Online Reliability Monitoring: a Hybrid Approach

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*—Assuring high reliability levels in complex software systems is difficult. The spread of component-based paradigm brought, along with many advantages, new thorny problems and challenges. Various approaches have been proposed to guarantee high reliability and cope with such problems– among these, proactive policies are particularly effective and inexpensive. The ability to monitor the system at runtime and to give online estimations about the trend of dependability attribute of interest, is the key to implement strategies aiming at forecasting, and thus proactively preventing, the system failure occurrence. In this paper, an online reliability monitoring approach is proposed. It combines benefits of architecture-based reliability model and dynamic analysis, so as to integrate static modeling power with representative operational data. Its usage is illustrated by a prototype implementation, a case-study and preliminary results.**

I. Introduction

Software reliability assessment and assurance have become crucial concerns to enable systems and infrastructures to be employed in critical contexts. Although significant improvements have been achieved over time, the increasing software complexity, the heterogeneity of components, and the usage of OTS (off-the-shelf) items, including Operating Systems (OSs), third-party libraries and virtual machines, pose tricky issues that traditional methods need to cope with. Reactive policies, in which actions are taken in reponse to errors or component failures to prevent them from causing the system failure, have been widely adopted in many contexts. However, the described complexity and the need for balancing dependability requirements with costs are making their application difficult.

In recent years, another point of view is gaining importance: proactively acting before the system fails by attempting to forecast failure occurrence. The goal is to avoid the system failure (rather than tolerate it) or, at least, to make the system fail safely, by observing the current system behaviour and taking proper preventive actions when the system is suspected to soon becoming unsafe. Some of these actions include: checkpointing, process migration, and software rejuvenation.

A crucial role in proactive failure prevention is that of the system monitor and the subsequent dynamic evaluation of dependability attributes of interest during operation. Runtime dependability evaluation is essential to enable proper proactive actions to be designed, implemented and promptly triggered. Dependability attributes are often estimated, statically, in the development phase by using modeling techniques that are assumed to be representative also in the runtime phase. The usual approach is to develop a stochastic model of the system and solve it analytically or via discrete-event simulation to predict the attributes 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 estimation may not be accurate, due to the necessary simplifying assumptions (about components and their interactions) 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 to design proactive actions that have to prevent runtime system failures (e.g., unneeded actions are applied or, even worse, needed actions are not applied). On the other hand, just using operational data, without a model that is able to give preliminary estimates, is not possible if we want to rely on proactive policies (in fact, we should wait for collecting enough failures data to get a satisfactory confidence, and proactive actions would be not possible before these data are available).

In this paper, 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. 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 estimation when software is released (i.e., after testing) and then (ii) to continuously refine 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 experimentally evaluated on a case-study consisting of an application in the field of queueing systems simluation.

II. Backgroun and Related Work

A. Reliability Evaluation

Reliability can be evaluated by using several approaches, generally classified into two categories: model-based and

measurements-based approaches. 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 modelling approaches have appeared in the literature:

1) compositional approaches (e.g., [1], [2], [3], [4], [5]), where the system model is constructed in a bottom-up fashion. The models representing parts of the systems are built in isolation, and suitable composition operators and composition rules are defined;
2) decomposition/aggregation approaches (e.g., [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. The advent of object-oriented and component-based systems paved the ground for this kind of models, and they have increasingly been adopted for performance and reliability evaluation [10], [11], [12]. The software architecture is usually extracted from design, source code or even object code, and the level of decomposition (i.e., component granularity) can be defined depending on the needs ("components" are intended as logically independent unit performing a well-defined function [11]). Interactions among components are modelled by transition probabilities (i.e., the probabilities that the control flows among components), also extracted from design or operational profile estimations. Depending on the way the architecture is combined with the failure behaviour, they are categorized as [13]:

- 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, modelling 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 modelled by non-homogeneous Poisson process (NHPP) and the system failure intensity is estimated as the sum of the individual components failure intensities.

Models, in general, are very useful for their ability to abstract from unnecessary details, and allow to suitably analyze the architecture, to evaluate different configurations, to pinpoint performance/reliability bottlenecks, and to compare design alternatives without physical implementation.
However, they may be not accurate enough, when the input parameters values are not representative of the real system behavior. 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 obtain the detailed characterization of the system behaviour in presence of faults. However, since real data are needed, it is not always possible to apply this approach, because data may be not available. 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 [14]. An overview of experimental approaches to dependability evaluation is in [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 towards system availability evaluation is in [18], [19]. The approach proposed in this paper combines both kinds of evaluation methods in order to implement a monitoring system for autonomic reliability management.

