Online Monitoring of Software System Reliability

R. Pietrantuono*, S. Russo*†, K. S. Trivedi‡
*Dipartimento di Informatica e Sistemistica, Università degli studi di Napoli Federico II, Via Claudio 21, 80125, Naples, Italy.
†Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Ed. 1, Via Cinthia, 80126, Naples, Italy.
‡Department of Electrical and Computer Engineering, Duke University,Durham, NC 27708.

Email: {roberto.pietrantuono,stefano.russo}@unina.it, kst@ee.duke.edu

***Abstract*—Reliability is one of the major concerns for software engineers. The increasing size of software systems and their inherent complexity - which is essentially related to the intricate interdependencies among many heterogeneous components - pose serious difficulties to its assessment and assurance. The actual system runtime behavior is difficult to forecast during the development phase, and just relying upon sound design and testing techniques is often not sufficient to deliver highly reliable systems. In order to guarantee high reliability, system behavior needs to be monitored at runtime and its reliability needs to be periodically estimated during operation, taking into account both structural/static and behavioral/dynamic information. In this paper, we propose an online reliability monitoring approach, which combines static reliability modeling and dynamic analysis to periodically evaluate system reliability trend during operation. Its usage is illustrated by a prototype implementation and a case-study.**

I. Introduction

Software is known to be the main source of system failures and as one of the weakest link in system reliability. Assessing software reliability is of paramount importance for mission-228critical systems. Engineers need to have quantifiable evidence to evaluate software reliability, and, if required, to adopt proper actions aiming at assuring a desired reliability level.
However, this is not a trivial issue. Understanding and managing software is increasingly difficult, due to the increasing size of systems, the heterogeneity of their components, and the intricate interdependencies and interactions among them. Indeed, software architects and developers, make increasing use of OTS (off-the-shelf) software items (including Operating Systems, third-party libraries and virtual machines) to build their systems, even in critical contexts, in order to cope with the huge systems dimension and complexity. However, though this leads to significant improvement as for time-to-market constraints, their heterogeneity, the unknown interactions side-effects and their incomplete specifications inevitably lead to intricate interconnections and subtle dependencies, posing tricky issues that traditional methods for assessment need to cope with. Such issues, exacerbated by the still debated nature of software faults, make system runtime behavior difficult to forecast based on testing, and make it very tough to carry out accurate assessments of dependability attributes.
As other dependability attributes, reliability is usually estimated in the development phase in a static way, by using modeling techniques that are assumed to be representative also of the runtime phase. By adopting this approach, a stochastic model of the system is developed and solved analytically or via discrete-event simulation. The result allows one to predict the attribute of interest. In the case of software reliability, the assessment is often done during the testing phase, e.g., by collecting interfailure times and by fitting a model. However, this kind of estimation does not allow one to deal with the described issues, since runtime behavior is neglected. The result may be not accurate, due to the necessary simplifying assumptions that undermine the model representativeness in the real operational environment, and that need to be made when real operational data are not available.

While this inaccuracy can be accepted when the estimation is performed to implement optimal release (and testing) policies, it cannot be accepted when it is used as a basis to apply proper actions aiming at preventing runtime system failures (e.g., unneeded actions are applied or, even worse, needed actions may be not applied). We believe that the actual reliability of an operational system has to be regarded as depending on two complementary features: (i) its structure, determined by system components, their reliability and their interdependencies, and (ii) its runtime behavior, which can be significantly different from the one observed during testing. Thus, reliability assessment in the development phase does not suffice, if we want to provide accurate estimations and implement reliability assurance policies.
On the other hand, just using operational data, without a model that is able to give preliminary estimates, has several shortcomings: (i) first, to provide confident results, we should wait for several system failures to get sufficient data; this would prevent proactive actions from being applied before such failure data are available. (ii) Second, model-based approaches are able to provide a preliminary estimate, even if not accurate, before the operational phase starts. (iii) Third, the usage of models allows for suitable analyses, that may be useful to evaluate individual component behavior, their interactions and their impact on overall system reliability.

In this paper, we attempt to bring together these two extremes. We propose a method to carry out runtime reliability estimation, based on a preliminary modeling phase followed by a refinement phase, where real operational data are used to counterbalance potential errors due to model simplifications.

To give accurate estimates, the proposed solution aims to integrate and exploit modeling power with representativeness of real operational data. The basic idea is to utilize an architecture-based software reliability model together with a dynamic analysis tool in order to (i) give a preliminary estimate when software is released (i.e., after testing) and then (ii) to continuously refine the model at runtime on the basis of information that becomes available as the system execution proceeds. A prototype version of the monitoring system is implemented, that is initially trained with the reference model and the preliminary reliability estimation, and then uses operational data to compute the online reliability level. The prototype is evaluated on a case-study consisting of an application in the field of queuing systems simulation.

II. Background and Related Work

A. Reliability Evaluation

Reliability can be evaluated by using several approaches, generally classified into two categories: model-based and measurements-based. Each of them shows different peculiarities, which determine the suitability of the method for the analysis of a specific system aspect. Model-based approaches are widely used for reliability evaluation of complex software/hardware systems. They are based on the construction of a model that is a "convenient" abstraction of the system, with enough level of detail to represent the aspects of interest for the evaluation. A number of modeling approaches have appeared in the literature:

1) compositional approaches (e.g., [1], [2], [3], [4]), where the system model is constructed in a bottom-up fashion. The models representing parts of the system are built in isolation, and then composed via suitable operators and composition rules;
2) decomposition/aggregation approaches (e.g., [5], [6], [7], [8]), where the overall model is divided into simpler and more tractable sub-models, and the measures obtained from their solution are then aggregated to compute those concerning the overall model;
3) derivation of dependability models from high-level specification, e.g. from UML design (e.g., [9]).

