9] L. Painton and J. Campbell, "Genetic algorithms in optimization of system reliability," IEEE Trans. on Reliability, vol. 44, pp. 172-178, 1995.

[10] F. A. Tillman, C. L. Hwang, and W. Kuo, "Determining component reliability and redundancy for optimum system reliability," IEEE Trans. on Reliability, vol. R-26, pp. 162-165, 1977.

[11] A.O.C. Elegbede, C.Chu, K.H. Adjallah, and F.Yalaoui "Reliability Allocation Through Cost Minimization," IEEE Trans. on Reliability, vol. 52, no. 1, Mar 2003.

[12] Rani, R.B. Misra, "Economic Allocation of Target Reliability in Modular Software Systems," Proc. of the Annual Reliability & Maintainability Symposium (RAMS '05), pp. 428- 432, 2005.

[13] A.Mettas, "Reliability Allocation and optimization for complex systems," Proc. of the Annual Reliability and Maintainability Symposium (RAMS '00), pp. 216-221, 2000.

[14] M. R. Lyu, S. Rangarajan, A.P.A. van Moorsel, "Optimization of Reliability Allocation and Testing Schedule for Software Systems," Proc. of the 8th International Symposium On Software Reliability Engineering (ISSRE '97), pp. 336-347,1997.

[15] M. E.Helander, M.Zhao, N.Ohisson, "Planning Models for software reliability and Cost," IEEE Trans. on Software Engineering, vol. 24, No. 6, June 1998.

[16] W.Everett, "Software Component Reliability Analysis," Proc. of the Symp. Application specific Systems and Software Engineering Technology (ASSET '99), pp. 204-211, 1999.

[17] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," Performance Evaluation, vol. 45, issue 2-3, pp. 179-204, 2001.

[18] S.S. Gokhale, W. E.Wong, J.R. Horganc, K. S. Trivedi, "An analytical approach to architecture-based software performance and reliability prediction," Performance Evaluation, vol. 58, issue 4, pp. 391-412, 2004.

[19] K. Goseva-Popstojanova, A.P. Mathur, K.S. Trivedi, “Comparison of architecture-based software reliability models,” Proc. of the 12th International Symposium on Software Reliability Engineering (ISSRE ’01), pp. 22-31, 2001.
[20] S. Gokhale, M.R. Lyu, K.S. Trivedi, “Incorporating fault debugging activities into software reliability models: A simulation approach,” IEEE Trans. on Reliability, vol. 55, No. 2, pp. 281–292, June 2006.
[21] W.Wang, Y.Wu, M.H. Chen, “An architecture-based software reliability model,” Proc. of the Pacific Rim Dependability Symposium, 1999.
[22] S. Gokhale, K.S. Trivedi, “Time/Structure Based Software Reliability Model,” Annals of Software Engineering, vol. 8, pp. 85-121, 1999.
[23] A. Reibman, K.S.Trivedi, “Numerical transient analysis of Markov models,” Computers & Operations Research, vol. 15, n. 1, pp. 19-36,1988.
[24] S.S. Gokhale, K.S. Trivedi, “Reliability prediction and sensitivity analysis based on software architecture,”Proc. of the 13th International Symposium on Software Reliability Engineering (ISSRE’02), p. 64, 2002.
[25] K.S. Trivedi, “Probability and Statistics with Reliability, Queuing and Computer Science Applications,” John Wiley and Sons, 2001.
[26] R.A. Sahner, K.S. Trivedi, A. Puliafito, “Performance and Reliability Analysis of Computer Systems: An Example-Based Approach using the SHARPE Software Package”, Kluwer Academic Publishers, Boston,1996.
[27] C. Huang, S. Kuo, and M.R. Lyu, “An Assessment of Testing-Effort Dependent Software Reliability Growth Models,” IEEE Trans. On Reliability, vol. 56, no. 2, June 2007.
[28] V.S.Sharma, K.S.Trivedi, “Quantifying software performance, reliability and security: An architecture-based approach,” The Journal of Systems and Software, vol. 80, Issue 4. pp. 493-509, April 2007.
[29] S. Yamada, H. Ohtera, and H. Narihisa, “Software reliability growth models with testing effort,” IEEE Trans. on Reliability, vol. R-35, pp. 19-23, Apr. 1986.
[30] C. Huang and M.R. Lyu,“Optimal Release Time for Software Systems Considering Cost, Testing-Effort, and Test Efficiency,” IEEE Trans. On Reliability, vol. 54, no. 4, Dec 2005.
[31] A. Pasquini, A. N. Crespo, and P. Matrella, “Sensitivity of reliability-growth models to operational profile errors vs testing accuracy,” IEEE Trans. on Reliability, vol. 45, no. 4, pp. 531-540, 1996.
[32] S.Gokhale, M.R.Lyu,“Regression Tree Modeling for the Prediction of Software Quality,” Proc. of 3rd ISSAT International Conference On Reliability (ISSAT ’97), pp. 31- 36, Mar 1997.
[33] M. Lipow, “Number Of Faults Per Line of Code,“IEEE Trans. on Software Engineering, Vol. 8, no. 4, pp. 437-439, July 1982.
[34] V. Almering, M. Van Genuchten,G. Cloudt, P.J.M. Sonnemans, “Using Software Reliability Growth Models in Practice,” IEEE Software, vol. 24, no. 6, pp. 82-88, Nov.-Dec. 2007.
[35] A. Srivastava, A. Eustace, “Atom: a system for building customized program analysis tools,” In: Proc. of the SIGPLAN’94 Conference on Programming Language Design and Implementation, pp. 196-1349, 1994.
[36] S. Yacoub, B. Cukic, and H.H. Ammar,“A Scenario-Based Reliability Analysis Approach for Component-Based Software,” IEEE Trans. on Reliability, vol. 53, no. 4, Dec. 2004.
[37] S.S. Gokhale, T. Philip, P.N. Marinos, K.S. Trivedi, “Unification of finite failure non-homogeneous Poisson process models through test coverage,” Proc. of the 7th International Symposium on Software Reliability Engineering (ISSRE ’96), 1996.
[38] C. Stringfellow and A.A. Andrews, “An Empirical Method for Selecting Software Reliability Growth Models,” Empirical Software Eng., vol. 7, no. 4, pp. 297-318, 2002.
[39] S. Ramani, S. Gokhale and K.Trivedi, “SREPT: Software Reliability Estimation and Prediction Tool,” Performance Evaluation, special issue on Tools for performance evaluation, 2000.
[40] K. Schittkowski, “NLQPL: A FORTRAN-Subroutine Solving Constrained Nonlinear Programming Problems,” Annals of Operations Research, Vol. 5, pp 485-500, 1985.