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. There are several dynamic analysis tools (e.g., [20], [21], [22]).
For our purpose, we rely on one of the most successful inferential engines, that is Daikon [22]. Daikon allows us to infer the likely I/O invariants of a program execution (i.e., invariants on exchanged paramters), by using more than 160 invariants templates. It provides useful information on the relation between the values of the variables at different execution points. It starts with a set of syntactic constraints for the considered 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: (i) instrumentation, execution and monitoring of the application; (ii) recording of the I/O (Input/Output) behaviors; (iii) 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 indentifies invariants in specific points of the program; we are interested in using it for deriving constraints on exchanged parameter values in the I/O flow of each component.

III. The Runtime Monitoring System

What we propose is a monitoring system that triggers alarms when the online estimated reliability R_{ONLINE} is lower than the expected reliability R_{EXP}, that is estimated at the end of testing phase, for a given threshold quantity Q. This means that the probability that the system fails 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. 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 that 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 components.

A. Modelling phase

Suppose we have a system that has to undergo a testing phase, and that we want to give reliability estimation by using its DTMC model representation. During the testing, data about components failures are collected in order to estimate their reliability. Components reliability can be estimated either by building a software reliability growth model (SRGM) that use interfailure times to fit a failure intensity model and by taking its values at the end of the testing phase, or by using the following formula:

$$R_i \approx 1 - \lim_{n_i \to \infty} \frac{f_i}{n_i} \tag{1}$$

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 test cases [11]. By using the DTMC model, the system reliability is computed as described in [10], i.e.:

$$E[R] \approx \prod_{i}^{n} E[R_i^{X_{1,i}}] = (\prod_{i}^{n-1} R_i^{E[X_{1,i}]}) R_n \tag{2}$$

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*), also known as Visit Counts (V_i). Second-order architectural effects can be considered as in [23]. By observing the control flow among components, execution counts, and then the expected number of visits to each component during an execution (i.e., visit counts), are computed and used in the model. An example on how to compute the visit counts is in [10]. Notice that this model will be used also in the runtime phase: if we use SRGMs to estimate components reliability, we will have to take the failure intensity function values at the end of each period of observation (since, in general, the failure intensity after software release is not constant), whereas the second type of estimation (eq.1) is more simple but less accurate. What affects the accuracy of the theoretical estimation given in the eq. 2, are the assumptions on which the adopted model is based:

- 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. [24] show that the impact of the operational profile estimation error is not high).

B. Runtime Phase

To overcome these limitations, a runtime refinement phase is carried out on the base of the following observations: the error between the reliability estimated after testing by the model and the actual reliability may be due (as a consequence of the made assumptions) 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 order.

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; after some time, we can give a reliable estimation (i.e., with a given confidence level) of random variables V_i, and compute the reliability by using these new values. Despite this estimation, reliability values of the single components 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 a dynamic analysis tool (we used Daikon) that describes the behavior of components by monitoring their interactions and by building invariants on exchanged values, (ii) and then to detect deviations from the defined expected behavior.

In particular, in the testing phase, the system is instrumented with this tool that "observes" the interactions in order to build the expected, and thus supposedly "correct", behavioral model. It is an "estimation" of the correct behavior, because the testing, of course, 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 an estimation R_{EXPi}, carried out for the component *i* 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 overall process is depicted in figure 1. The same architectural model is used in the testing and in the operational phase. In the operational phase the monitoring tool uses real collected data to statistically estimate the visit counts and component reliabilities; 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 about the cause of possible deviating behaviors, as mentioned in the next section.

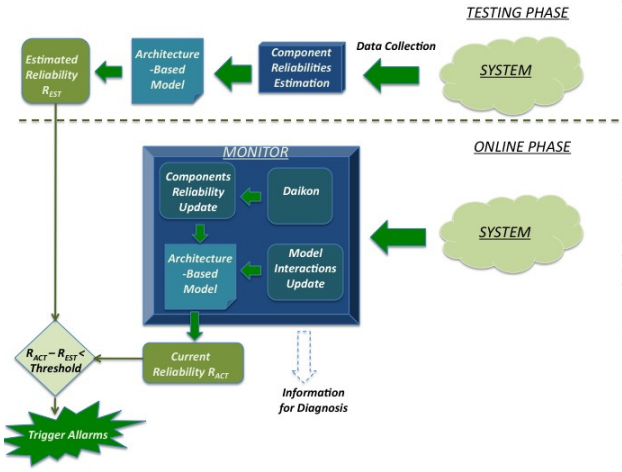

Fig. 1: Monitoring System.