When the model is required to capture and analyze the attributes of interest from the architectural point of view (i.e., considering the system as components and their interactions), architecture-based models are sought. With the advent of object-oriented and component-based systems, these models have increasingly been adopted for performance and reliability evaluation [10], [11], [12], [13]. The software architecture is usually extracted from design, source code or even object code, and the level of decomposition (i.e., component granularity) is defined depending on the needs. Architecture-based models are categorized as [14]:

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

In this work we adopt a state-based model to represent the software architecture. Models, in general, are very useful for their ability to abstract from unnecessary details, and allow to suitably analyze the architecture, to pinpoint performance/reliability bottlenecks, and to compare design alternatives without physical implementation. However, they may be not accurate enough, when the input parameter values are not representative of the real system behavior. A measurements-based approach may allow for more accurate results: it is based on real operational data (from the system or its prototype) and the usage of statistical inference techniques. It is an attractive option for assessing an existing system or prototype, and constitutes an effective way to assess the efficiency of fault tolerance mechanisms and to obtain the detailed characterization of the system behavior (or parts of it) in presence of faults. However, since real data are needed, it is not always possible to apply this approach. Moreover, just relying on measurement-based approach does not yield insight into the complex dependencies among components, and does not allow system analysis from a more general point of view. It is often more convenient to make measurements at the individual component/subsystem level rather than on the system as a whole, and then to feed them in a model [15]. Although the most of papers use either the model-based or the measurement-based approach, some papers use a combined approach, even if not producing results in an online manner [16], [17]. An online monitoring system combining both the approaches is in [18], [19]. but it addresses system *availability* evaluation. The approach proposed in this paper focuses on online autonomic reliability management, by combining both the model-based and the measurements-based evaluation methods.

B. Dynamic analysis

In order to evaluate the system reliability at runtime, we need a way to describe not only the system architecture (that is a static description), but also its dynamic behavior. The most attractive option is to monitor the execution, to analyze the resultant execution traces and give a description of the observed behavior (i.e., a behavioral model). The usage of dynamic analysis tools seems to be the best solution for this. Dynamic analysis aims to give information about the system by analyzing its execution traces. It attempts to overcome the static analysis limitations, (where all the source code is analyzed in order to verify some property of interest), such as the difficulty to cope with large dimensions, with the extreme dynamism of systems and with the use of OTS items (where the source code is often not available). There are several dynamic analysis tools (e.g., [20], [21], [22], [23]). For our purpose, we rely on one of the most successful inferential engines, that is Daikon [23]. Daikon is a tool that aims to infer likely invariants [1] from execution traces. In particular, it focuses on Input/Output (I/O) invariants, that are invariants on exchanged argument values at the entry/exit point of a function, by considering more than 160 invariant templates. An example of such invariants over a single variable x may be the relation $a < x < b$; whereas an invariant involving two variables may require them to respect the relation $x < y$ (see [23] for a list of invariants inferred by Daikon). In order to build invariants, Daikon starts with a set of syntactic constraints for the monitored variables, and incrementally considers the input values. At each step, it eliminates the constraints violated by the value to obtain a set of constraints satisfied by all inputs. Statistical considerations allow Daikon to identify constraints that are verified incidentally (this is an important feature for our purpose, as detailed in the following sections). In particular, invariants are identified by:

1) Instrumentation, execution and monitoring of the application;
2) Recording of the I/O (Input/Output) behaviors;
3) Determination of the invariants, by the analysis of the collected traces. The monitored variables are combined with each other to form Boolean expressions to be compared with the actual observed execution values and potential invariants are generated by attempting to infer possible relations among the variables.

Daikon identifies invariants at specific points of the program; we are interested in using it for deriving constraints on exchanged argument values in the I/O flow among components (i.e., at their interfaces). This will be useful in the runtime phase.

III. The Monitoring System

The goal of the proposed monitoring approach is to give an estimate of the actual runtime reliability, R_{ONLINE}, to be compared with the expected reliability, R_{EXP}, estimated at the end of the testing phase by the model. If the runtime reliability is lower than the expected reliability for a given threshold quantity Thr, an alarm is triggered. This indicates that the probability of system failure at time *t* is greater than expected. The basic idea is to utilize an architecture-based model together with a dynamic analysis tool at runtime to evaluate the online system reliability, by using the model fed by operational data. Runtime estimation aims at removing errors introduced by the assumptions of the model built in the testing phase. In particular, we used an absorbing DTMC as architecture-based model, which describes the software components as states and the flow of control among them as transition probabilities [10][11].Other architecture-based models could be used, without loss of generality. The usage of architecture-based models (rather than other kinds of reliability estimation models) is required to have a fine-grained description of the system, where the contribution of individual component reliability and of their interactions to the overall reliability can be clearly distinguished. This allows us to adjust the estimation in the runtime phase by independently adjusting the estimations of components reliability and the values describing the interactions among them.

[1]An invariant is a property in a program, described by a relation, that must be always true

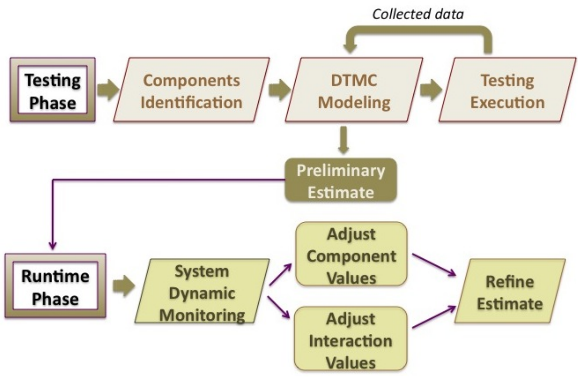

Fig. 1: The Monitoring Process

Figure 1 outlines the process. The first step requires engineers to establish component granularity and, as a consequence, identify components to be represented as states of a DTMC. Then, during the testing execution, data about components behavior and visits among components are collected and used to feed the DTMC model parameters, as detailed in the next subsections. During operation, the system is monitored in order to detect differences between the real runtime behavior and the behavior observed in the testing phase; data collected are used to adjust values describing interactions among components and their individual reliability. By using the same DTMC model as in the testing phase, updated with adjusted values, a refined reliability estimate is obtained, reflecting the system current behavior. In the following, both phases are detailed.

