7-2013

Combining model checking and testing with an application to reliability prediction and distribution

Lin GUI

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Yuanjie SI

Jin Song DONG

See next page for additional authors

Citation

GUI, Lin; SUN, Jun; LIU, Yang; SI, Yuanjie; DONG, Jin Song; and WANG, Xinyu. Combining model checking and testing with an application to reliability prediction and distribution. (2013). *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15–20*. 101-111.

Available at: https://ink.library.smu.edu.sg/sis_research/5004

Combining Model Checking and Testing with an Application to Reliability Prediction and Distribution

Lin Gui
National University of Singapore
lin.gui@nus.edu.sg

Jun Sun
Singapore University of Technology and Design
sunjun@sutd.edu.sg

Yang Liu
Nanyang Technological University, Singapore
yangliu@ntu.edu.sg

Yuan Jie Si
Zhejiang University, China
siyuanjie@zju.edu.cn

Jin Song Dong
National U. of Singapore
dcsdjs@nus.edu.sg

Xin Yu Wang
Zhejiang University, China
wangxinyu@zju.edu.cn

ABSTRACT

Testing provides a probabilistic assurance of system correctness. In general, testing relies on the assumptions that the system under test is deterministic so that test cases can be sampled. However, a challenge arises when a system under test behaves non-deterministicly in a dynamic operating environment because it will be unknown how to sample test cases.

In this work, we propose a method combining hypothesis testing and probabilistic model checking so as to provide the "assurance" and quantify the error bounds. The idea is to apply hypothesis testing to deterministic system components and use probabilistic model checking techniques to lift the results through non-determinism. Furthermore, if a requirement on the level of "assurance" is given, we apply probabilistic model checking techniques to push down the requirement through non-determinism to individual components so that they can be verified using hypothesis testing. We motivate and demonstrate our method through an application of system reliability prediction and distribution. Our approach has been realized in a toolkit named RaPiD, which has been applied to investigate two real-world systems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking , Statistical methods, Reliability*

General Terms

Reliability, Verification

Keywords

MDP, hypothesis testing, reliability prediction, reliability distribution

ISSTA'13, July 15–20, 2013, Lugano, Switzerland
ACM 978-1-4503-2159-4/13/07
http://dx.doi.org/10.1145/2483760.2483779

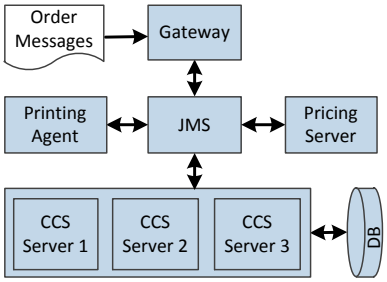

Figure 1: Architecture of the CCS System

1. INTRODUCTION

Testing is useful because it provides a certain level of assurance of system correctness/reliability. The more testing, the more likely a system behavior (being a bug or not) is demonstrated. When the system under test is deterministic, the level of "assurance" can be precisely captured through hypothesis testing, which is a statistical process determining whether to reject a null hypothesis based on tests generated according to the probability distribution in a model [3]. However, the testing method for quantifying the level of "assurance" remains unknown if the system is non-deterministic (or equivalently that the probability distributions of certain events are unknown or hard to predict). In this work, a probabilistic "assurance" for non-deterministic system is achieved through combing hypothesis testing and probabilistic model checking, and the underlying principle are demonstrated through an application of system reliability analysis.

A motivating example One product of our industrial collaborator, a financial software solution provider, is the Call Cross System (CCS), which is a stock trading system accepting order flow in a global operating environment. It operates on a 24 hours basis for 6 days per week. It has been successful in operation since 2005 and playing a crucial part in the core business of a financial institute in Boston. The CCS system is required to be highly reliable, and quantitatively 99.99% of the transactions must be correctly handled. With such a requirement, an immediate question is: how do we calculate the system reliability and present our calculation as a formal evidence to show that the delivered system will meet the requirement? A related question is: given the system level reliability requirement, what is the reliability requirement on each of the system components, so that the component development teams can carry out reliability

measures on their own? The former is known as reliability prediction problem and we term the latter as reliability distribution problem.