Roberto Pietrantuono received the BS degree (2003) and the MS degree (2006) in computer engineering from the Federico II University of Naples, Italy. He is a PhD student in computer and automation engineering at the Department of Computer and Systems Engineering (DIS) of the Federico II University of Naples. His research interests are in the area of software engineering, particularly in the software verification of critical systems, and software reliability modelling.

Stefano Russo , PhD, IEEE Member, is Professor and Deputy Dean at the Department of Computer and Systems Engineering (DIS) of the Federico II University of Naples, Italy, where he leads the Distributed and Mobile Computing Group. He was Assistant Professor at DIS from 1994 to 1998, and Associate Professor from 1998 to 2002. He is Chair of the Curriculum in Computer Engineering, and Director of the National Laboratory of CINI (National Inter-universities Consortium for Informatics) in Naples. He teaches the courses of Advanced Programming and Distributed Systems. His current scientific interests are in the following areas: (i) distributed software engineering; (ii) middleware technologies; (iii) mobile computing.

Kishor S. Trivedi holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, Durham, NC. He has been on the Duke faculty since 1975. He is the author of a well known text entitled, Probability and Statistics with Reliability, Queuing and Computer Science Applications, published by Prentice-Hall; a thoroughly revised second edition (including its Indian edition) of this book has been published by John Wiley. He has also published two other books entitled, Performance and Reliability Analysis of Computer Systems, published by Kluwer Academic Publishers and Queueing Networks and Markov Chains, John Wiley. He is a Fellow of the Institute of Electrical and Electronics Engineers. He is a Golden Core Member of IEEE Computer Society. He has published over 420 articles and has supervised 42 Ph.D. dissertations. He is on the editorial boards of IEEE Transactions on dependable and secure computing, Journal of risk and reliability, international journal of performability engineering and international journal of quality and safety engineering. He is the recipient of IEEE Computer Society Technical Achievement Award for his research on Software Aging and Rejuvenation. His research interests in are in reliability, availability, performance, performability and survivability modeling of computer and communication systems. He works closely with industry in carrying our reliability/availability analysis, providing short courses on reliability, availability, performability modeling and in the development and dissemination of software packages such as HARP, SHARPE, SPNP and SREPT.