Note that components granularity, which defines how the system is decomposed, is an analysis choice. Engineers have to decide whether to represent the system as a large number of small components, or as a small number of large units. [2]

A. Modelling phase

Once the level of decomposition is decided, the system is represented by an absorbing DTMC (suitable for terminating applications). 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

[2]In the context of architecture-based analysis a component is intended as a logically independent unit performing a well-defined function [11]. Choosing many small components allows a more accurate identification of those parts that mostly affect reliability, but leads to difficulties in measurements, parameterization and solution; while using few large components leads to easier computations with the risk of neglecting important details about the internal structure.

components and the transition from state *i* to state *j* represents the transfer of control from component *i* to component *j*, with the associated transition probabilities. The goal of the DTMC model is to represent how system reliability depends on components reliability and on their interactions.
Transition probabilities can be used to compute the expected number of times a component is visited during an execution. These values are also known as *Visit Counts* [24]. To compute them, a possible way is the following: the one-step transition probability matrix of an absorbing DTMC with *n* states and *m* absorbing states is partitioned as:

$$P = \begin{pmatrix} Q & C \\ 0 & I \end{pmatrix} \quad (1)$$

where Q is an *(n-m)* by *(n-m)* stochastic submatrix (with at least one row sum < 1), I is an *m* by *m* identity matrix, 0 is an *m* by *(n -m)* matrix of zeroes and C an *(n-m)* by *m* matrix. Then, 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 s_j from the state s_i after k steps), it can be shown [24] that the so-called fundamental matrix M is obtained as

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

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 state *i* to state *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:

$$v_{1,j} = m_{1,j} \quad (3)$$

We denote visit counts with V_j = $v_{1,j}$. They are particularly useful to describe the usage of each component in the application control flow. After having built the model, the next step is about the collection of information needed to feed input parameters, and thus to give a preliminary reliability estimate. This information is gathered during the testing phase.

In particular, two kinds of data are needed: (i) data about component failures that have to be used to estimate their reliability; (ii) data about control flow among components (i.e., the number of times the control flows from a component to another, also known as *execution counts* [10]) to compute transition probabilities and then the *visit counts*. As for component reliability, it can be estimated by using the following formula:

$$R_i \approx 1 - \lim_{n_i \to \infty} \frac{f_i}{n_i} \quad (4)$$

where f_i is the number of failures of component *i* and n_i is the number of executions of component *i* in N randomly generated executions. To give a correct estimate with this method, authors in [11] explain that when a failure occurs during the executions, the corresponding fault has not to be removed (i.e, the system has not to be altered during the measurement). Thus, other than test cases (where faults are instead removed), this method requires additional executions to be run. However, we can take advantage of these executions, because by profiling them we can further collect data for building a more robust model (e.g., we obtain more data about transitions among components and data for the adopted dynamic analysis technique), as detailed in the next sections. An alternative method could employ a software reliability growth model (SRGM), built by using interfailure times, fitting a failure intensity model, and by taking its values at the end of the testing phase.

As for *visit counts* computation, we need to obtain the *Execution counts*. They can be inferred by profiling the test cases execution (with a tool like *gprof*[3], or by analyzing the output of the adopted dynamic analysis tool, i.e., *Daikon*), from which the number of transfers from a component to another can be derived (e.g., *gprof* reports the number of times each function has been visited, as well as the calling function). Transition probabilities are estimated as *ExecutionCounts*$_{i,j}$/$\sum_j ExecutionCounts_{i,j}$. From transition probabilities, *visit counts* are obtained as described above.
By using the DTMC model and the estimated parameters, the system reliability is computed as described in [10], i.e.:

$$E[R] \approx \prod_{i}^{n} E[R_i^{X_{1,i}}] = (\prod_{i}^{n-1} R_i^{E[X_{1,i}]})R_n = (\prod_{i}^{n-1} R_i^{V_i})R_n \quad (5)$$

where $X_{1,i}$ denotes he number of visits from the state 1 to the state *i* before absorption and E[$X_{1,i}$] is the expected number of visits to component *i* ($X_{1,n}$ is always 1 for the final component *n*), i.e., the *visit counts* (V_i). Second-order architectural effects can also be considered as in [25] for more accurate result. The described DTMC model, that will be used also in the runtime phase, gives a preliminary estimate of system reliability. However its accuracy depends on how the assumptions it relies on are verified. In particular, what affects the accuracy of the theoretical estimation given in the eq. 5, may be summarized by the following assumptions:

- First-order Markov chain (this assumption affects the visit counts estimation, since the control flow transitions from a particular component are assumed to not depend on the path taken to reach this component);
- Components fail independently and component failure leads to the system failure (it is a conservative assumption, that leads to an underestimation; a correlated failure adds to the failure probability of individual components);
- When every kind of reliability model is applied in the testing stage, the underlying assumption is that test cases execution does not reflect the real operational profile (even if Pasquini et al. [26] shown that the impact of the operational profile estimation error is not high).

B. Runtime Phase

To overcome these limitations, the runtime refinement phase is based on the following observations: the error between the

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

reliability estimated by the model and the actual reliability may be due (as a consequence of the assumptions made) to (i) the estimation error of expected visit counts, and (ii) the error made by assigning a reliability value to components on the base of collected data coming from testing; in this case the error is due to the difference in the "behavior" caused by the non-correspondence between the real operational profile and the test cases execution. The runtime monitoring system will refine the estimation by observing the real behavior. As for the first type of error, we need to monitor the interactions among components in order to record real "visits" among them, by collecting the execution counts, and coming to an estimate of V_i values as described in the previous section.

Despite this estimation, reliability values of the single component may be, as stated, affected by the second type of error. In this case, it is not possible to estimate the actual value during execution, since, in order to get failure data, the system should fail (and the estimation does not make sense anymore). What we propose is (i) to monitor components with Daikon, that captures the interactions at components interface level, and builds components behavioral model (inferring I/O invariants), (ii) and then to detect at runtime deviations from the defined expected behavior, i.e., violations to the inferred invariants. Examples of useful invariants are relationships over exchanged argument values among function calls: over a single numeric variable (e.g., range limits: $x < a$; $x < b$, and $a < x < b$, nonzero: $x \neq 0$, modulus: $x \bmod b = a$), and over two numeric variables (e.g., linear relationship: $y = ax + b$, ordering comparison: $x < y$; $x > y$;$x = y$, functions: $y = fn(x)$). See [23] for a full description of possible invariants.

In particular, the built model represents the expected, and thus supposed "correct", behavior. Of course, it is an "estimate" of the correct behavior, because the testing does not cover all the possible correct behaviors. In the operational phase, if the observed behavior is different from the expected one, then, it is no longer guaranteed that the system behaves as in the testing phase and it might fail earlier than expected. Thus, considering a reliability estimate R_{EXPi} for component i, obtained during the testing, we need to identify a "penalty function" that properly lowers this value, each time the component interacts with other components in unexpected ways (see next section). The monitoring architecture is depicted in Figure 2. The same DTMC model is used in the testing and in the operational phase. In the operational phase the monitoring tool uses real collected data to estimate visit counts and component reliabilities, updating the model; the monitor is responsible for triggering alarms when the actual estimated reliability is lower than the expected reliability. The monitor is also able to provide some insights into the cause of possible deviating behaviors, as mentioned in the next section.