In order to answer the questions, we must understand the architecture of the CCS. Figure 1 shows the high-level architecture of the CCS, where arrows represent the directions of dataflow. At the top level, the system consists of six components. *Gateway* serves as a linkage between peripheral applications and the CCS. It receives order messages in the data batch manner and dispatches them to different symbol partitions in the CCS Servers via Java Message Service (JMS) server. *JMS* serves as the messaging engine for order flows, executions, pricing requests and responses, printing trade reports in several markets. *CCS Servers* are the core of the system to perform the business logics.They consist of a cluster of Websphere AppServer nodes with WebSphere Partitioning Facility (WPF) enabled. The workload is dynamically distributed to the server nodes based on the partitioning policies. The status of each partition is monitored in real-time. If one node fails, partitions in the failed node are reloaded to other healthy nodes. The partitions communicate with the external objects via a JMS server. *Pricing Server* is a JMS client that processes pricing requests and responds with pricing events to CCS servers. *Printing Agent* is a JMS client that receives printing requests and responses to JMS after printing. *DB* is the database server for the CCS system. A stock trading transaction is accomplished through a series of steps involving multiple system components. First, the Gateway sends order messages to its inbound queue through the JMS. The CCS servers receive order messages from inbound queue, process and store them into the database through underlying service framework. Afterwards, the Pricing Server provides the current price to CCS Servers. After the transactions are finished, the trading information is shown by the Gateway or printed by the Printing Agent. The transaction completes after displaying out. Furthermore, components are often duplicated in the CCS system to achieve high reliability.

Why existing approaches are not enough? Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment [22]. In this work, we consider reliability of a system based on the probability of failure of the system. Existing approaches on the reliability prediction problem fall into two categories: black-box approaches [18, 38] and white-box approaches [7]. The black-box approaches treat a system as a monolith and evaluate its reliability using testing techniques. They use the observed failure information to predict the reliability of software based on several models such as Jelinski-Moranda model [19], Musa Okumoto model [30], Littlewood-Verrall model [25], etc. On the contrary, the white-box approaches assume reliability of system components are known and evaluate software reliability analytically based on a model of the system architecture including Discrete Time Markov Chains (DTMCs) [7], Continuous Time Markov Chains (CTMCs) [22], or Semi-Markov Processes (SMPs) [20]. In these approaches, the probabilistic transfer of control among components is assumed to be known. For instance, the probability is assumed to be a constant in DTMC-based approaches or a function of time in CTMC/SMP-based approaches.

In the following, we argue that because the CCS system's behavior relies on the run-time environment, the existing approaches are not ideal for it, nor for non-deterministic systems in general. The white-box approaches rely on modeling systems in DTMCs, CTMCs or SMPs, which imply that there is only one probability distribution out of any system component. In other words, if the system's behavior is hard to predict, the assumption that the probability distribution of transitions among system components is known should be problematic. For instance, if we model the CCS system using a DTMC, one probability distribution is required to capture the probability that an order is processed by different CCS servers. Obtaining this probability distribution is highly non-trivial as the target CCS server is chosen at run-time using a sophisticated dynamic load balancing algorithm. A probability distribution obtained in a testing environment is likely to be different from that of the real system. As a result, the estimated system reliability may lose its accuracy. A "safer" (and more convincing to the stakeholder) prediction is to assume no knowledge on the distribution and assume that an order may be *nondeterministically* assigned to any CCS server. The existing black-box approaches rely on testing the overall systems. However, with a non-deterministic system under test, it is unclear how test cases should be generated systematically so that testing can provide a quantifiable level of "assurance".

There is another issue with the existing white-box approaches. Two inputs are required including the reliability of the system components and a model of the system architecture. The former is usually obtained simply through component-based testing, which could be misleading. For instance, a component which failed 2 out of 50 test cases and a component which failed 40 out of 1000 test cases would have the same reliability of 96%. It is, however, obvious that 96% for the second component is more accurate. In the CCS system, we have indeed discovered that the number of tests for different components varies significantly. Mixing these semantically different data in calculating the system reliability gives inaccurate results.