IV. Estimating the Reliability Degradation

Component reliabilities need to be reduced to take into account new online behaviors they exhibit. 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. It aims to estimate the risk associated with the set of all violations occurring in the instrumented program points during the 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). Its value depends (i) on how many violations occurred for each monitored parameter, (ii) on how many distinct program points 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. We computed the risk factor as: RF_i = *#Violation/#MaxViolation* * *#DistinctPoints/#MonitoredPoints*, where *#MaxViolation* is the number of interactions occurred in the monitored program points, *#DistinctPoints* is the number of distinct parameters that experienced a violation and *#MonitoredPoints* is the number of distinct monitored parameters.

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. 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 [22]. 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 so-computed risk factors are used to penalize the reliability of components, at the step *n*, as follows:

$$R^n_{ONLINEi} = R^{n-1}_{ONLINEi} - R^{n-1}_{ONLINEi} * RF_i * W \quad (3)$$

where W is a parameter set by the user in order to establish "how much" the risk factor has to impact on the reliability penalization. This parameter has to be 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. The higher is the confidence parameter the higher the value of W should be, because a violation to "robust" invariants are more serious. Based on the new computed visit counts and reliability values, the overall system reliability R_{ONLINE} is computed (by eq.2), at regular intervals of time T. When it goes under the threshold (R_{EXP} - Q) an alarm is triggered. The monitoring system then shows the differences between ideal values and the estimated ones, from which an indication about the cause can be deduced: if the difference is in the reliability of a component, then the violations (and thus the involved methods and parameters) causing it are identified; if the difference is in the visit counts values, the cause is inferred from the interaction among involved components.

V. Experimentation

A. Application

We experimented the proposed approach by monitoring a simple application for queuing systems simulation, based on *javaSim*[1]. As known, queuing theory and queuing networks simulation have a large number of applications, ranging for performance and dependability analysis to resource allocation in telecommunication systems. The developed

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

application performs job queues simulations according to several models, such as M/M/1 and M/M/n. The application periodically reports graphical results to the user, which, online, can evaluate the attributes of interest (e.g., the blocking probability trend) and takes proper actions according to the results. The user can also make modifications to some parameters for the successive simulation runs. 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 blocking probability trend. The application uses the well-known *JFreeChart*[2] Java library to plot charts on the user terminal. The block diagram is depicted in figure 2. It is composed of two basic blocks:

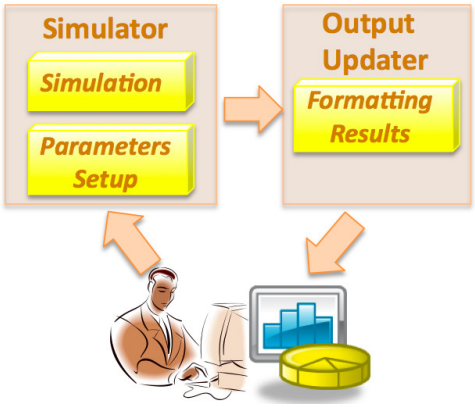

Fig. 2: Experimental Application.

the *Simulation* block, which accepts user settings and performs the simulation, and the *Output Updater* block, which manages the results and plots them on the terminal. We instrumented, for illustrative purpose, the *Output Updater* block and evaluated its reliability at runtime. It is mainly made by *JFreeChart* code. We considered as components the Java packages, that became the states of a DTMC. Not all the *JFreeChart* packages are used by the rest of the application, thus the transition probabilities assignment makes sense only for "visited" packages, i.e., those packages exercised by the control flow execution. Specifically, exercised packages are reported in table I.

B. Experimental procedure and results

Testing Phase. According to figure 1, the application was instrumented during its testing. Data about execution counts (i.e., the number of times the execution flows from a component to another) for the *visit counts* computation and data for the invariants computation were collected and extracted from execution traces. Execution traces and invariants were produced by the *Daikon* tool, that monitored the application in some specific program points. Reliabilities of single components (i.e., the *JFreeChart* packages) were computed by the equation 1 and as described in [11]. However, during the tests no failure were observed due to the monitored

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

TABLE I: Components reliabilities for the firsts three intervals

Interval T_i	1	2	3
org.jfree.chart	0.99953	0.99975	0.99961
org.jfree.data	0.99978	0.99987	0.99970
org.jfree.chart.renderer.xy	0.99949	0.99944	0.99931
org.jfree.chart.axis	0.99948	0.99934	0.99925
org.jfree.chart.plot	0.99926	0.99981	0.99992
org.jfree.chart.ChartFrame	0.999915	0.99996	0.99991

subsystem, resulting in a component reliability estimates equals to 1. This is clearly an overestimation, caused by the assumption that the testing operational profile will be the same as the runtime operational profile. It will be adjusted at runtime. With the obtained values for both reliabilities and visit counts, the overall reliability was computed by the equation 2, giving $R_{EXP} = 1$.