IV. Estimating the Reliability Degradation

In the runtime phase, the online behaviors can be checked by comparing the current execution trace with the invariants built in the testing phase. In order to take into account the new

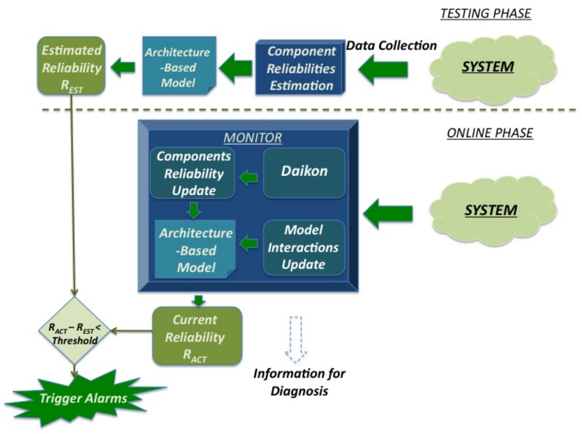
TESTING PHASE
Estimated Reliability R_{EST}
Architecture -Based Model
Component Reliabilities Estimation
Data Collection
SYSTEM
ONLINE PHASE
MONITOR
Components Reliability Update
Daikon
Architecture -Based Model
Model Interactions Update
SYSTEM
$R_{ACT} - R_{EST} <$ Threshold
Current Reliability R_{ACT}
Information for Diagnosis
Trigger Alarms

Fig. 2: Monitoring System.

unexpected behaviors that components exhibit, component reliabilities need to be diminished. However they have to be diminished in a proper way, since a deviation from the expected behavior (we call it *violation*) can represent either an incorrect behavior or can be a false-positive (i.e., with respect to the behavior observed during the testing, the deviation is a new, unexpected, but correct, behavior).

The evaluation of the *penalty values* to be used to lower the reliability of components is carried out periodically, at each time interval T, when the overall reliability estimation is computed. They aim to estimate the risk associated with the set of all violations that occurred in the instrumented program points in the considered time interval, for each component *i*. *This is the risk of the observed violations to be representative of incorrect behaviors, we call it* ***Risk Factor (RF)***. The higher the risk for component i, the lower its reliability should be. *Risk Factor* values depend (i) on how many violations occurred in the considered period of observation, (ii) on how many distinct monitored program points (more precisely, the distinct monitored parameters) experienced violations in the same period of observation, and (iii) on the robustness of the built model (i.e., the confidence that can be given to the built invariants). The first two points are easily computable by observing the Daikon output. The risk factor (RF) has to be proportional to them, since the higher is the number of violations and the number of distinct parameters involved in a violation, the higher is the risk of incorrect behavior. These values indicate how different the runtime behavior is with respect to the expected behavior (observed in the testing phase).

As for the third point, it is taken into account in the invariants construction phase (i.e., in the testing phase). In that phase, Daikon allows setting a confidence level of the built invariants, that determines the robustness of invariants and can significantly impact the probability for a violation of being a false-positive. It computes, for each invariant, the probability that the considered property would appear by chance in a random input. If that probability is smaller than a user-specified confidence parameter, then the property is considered non-coincidental and is reported as invariant. It assumes a distribution and performs a statistical test where the *null hypothesis* states that the observed values were generated by chance from the distribution [23]. If the *null hypothesis* is rejected at a certain level of confidence, the observed values are non-coincidental and the corresponding property is reported as invariant. For instance, if the probability limit is set to 0.01, Daikon reports invariants that are no more than 1 percent likely to have occurred by chance. The lower is the confidence parameter, the more robust is the invariant and the smaller will be the percentage of false positives in the runtime phase.

We compute the risk factor as: RF_i = *#Violation/#MaxViolation* * *#DistinctPoints/#MonitoredPoints*, where *#MaxViolation* is the number of potential violations that may have occured in the monitored program points (that is the number of monitored parameters per each occurred interaction), *#DistinctPoints* is the number of distinct exchanged parameters that experienced a violation and *#MonitoredPoints* is the total number of distinct monitored parameters. The so-computed risk factors are used to penalize the reliability of components, at the step *n*, as follows:

$$R^{n}_{ONLINEi} = R^{n-1}_{ONLINEi} - R^{n-1}_{ONLINEi} * RF_i * W \quad (6)$$

where W is a parameter set by the user in order to establish “how much” impact the risk factor has on the reliability. This parameter is set empirically in the tuning phase of the monitoring system. To set this value, we strongly recommend to consider the confidence parameter that has been set for the invariant building phase. Indeed, the smaller is the value of the confidence parameter, the higher the value of W should be, because a violation to “robust” invariants are more serious. In our experiments, we set it to the confidence level adopted for Daikon, i.e., to 0.01.

Based on the new computed visit counts and reliability values, the overall system reliability R_{ONLINE} is computed (by eq.5), at regular intervals of time T. When it goes under the threshold (R_{EXP} - Thr) an alarm is triggered. The monitoring system then shows the differences between ideal values and the estimated ones (i.e., differences between component reliability values and between visit counts), from which an indication about the cause can be also deduced: if the difference relates to the reliability of a component, then the violations (and thus the involved methods and parameters) causing it is identified; if the difference relates to the visit count values, the cause is inferred from the interaction among involved components.

V. Experimentation

A. Application

We applied the proposed approach by monitoring an application for queuing systems simulation, based on *javasim*[4]. As known, queuing theory and queuing networks simulation have a large number of applications, ranging for performance and dependability analysis to resources allocation in telecommunication systems. The developed application performs job queues simulation according to several models, such as M/M/1 and M/M/n, and periodically reports graphical results to the user. The latter can, online, evaluate the attributes of interest (e.g., the block probability trend) and takes proper actions depending on the results. He can also make modifications to simulation parameters for each simulation run, thus allowing an iterative analysis until the desired tuning level is achieved. Results consist of graphical representations of several statistics, such as the response times for processed jobs, the average response time trend, the response time distribution, the steady state probability and the block probability trend.