Combining testing and model checking From the above analysis, testing is ineffective in non-deterministic systems. On the contrary, model checking is well-known to be able to handle non-deterministic systems systematically [9, 5]. We thus propose to combine testing (in particular, hypothesis testing) and model checking (in particular, probabilistic model checking) for non-deterministic systems. The idea is to apply hypothesis testing to system components which are deterministic and use probabilistic model checking to lift the results through non-determinism.

In the example of reliability prediction and distribution, we propose to apply hypothesis testing to measure component reliability with error bounds, and to use MDP-based probabilistic model checking to obtain system-level reliability. Hypothesis testing is one of the testing methods [33], and can be used to bound the number of test cases by indicating when the test can be stopped [38]. With hypothesis testing, users can quantify the accuracy of a test result by giving error bounds, i.e., the probability of false positive and false negative testing conclusions. MDPs are used in our probabilistic model checking. Compared to DTMCs, MDPs support nondeterminism, i.e., there may be multiple probability distributions from a state in the model. With its expressiveness, we can then properly model complicated systems like the CCS. However, compared to DTMC-based

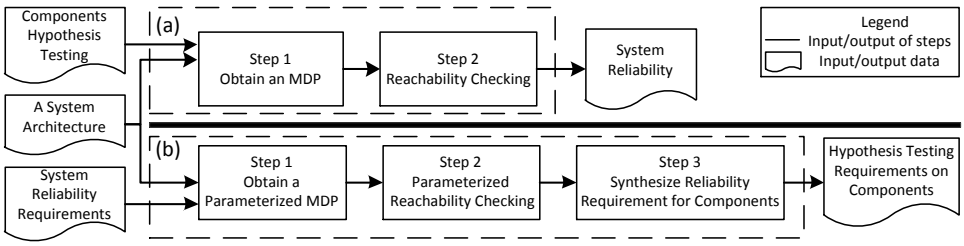

Figure 2: Workflow: (a) reliability prediction; (b) reliability distribution

reliability prediction or distribution, MDP-based algorithms are more challenging, as we show later.

Figure 2-a shows our workflow of solving the reliability prediction problem. Firstly, hypothesis testing is applied to obtain the reliability of system components. The result is a probability, i.e., the reliability of a component being larger or equal to this probability, with error bounds defined by users. Next, MDP-based reachability analysis is used to compute the overall system reliability, which is the probability that the system reaches the success state. Notice that existing algorithms on probabilistic reachability checking must be extended to handle the error bounds obtained with hypothesis testing. Figure 2-b shows the workflow of solving the reliability distribution problem. Given a reliability requirement for the system with error bounds, we solve the problem in three steps. Firstly, we construct a parameterized MDP model within which each component is associated with variables representing its reliability measurement. Next, we develop a parameterized probabilistic reachability checking algorithm to obtain the minimum constraints on the variables. Lastly, we synthesize concrete reliability requirement for each component based on the constraints. We develop a toolkit named RaPiD to fully automate our approach and apply it to investigate two real-world systems. The rest of paper is organized as follows. Section 2 reviews background on MDP-based probabilistic model checking and hypothesis testing. Section 3 presents our approach on combining probabilistic model checking and hypothesis testing. Section 4 shows an application of our approach to reliability prediction and distribution. Section 5 evaluates our approach. Section 6 concludes with related works.

2. BACKGROUND

In this section, we briefly introduce the background on probabilistic model checking and hypothesis testing.

2.1 Probabilistic Model Checking for MDPs

Discrete Time Markov Chains (DTMCs) and Markov Decision Processes (MDPs) are popular choices to model probabilistic systems. Given a set of states S, a distribution is a function $\mu : S \to [0,1]$ such that $\Sigma_{s\in S}\,\mu(s) = 1$. Let $Distr(S)$ be the set of all distributions over S.

Definition A DTMC is a tuple $\mathcal{D} = (S, init, Pr)$ where S is a set of states; $init \in S$ is the initial state; $Pr : S \to Distr(S)$ is a transition function. □

DTMCs are discrete stochastic processes satisfying the Markov property. A DTMC model can be expressed by a stochastic matrix $P : S \times S \to [0,1]$ such that $\sum_{s'\in S} P(s,s') = 1$. An element $P(s_i, s_j)$ represents the transition probability from state s_i to state s_j. A state is an absorbing state if it has only self-looping outgoing transitions, i.e., $P(s_i, s_i) = 1$.