Runtime Phase. The value for the threshold was set to Q = 0.0055 and the update interval was set to T = 30 seconds. In the second phase, the application was run and re-instrumented by Daikon. Faults were injected in the code to evaluate the defined mechanism. In particular, we injected trivial faults aiming at causing little variations in the monitored parameters values (randomly) and at observing the behavior of the monitoring system. At each time interval T, the prototype monitoring system compared, by using the tool *Invariant Diff* of *Daikon*, the built invariants with the current trace file, in order to detect violations in the monitored points. Violations were then used to compute the risk factors. In fact, by analyzing the trace file and the violations file, the number of violations, the *MaxViolation*, *DistinctPoints* and *MonitoredPoints* values were derived, and risk factors for each component were computed (for each time interval). The weight W was set to 0.01, according to the confidence level assigned to Daikon invariants. Table I shows the computed reliability values (just for some time intervals), according to eq. 3. Figure 3 shows the final results, by reporting the values for the overall reliability of the system, for several intervals of observations. As may be

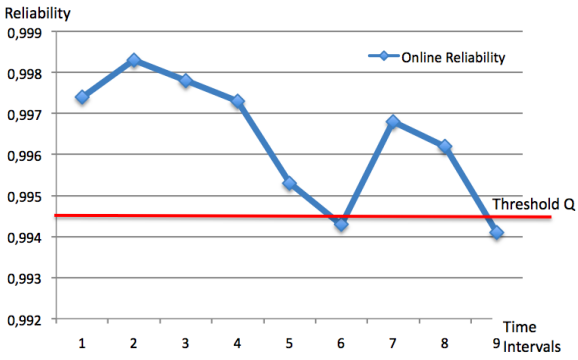

Fig. 3: Online Estimated Reliability

noted, in two cases the online reliability value was estimated to be under the threshold. In one of these case, a false-alarm was triggered. The low value was caused by the high number of violations detected for various components (especially for

org.jfree.data.xy and *org.jfree.chart.plot*) that, however, did not indicate any failure-causing fault, but only an anomalous behavior. After this point, the reliability value increased; this happened because of a change in the performed operations, that caused a different operational profile to be executed (and thus different components were exercised, with different visit counts and violations). In the last case, before the application failure, the significant number of detected violations caused the overall reliability to be estimated under the threshold, correctly raising an alarm activation, since the application failed in the subsequent interval.

Notice that a key role for avoiding false-positives and false-negatives is played by the threshold value. The higher is the threshold, the lower is the number of false-negatives (but the higher the number of false-positives). The threshold has to be defined according to the user needs, the application requirements and based on the experience. A dynamic adaptive threshold could be set, in order to take into account the experience. Finally, the choice of T is also important, since longer intervals cause minor overhead, but also cause the reliability trend to be evaluated more rarely, with greater risks. This also depends on application requirements and on the desired trade-off between the accuracy and the overhead.

VI. Conclusion and Future Work

We presented an online reliability monitoring approach that takes advantages of static modelling and dynamic analysis to give continuous estimations of the system reliability. A prototype implementation is been experimented with and results have shown the benefits brought by the combination of modelling and operational data usage. Experiments have also highlighted the issues that need to be addressed in the future. In particular, we aim 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, 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 obtain this in the future, by combining the proposed approach with other online diagnosis mechanisms (such as [25]).

References

[1] C. B. Almeida and K. Kanoun, Construction and Stepwise Refinement of Dependability Models, *Performance Evaluation*, vol. 56, 277-306, 2004.

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

[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] G. A. Hoffmann, K. S. Trivedi , M. Malek, A Best Practice Guide to Resource Forecasting for the Apache Webserver, *Proc. of the 12th IEEE Pacific Rim Intl. Symposium on Dependable Computing*, 2006,183-193.

[5] 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.

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

[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 Intl. Symposium on Software Reliability Engineering* (ISSRE '01), 22- 31, 2001.

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

[13] 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.

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

[15] Silva, G.J., Madeira, H., Experimental dependability evaluation. In Diab, H.B., Zomaya, A.Y., *eds.: Dependable Computing Systems: Paradigms, Performance Issues, and Applications*. Wiley (2005) 319-347.

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

[22] 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.

[23] 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.

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

[25] 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.