The application supports two methods for the output analysis, namely the *independent replication* method, where the simulation is repeated *n* times, using a different random-number stream and with independent initial conditions for each run, and the *batch means* method, where a single, long simulation run is divided into contiguous segments (or batches), each having length *m* and treated as an individual observation. To graphically report results on the user terminal, the application uses the well-known *JFreeChart*[5] Java library. The block diagram is depicted in Figure 3. It is composed of three

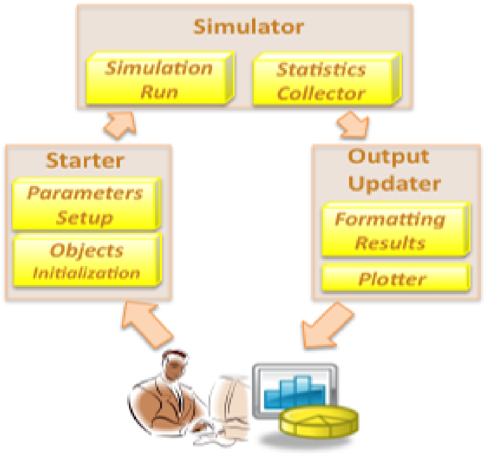

Fig. 3: Experimental Application.

basic blocks: the *Starter* block, which is responsible for initial instantiations (e.g., the queue, the simulation controller), and accepts simulation parameter settings; the *Simulator* block, which performs the actual simulation and collects statistics of interest, and the *Output Updater* block, which manages and formats the results, and plots them on the terminal. The total size of the case-study application amounts to 5426 Lines of Code (LoC), 37 classes and 236 methods (without considering the *JFreeChart code*, which would considerably increase these numbers). As first step of our approach (see Figure 1) we identified 12 components in this application,

[4]JavaSim is a simulation package available at: http://javasim.codehaus.org/

[5]JFreeChart is a free Java chart library to develop professional quality charts. It is available at: http://www.jfree.org/jfreechart/

corresponding to the exercised Java Packages (i.e., we chose packages as component granularity). They became the states of a DTMC. Obviously, not all the *JFreeChart* packages are used by the rest of the application; thus we considered only the exercised packages, i.e., those ones "visited" by the control flow execution. Similarly, the transition probabilities assignment makes sense only for these packages. Instrumented components are reported in the first column of Table I.

B. Experimental procedure

Testing Phase. According to the described approach, the application was instrumented during its system testing. We generated a test suite of 180 test cases, randomly picking valid combinations of input parameters (e.g., *interarrival time distribution, service time distribution, queue length, number of jobs, simulation method* (independent replication or batch means), *interarrival and service time means*). Execution traces were produced by the *Daikon* tool, that monitored the application in the methods entry/exit of the interfaces among components, and then built the invariants from the observed values. Data about *execution counts* (i.e., the number of times the execution flows from a component to another) were extracted from execution traces, and used for the *visit counts* computation as explained in section III-A.

Reliabilities of single components were computed by equation 4 and as described in [11]. Hence, at the end of the testing phase, additional 360 random executions were produced with the only goal of measuring reliabilities (i.e., if a failure was detected in these runs, the bug was not removed, as explained in section III-A). The advantage of this simple method (and the reason because we preferred it to the SRGM construction) is that during these executions, we were able to collect further data for *visit counts* computation and for invariants generation, which along with the previous data, improved the accuracy of these estimates. During the testing session, we fixed in total 11 bugs. In the additional 360 executions for reliabilities estimation (i.e., after the testing), just 1 application failure was experienced, in the *Format* component, leading to a final expected reliability for the application of $R_{EXP} = 0.9972$ (computed by eq. 5). It is clear that just using the model-based approach in the testing phase, where the system is tested for its intended use, usually leads to overestimation. The task of dynamic analysis during operational phase is to adjust it, by taking into account the real behavior that may significantly differ from the tested one.

Runtime Phase. The threshold value Thr, which determines the reliability R_{MIN} to be not overcome at runtime, is usually an application requirement. For these experiments, it was set to Thr = 0.0027, giving $R_{MIN} = R_{EXP} - Thr = 0.9945$. This means that when reliability estimate is lower than R_{MIN}, an alarm is triggered. The choice of the update interval T depends, as discussed below, on the desired trade-off between accuracy and overhead. For these experiments, it was set to T = 30 seconds. In the second phase, the application was run and observed by our monitor. Starting from the mentioned input parameters, we defined equivalence classes. By assuming a uniform distribution inside each class, we generated three distinct operational profiles, by assigning distinct occurrence probability values to each class. Each experiment consists of an execution randomly picked from the equivalence classes according to the defined operational profile distribution. Executions are generated from different profiles in order to average the effect due to three possible distinct usage of the application. A set of 30 executions per each operational profile was run. Thus, the monitoring system is evaluated over a total of 90 executions. Each execution performs a simulation that may terminate successfully or may fail, due to residual bugs in the code.

At each time interval T, the prototype monitor (i) traces the execution counts, and (ii) compares the current execution values with the built invariants, in order to detect violations in the monitored points. It produces a list of execution count values and a list of occurred violations. Violations are detected in an online manner by the *Daikon* tool *runtime-checker*. By using static information about the instrumentation (i.e., the *MonitoredPoints* value), and by reading the violation list to obtain the *MaxViolation*, *DistinctPoints* and the number of *Violations* value, risk factors for each component (and for each time interval) are computed, as explained in section IV (i.e., RF_i = *#Violation/#MaxViolation* * *#DistinctPoints/#MonitoredPoints*). *Visit counts* are instead obtained from the execution counts list. The weight W was set to 0.01, according to the confidence level assigned to Daikon invariants.

C. Results

For illustrative purpose, Table I shows the reliability values for one of the experiments (the experiment number 4). It reports the estimated values for visit counts (V_i), risk factors (RF_i), components reliability (R_i), and the total reliability (the last row) per each time interval. Components reliability at a given interval is obtained by equation 6 (hence using RF_i values), whereas the total reliability value at each time interval is computed by equation 5 (hence using V_i and R_i values). Reported values show how the total reliability progressively decreases, except from the fourth interval, where it slightly increases. This phenomenon occurred in many experiments, and it is due to changes in the types of performed operations that may occur during one experiment. For instance, the experiment may initially exercise some components, and after some point it may start exercising other different components, generating different visit count values and violations, which cause the reliability estimate to increase. In the reported case, it is possible to note a progressive change in the visit count values, which basically increase during the five intervals in those components responsible for formatting results and plots graphs (such as *org.jfree.chart.plot* or *org.jfree.chart.ChartFrame* or *Format*), whereas decrease in components dealing with initial simluation data and with the actual computation (such as *ObjectInitializer* or *Setup*, that are progressively less visited). In this experiment an alarm is triggered in the third interval, since the estimated reliability went under the minimum value

TABLE I: Component reliabilities and risk factors for the first three intervals in the experiment number 4. V_i values are visit counts to component i; RF_i values are risk factors, and R_i values are component reliabilities.