Definition An MDP is a tuple $\mathcal{M} = (S, init, Act, Pr)$ where S is a set of states; $init \in S$ is the initial state; Act is an alphabet; and $Pr : S \times Act \to Distr(S)$ is a labeled transition relation. □

Different from a DTMC, there may be multiple distributions from a state, and each is labeled with a different action in an MDP. Intuitively, given a state s, an action (and the corresponding distribution) is first selected nondeterministically by a *scheduler*, and then one of the successor states is reached according to the probability distribution. A scheduler is a function deciding which action to choose based on the execution history. A DTMC can be defined by an MDP $\mathcal{M}$ and a scheduler δ, which we denote as $\mathcal{M}_\delta$.

With different schedulers, a state s may be reached with different probabilities. The measurement of interest is thus the maximum and minimum reachability probabilities. Let B be a set of target states. The maximum probability of reaching any state in B is denoted as $P^{max}(\mathcal{M} \models \diamond B)$, which is defined as: $P^{max}(\mathcal{M} \models \diamond B) = \sup_\delta P(\mathcal{M}_\delta \models \diamond B)$. Similarly, the minimum is defined as: $P^{min}(\mathcal{M} \models \diamond B) = \inf_\delta P(\mathcal{M}_\delta \models \diamond B)$ which yields the lower bound of the probability of reaching B. The supremum/infimum ranges over all, potentially infinitely many, schedulers. Existence of optimal memoryless schedulers, in which the decision for choosing next action/distribution based on the current state is independent of the previous choices, has been proved in [5]. Based on the result, different methods (e.g., by value iteration [5]) have been developed to calculate the maximum and minimum reachability probabilities.

Value iteration is an iterative approximation technique used to calculate the maximum and minimum probabilities of reachability, and often yields better performance than solving linear programs in practice [21, 36]. In the following, we will demonstrate the application of value iteration on finding the maximum probability for reaching any state in B from the initial state. Let V be a vector such that, given a state s, $V(s) = P^{max}(\mathcal{M} \models \diamond B)$ is the maximum probability of reaching B from s. For instance, $V(init)$ is the maximum probability of reaching B from the initial state. First, using backward reachability analysis, we can identify the set of states X which have non-zero probability of reaching B, i.e., B is reachable from any state in X. Next, we iteratively build an approximation of V based on the previous approximation. Let V^i be the i-th approximation. We define V^i such that $V^i(b) = 1$ for all $b \in B$ and any i; $V^i(n) = 0$ for all $n \notin X$ and any i; and for each state $s \in X - B$, we have

$V^0(s) = 0$;

$V^{i+1}(s) = max\{\sum_{t\in S} Pr(s,a,t) \times V^i(t) \mid a \in Act(s)\}$.

It can be shown that for every state s, $V^{i+1}(s) \geq V^i(s)$ and we can obtain V in the limit, i.e., $\lim_{i\to\infty} V^i = V$. In reality, it may take many iterations before V^i converges and thus value iteration is often stopped using a number of different conditions (e.g., when a fixed number of iterations have been reached or when the difference between two suc-

cessive iterations falls below a certain threshold). Minimum probability of reaching B can be calculated similarly.

Each iteration involves a matrix-vector multiplication, which has a complexity of $\mathcal{O}(n^2 \times m)$ in the worst case, where n is the number of states in S and m is the maximum number of actions from a state. Note that for sparse MDP models, the complexity is often $\mathcal{O}(n \times m)$. The number of iterations required to achieve certain numerical precision is related to the subdominant eigenvalue of the transition matrix [35].