Interval T_i	**1**			**2**			**3**			**4**			**5**		
Components	V_i	RF_i	R_i	V_i	RF_i	R_i	V_i	RF_i	R_i	V_i	RF_i	R_i	V_i	RF_i	R_i
org.jfree.chart	1.46	0.0250	0.99975	1,42	0.0140	0.99961	1.49	0.0080	0.99953	1.39	0.0040	0.99949	1.37	0.0090	0.99940
org.jfree.data	0.64	0.0130	0.99987	0.66	0.0090	0.99978	0.85	0.0080	0.99970	0.86	0.0020	0.99968	0.37	0.0070	0.99961
org.jfree.chart. renderer.xy	0.66	0.0510	0.99949	0.58	0.0050	0.99944	0.67	0.0130	0.99931	0.71	0.0020	0.99929	0.85	0.0030	0.99926
org.jfree.chart.axis	0.54	0.0520	0.99948	0.51	0.0140	0.99934	0.56	0.0090	0.99925	0.58	0.0010	0.99924	1.02	0.0020	0.99922
org.jfree.chart.plot	0.32	0.0080	0.99992	0.34	0.0110	0.99981	0.37	0.0550	0.99926	0.52	0.0010	0.99925	1,21	0.0010	0.99924
org.jfree.chart. ChartFrame	0.41	0.0040	0.99996	0.49	0.0045	0.99991	0.49	0.0005	0.999915	0.78	0.0020	0.99989	1	0.0020	0.99987
ObjectInitializer	1.72	0.0210	0.99979	0.44	0.0300	0.99949	0.42	0.0350	0.99914	0.21	0.0060	0.99908	0.1	0.0050	0.99903
Setup	1.39	0.0200	0.99980	0.98	0.0250	0.99955	0.89	0.1371	0.99818	0.34	0.0120	0.99806	0.19	0.0261	0.99780
Simulator	1.79	0.0740	0.99926	1.82	0.0020	0.99924	1.91	0.0010	0.99923	1.88	0.0020	0.99921	0.86	0.0020	0.99919
Collector	0.78	0.0250	0.99975	0.52	0.0440	0.99931	0.49	0.0180	0.99913	0.78	0.0030	0.9991	1.12	0.0020	0.99908
Format	0.75	0.1180	0.99882	0.29	0.0200	0.99862	0.48	0.0140	0.99848	0.72	0.0040	0.99844	1.05	0.0260	0.99818
Plotter	0.78	0.0120	0.99988	0.71	0.0050	0.99983	0.69	0.0010	0.99983	0.81	0.0040	0.99978	1.21	0.0060	0.99972
Total Reliability			**0.9958**			**0.9956**			**0.9931**			**0.9933**			**0.9921**

TABLE II: Results referring to experiments that caused an alarm triggering

Test Case #	**Lowest Estimated Reliability**	**False Alarms**
3	0.9939	Yes
4	0.9921	**No**
9	0.9941	**No**
19	0.9943	Yes
20	0.9936	Yes
21	0.9927	**No**
32	0.9944	Yes
36	0.9936	Yes
41	0.9939	Yes
52	0.9942	Yes
66	0.9932	**No**
71	0.9944	Yes
73	0.9939	Yes

that we set to $R_{MIN} = 0.9945$.

Table II reports the list of all the executions in which an alarm has been triggered by the monitor. Per each execution, the lowest reliability value estimated by the tool is reported (column 2) and a label indicating if the alarm triggering(s) turned out to be a *false-positive* (i.e., the execution terminated successfully) or not (i.e., the application actually failed). 28 out of 90 executions reported violations with respect to the built invariants. In 13 cases (the ones reported in Table II), the estimated reliability went under the threshold, causing the monitoring system to trigger an alarm. The low values were caused by violations detected for various components; however, this did not always indicate a failure-causing fault. As may be seen, just in 4 of these cases the system actually failed; 9 cases were tagged as *false-positive*, since, although alarm triggering, the application completed the task without failing. In these experiments, other than the mentioned 4 failures, the application failed in one more experiment, but without any alarm triggering. This means that in 4 cases the predicted failures could be avoided, by applying proper proactive actions, but in one case the monitor failed to trigger an alarm (i.e., a *false-negative* occurred).

Table III reports synthetic results of the experimental campaign. For each operational profile: column 1 reports the

TABLE III: Results per Operational Profile

Operational Profile	**Failures**	**False-negatives**	**False-positives**
Profile 1	3	0	3 of 6
Profile 2	1	0	2 of 3
Profile 3	1	1	4 of 4

number of experienced failures (5 in total), column 2 reports the number of *false-negatives* (i.e., the number of times a failure occurred without any alarm triggering); column 3 reports the number of *false positives* (i.e., the number of times an alarm is triggered but the application did not fail, that is 9 of 13 alarm generation in total). It is finally interesting to observe the online reliability estimate in the five cases when the system failed. Figures 4, 5 and 6 show the reliability estimation trends, for several intervals of observations until the failure occurrence, and per each operational profile.

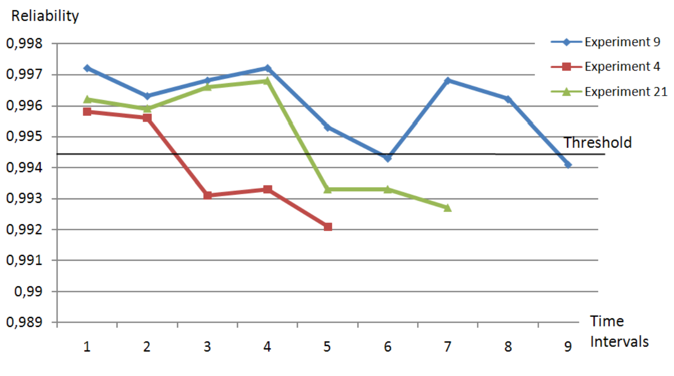

Fig. 4: Online estimated reliability from operational profile 1

In the first graph (i.e., executions generated from the first operational profile), the application failed three times. In the second and third graph, the application failed once. One of these executions (the number 83) caused the mentioned *false-negative*. In the experiment 4, 21 and 66, the alarm is triggered in the time intervals immediately before the actual failure, while in the experiment number 9, the alarm is triggered three

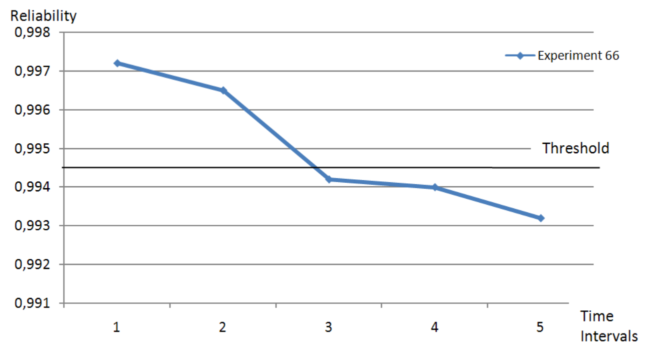

Fig. 5: Online estimated reliability from operational profile 2

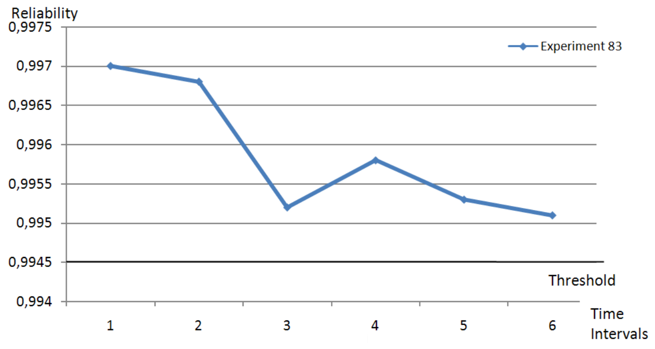

Fig. 6: Online estimated reliability from operational profile 3

time intervals before the actual failures, but the estimated reliability goes over R_{MIN} in the two time intervals before the final failure. A prudent choice is to apply proactive actions as soon as an alarm is triggered (this choice, of course, depends on user needs), as for instance carrying out a hot component replacement, a process migration, a checkpointing, a reconfiguration or a suspend operation. Another point to note is that in four out of five experiments, the reliability values have been noticed to increase after some points, for the reasons already explained (i.e., changes in visits to the involved components).

Summarizing, in four of the represented cases the significant number of violations caused reliability to be estimated under R_{MIN}, correctly triggering an alarm, since the application failed in the subsequent intervals. In all the other cases, either violations were not sufficient to trigger an alarm or they caused a false alarm. In the experiment number 83, the low number of violations and the nearly unchanged visit counts caused the reliability estimate to remain always over R_{MIN} before the application failure, causing a *false-negative*. The *false-negatives* can be more or less dangerous depending on the application. In some cases, a conservative choice could be made, by setting the threshold Thr to a lower value. For instance, setting Thr to 0.0017 would have caused more false-positives, but no false-negative. The choice of Thr depends on the specific application and user requirements (e.g., for a critical system, it may be desirable to have no false-negatives, hence, Thr will by very low).

VI. Overhead

The proposed approach introduces overhead both in the testing phase and in the runtime phase. The main part of of the overhead is due to the invariants computation (that includes the execution trace production and the actual invariants inference), carried out in the testing phase by the tool that we adopted, i.e., *Daikon*. In [23] *Daikon*'s authors analyze this cost as a function of three factors: i) it is linear in the number of potential invariants at a program point. Actually, since most invariants are soon discarded, time is linear in the number of *true* invariants, which is a small constant in practice. ii) It is linear in the number of times a program point is executed (i.e., linear to test suite size), and iii) linear in the number of instrumented program points (that is proportional to the size of the program). In order to reduce the overhead, we did not instrument each method, but only the methods of components interface. It could be further reduced to few critical points if the total overhead is judged to be too high. In our case, *Daikon* took 2283” (i.e., about 38’) , to build such invariants from execution trace file produced by the 180 test cases. It is worth to point out that this cost is, in general, highly variable depending on the tool configuration and on the actual values taken by the instrumented variables during executions, as stated by *Daikon*'s authors. The cost of the testing phase, even if accounting for the greatest part of the overhead, is an “offline” cost, that has to be incurred once for all.

A further contribution has to be considered in the runtime phase. Before the execution, the application is instrumented with the tool *runtime-checker* of the *Daikon* tool suite in order to detect violations in an online manner (which are appended to a list, but without any writing to files). The instrumented application has been experienced to run slightly slower than the non-instrumented one (in the average, it run 1.04 slower). In addition to this overhead, the time needed to compute equation 5 has to be considered. It includes the time to read the execution counts list and violations list (not from files) and to compute risk factors and visit counts value. This time is negligible compared with the other contributions (it was never greater than 0.26 seconds). The runtime overhead and its predictability may be very important in real time applications, because they determine the responsiveness of the system. In our experiments, it turned out to be reasonably low. This time is tightly related to the number of invariants to check, and such a dependence may be important for predictability: the number of invariants is known in advance and hence it can be used for predicting purpose.

It is finally worth to point out that these results are limited to our case study; however, as suggested in [27] several actions can be taken to reduce instrumentation and invariants computation cost. The most suitable solutions are (i) reducing the number of instrumented points, (ii) reducing the number of executions, (iii) reducing the number of variables. One more discussion point is about the choice of T. Longer intervals reduce the overhead contribution due to the computation of reliability, but also implies less frequent evaluations, causing greater risks. T has to be regulated based on the desired trade-off between accuracy and overhead. For instance, consider the case of experiment 9, with T = 45" and T = 1 minute, respectively. With T = 45", the estimation at the 6th interval (i.e, after 4' 30") would correspond to the estimation at the 9th interval with T = 30". The estimated value would probably be lower than the value estimated with T = 30", since more violations would be taken into account with T = 45", in the time between 3' 45" and 4' 30" (with T = 30, the interval is between 4' to 4' 30"). More violations means a higher risk factor and thus a lower estimate. Hence, in this case nothing would change. However, in the second case, with T = 1 minute, the last reliability evaluation would correspond to the 8th interval of the case with T = 30", and would account for violations between minute 3 and 4. This case (i.e., the estimate at the 8th interval) was estimated to be over R_{MIN}; with T = 1 minute, the total number of violations between the minute 3 and 4 is higher than with T = 30" (and, as a consequence, lower reliability), but the total reliability could not decrease under R_{MIN}, not causing any alarm triggering. Since the application failed before the reliability evaluation at the 5th minute (it failed between minute 4.5 and 5), *no alarm would be triggered in this case*. This suggests that T cannot be too high, since less frequent evaluations can cause higher risk.