2.2 Hypothesis Testing

Hypothesis testing is a statistical process to decide the truthfulness of two mutual exclusive statements: H_0 and H_1, where H_0 is the hypothesis that the probability of a given event is larger than or equal to a given value p_0, and H_1 is the alternative hypothesis (i.e., the probability of the event is less than or equal to a given value p_1). Besides, two parameters are required from users. One is the targeted assurance level (θ) over the system, and the other is the indifference region (2σ). Indifference region refers to the region (p_1, p_0), used to avoid exhaustive sampling and obtain the desired control over the precision [39]. With the input θ and σ, $p_0 = \theta + \sigma$, $p_1 = \theta - \sigma$. The probability of accepting H_1 given that H_0 holds is required to be at most α, called false negative, and the probability of accepting H_0 if H_1 holds should be no more than β, called false positive. In practice, the error bounds (i.e., α, β), and σ can often be decided by how much testing resource the component developer has. In general, it would require more resource for a smaller error bounds or a smaller indifference region.

Hypothesis testing has been applied for reliability estimation [38]. Let R be the reliability of a module. Suppose that we wish to test the hypothesis that the reliability R is at least θ. With a user defined σ, we have hypothesis $H_0 : R \geqslant \theta + \sigma$ and $H_1 : R \leqslant \theta - \sigma$. We remark that hypothesis testing requires a way of sampling system executions according to its operational usage. Many sampling methods have been developed and applied for software demonstrating testing [37, 34]. There are two main acceptance sampling methods to decide when testing can be stopped. One is fixed-size sampling test, which often results in a large number of tests [39]. The other one is sequential probability ratio test (SPRT), which yields a variable sample size. SPRT is faster than fix-sampling methods as the testing process ends as soon as a conclusion is made. The basic idea of SPRT is to calculate the probability ratio, after observing a test result and comparing with two stopping conditions [4]. If either of the conditions is satisfied, the testing stops and returns which hypothesis is accepted. Readers can refer to [39] for details.

Example In the CCS system, to verify whether the reliability of a CCS server is at least 0.8, users should define the test parameters (i.e., σ, α and β). Assuming $\alpha = 0.01$, $\beta = 0.01$, and $\sigma = 0.1$, the parameters define the goal for the testing, i.e., whether to accept H_0 "the reliability of the CCS Server is at least 0.9" or H_1 "its reliability is at most 0.7". If the "true" reliability is at most 0.8, it is guaranteed that the probability of wrongly accepting H_0 is less or equal to 0.01. By the stopping criterion, we have $x_m \geqslant 0.8138m + 3.4040$ (to accept H_0) or $x_m \leqslant 0.8138m - 3.4040$ (to accept H_1). We start the testing with $m = 0$; $x_m = 0$. After executing a test case (which is chosen randomly according to the user profile), m increases by 1. x_m either increases by 1 if the test finishes without failure or remains the same otherwise. Next, the updated m and x_m are used for stopping criteria. If any one of the stopping criteria is fulfilled, H_0 or H_1 is accepted accordingly; otherwise, sampling continues with another test case and the above steps will be repeated. □

The error bounds quantify the reliability measurement. They can differentiate the above-mentioned case, i.e., two different test cases which have concluded the same reliability of 96% with different number of tests. Assuming that a fixed sampling plan is adopted with θ set to 0.96 and σ set to 0.01, the minimum error bounds are: $\alpha = 0.5262$ and $\beta = 0.3357$ for 2 failures out of 50 tests case; and $\alpha = 0.2161$ and $\beta = 0.2026$ for the case of 40 failures out of 1000 tests. Therefore, the result based on the larger sample size is more accurate in terms of smaller error bounds.

SPRT is guaranteed to terminate [4], while the expected sample size is hard to determine. Wald [3] has provided a good approximation. The expected sample size increases from 0 to p_1 and decreases from p_0 to 1. The worst case is when the "true" probability is within the indifference region. If $a = 0.01, b = 0.01, p_0 = 0.99$, and $p_1 = 0.98$, the expected sample size will be 3.0005×10^3 by Wald's approximation. If considering the hypothesis testing parameters with high precision, which is normally the case in practice, e.g., $a = 0.001, b = 0.001, p_0 = 0.9999, p_1 = 0.9998$, the expected sample size will be 6.8811×10^5 in this case.