VII. Conclusion and Future Work

We presented an online reliability monitoring approach that takes advantage of static modeling and dynamic analysis to give continuous estimation of the system reliability. A prototype implementation was experimented and preliminary results show the benefits brought by the combination of modeling and operational data usage. Experiments also highlighted the issues that need to be addressed in the future. In particular, we need to explore new solutions to reduce the number of false positives, hence to improve the accuracy, and to provide the system with the ability to automatically learn the violations that did not result in a failure, in order to differently evaluate them when they re-appear. Moreover, we plan to explore other architecture-based modeling approaches, such as Stochastic Petri Nets (SPN), to also consider concurrent systems. Finally, the effectiveness of the monitoring system would improve if the choice of the threshold value were done adaptively. The monitoring system should learn by itself and then adapt the threshold value based on the acquired experience. We aim to do this in the future, by combining the proposed approach with other online diagnosis mechanisms (e.g. [28]).

Acknowledgment

This work has been partially supported by the project "CRITICAL Software Technology for an Evolutionary Partnership" (CRITICAL-STEP, http://www.critical-step.eu), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, within the context of the Seventh Framework Programme (FP7).

References

[1] Y. Dai, Y. Pan, X. Zou, A Hierarchical Modeling and Analysis for Grid Service Reliability, *IEEE Trans. on Computers*, vol. 56, 681-691, 2007.

[2] M. Rabah and K. Kanoun, Performability evaluation of multipurpose multiprocessor systems: the "separation of concerns" approach, *IEEE Trans. on Computers*, vol. 52, 223-236, 2003.

[3] Trivedi, K. Wang, D. Hunt, D.J. Rindos, A. Smith, W.E. Vashaw, B., Availability Modeling of SIP Protocol on IBM©WebSphere ©, *Proc. of the 14th IEEE Pacific Rim Intl. Symposium on Dependable Computing*, 2008, 323-330.

[4] W. E. Smith, K. S. Trivedi, L. A. Tomek, J. Ackaret, Availability analysis of blade server systems, *Ibm Systems Journal*, vol. 47, no. 4, 2008.

[5] G. Ciardo and K. S. Trivedi, Decomposition Approach to Stochastic Reward Net Models, *Performance Evaluation*, vol. 18, 37-59, 1993.

[6] J. B. Dugan, Automated Analysis of Phase-Mission Reliability, *IEEE Transaction on Reliability*, vol. 40, 45-52, 1991.

[7] D. Daly, W. H. Sanders, A connection formalism for the solution of large and stiff models, *34th Annual Simulation Symposium*, 2001, 258-265.

[8] I. Mura and A. Bondavalli, Markov Regenerative Stochastic Petri Nets to Model and Evaluate the Dependability of Phased Missions, *IEEE Transactions on Computers*, vol. 50, 1337-1351, 2001.

[9] J. P. Ganesh, and J. B. Dugan: Automatic Synthesis of Dynamic Fault Trees from UML System Models, *Proc. of the IEEE Int. Symposium on Software Reliability Engineering*, (ISSRE), 243-256, 2002.

[10] 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, 391-412, 2004.

[11] K. Goseva-Popstojanova, A.P. Mathur, K.S. Trivedi, Comparison of architecture-based software reliability models, *Proc. of the IEEE Intl. Symposium on Software Reliability Engineering*, 22- 31, 2001.

[12] S. Gokhale, M.R. Lyu, K.S. Trivedi, Reliability simulation of component-based software systems, *Proc. of the IEEE Intl. Symposium on Software Reliability Engineering* (ISSRE '98), pp. 192-201, 1998.

[13] W.Wang, Y.Wu, M.H. Chen, An architecture-based software reliability model, *Proc. of the Pacific Rim Dependability Symposium*, 1999

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

[15] Garzia, M.R., Assessing the Reliability of Windows Servers, *Proc. of IEEE Dependable Systems and Networks*, (DSN-2002).

[16] D. Tang, R.K. Iyer, Dependability Measurement and Modeling of a Multicomputer System, *IEEE Trans. on Computers*, 42(1), 62-75, 1993

[17] D.Long, A.Muir, R.Golding, A Longitudinal Survey of Internet Host Reliability, *Proc. of the 14th Symp. on Reliable Distributed Systems.*

[18] Kesari Mishra, K.S. Trivedi, Model Based Approach for Autonomic Availability Management, *Proc. of the Intl. Service Availability Symposium, Helsinki* , Finlande, 2006 , vol. 4328, 1-16

[19] Haberkorn, M. Trivedi, K., Availability Monitor for a Software Based System, *Proc. of the 10th IEEE High Assurance Systems Engineering Symposium*, 2007. HASE '07, 21-328

[20] V. Dallmeier, C. Lindig, A. Wasylkowski, A. Zeller, Mining Object Behavior with ADABU, *Proc. of the 2006 Intl. workshop on Dynamic systems analysis, Intl. Conference on Software Engineering*, 17 - 24.

[21] B. Schmerl, D. Garlan, H. Yan, Dinamically Discovering Architectures with DiscoTect, *Proc. of the joint meetings of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005, 103 - 106

[22] Sudheendra Hangal, Monica S Lam, Tracking Down Software Bugs Using Automatic Anomaly Detection, *Proc. of the 24th IEEE Intl. Conference on Software Engineering*, 2002. ICSE 2002. pp. 291- 301

[23] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, *IEEE Transactions on Software Engineering*, vol. 27, 2001, 99-123.

[24] Trivedi, K.S. "Probability and Statistics with Reliability, Queuing and Computer Science Applications," John Wiley and Sons, 2001.

[25] 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. 493-509, April 2007.

[26] A. Pasquini, A. N. Crespo, P. Matrella, Sensitivity of reliability growth models to operational profile errors vs testing accuracy, *IEEE Transaction on Reliability*, vol. 45, 531-540, 1996.

[27] The Daikon Invariant Detector, http://groups.csail.mit.edu/pag/daikon/

[28] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and F. Grandoni, Threshold-based mechanisms to discriminate transient from intermittent faults, *IEEE Transactions on Computers*, 49(3), pp. 230-245, 2000.