3. COMBINING MODEL CHECKING AND HYPOTHESIS TESTING

Hypothesis testing enables directly sampling on systems, but is not suitable to nondeterministic systems. On the contrary, probabilistic model checking can handle nondeterminism easily with exact solutions, but suffers from state explosion problem. The combination of both is proposed in such a way that hypothesis testing is conducted on each subsystem separately and probabilistic model checking method is performed on the system level modeled in an MDP. This can be formally presented as follows.

Let $\mathcal{M}$ be an MDP, ϕ be a property (which can be in LTL [32], PCTL [15], etc.). By probabilistic model checking, it can calculate the probability of the set of paths in $\mathcal{M}_\sigma$ that satisfy the ϕ for all schedulers σ, denoted as $P(\mathcal{M} \models \phi)$. It can also check whether a property holds with probability at least θ, i.e., $P_{\geqslant\theta}(\mathcal{M} \models \phi)$, which returns a Boolean value. The model is assumed to be composed of several components, denoted as $\mathcal{M}(\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_n)$, where each $\mathcal{D}_i$ is a deterministic system component. We connect the probability of satisfying global property ϕ with the probability of satisfying the local properties for each component with the following function.

$$P(\mathcal{M} \models \phi) = \\ f\left(P\left(\mathcal{D}_0 \models \phi_0\right), P\left(\mathcal{D}_1 \models \phi_1\right), \ldots, P\left(\mathcal{D}_n \models \phi_n\right)\right)$$

where ϕ_i is the local property for component i and f is a function that takes in the probability of satisfying the local properties in each component and outputs the probability of satisfying the global property in the whole model.

Similarly, the verification task of comparing the probability of satisfying a global property with a bound relates to that of local properties in each components is as follows.

$$P_{\geqslant\theta}(\mathcal{M} \models \phi) = \left(P_{\geqslant\theta_0}\left(\mathcal{D}_0 \models \phi_0\right)\right) \wedge \left(P_{\geqslant\theta_1}\left(\mathcal{D}_1 \models \phi_1\right)\right) \wedge \ldots$$

where $\theta = f(\theta_0, \theta_1, \ldots, \theta_n)$. In each deterministic component $\mathcal{D}_i$, the probability of satisfying a local property ϕ_i be-

ing larger than a given value θ, denoted as $P_{\geqslant\theta_i}(\mathcal{D}_i \models \phi_i)$, can be verified by hypothesis testing.

With the setting above, we show how to solve two different problems as explained below. Firstly, if the objective is to obtain a probability of a system satisfying a global property, we first perform hypothesis testing to obtain the probability of each component satisfying the corresponding local property. Notice that we need to specify certain discrete levels from high to low, e.g., $l1 = 1$, $l2 = 0.99$, etc. Hypothesis testing is performed against those discrete levels (e.g., l_i) sequentially, until $P_{\geqslant l_i}(\mathcal{D}_i \models \phi_i)$ is true. The l_i is the approximated maximum probability of satisfying the local property. Afterwards, we perform probabilistic model checking based on the obtained results to calculate $P(\mathcal{M} \models \phi)$. Second, a global verification task, i.e., $P_{\geqslant\theta}(\mathcal{M} \models \phi)$, can be distributed into several local verification tasks, i.e., $P_{\geqslant\theta_i}(\mathcal{D}_i \models \phi_i)$ for each component $\mathcal{D}_i$. This relies on the assumptions that there is an inverse function of the given f, denoted by f^{-1}, such that each θ_i is calculated from $f^{-1}(\theta)$; whenever the result of $P_{\geqslant\theta_i}(\mathcal{D}_i \models \phi_i)$ is false for a component $\mathcal{D}_i$, $P_{\geqslant\theta}(\mathcal{M} \models \phi)$ is also false, and vise versa.

Besides, it is still necessary to analyze how error bounds at system level are related to the ones at components.

LEMMA 3.1. *Let α_c and β_c be the error bounds of verifying components $\mathcal{D}_c$. Let $\mathcal{C}$ be the set of all components of a system. Let the error bounds for the overall system be (α, β). α is bounded by $max\{\alpha_c \mid c \in \mathcal{C}\}$ and β is bounded by $\sum_{c\in\mathcal{C}} \beta_c$.*

Proof Let $\mathcal{M}$ be a system. We define the following hypothesis: Φ ("Accept $P_{\geqslant\theta}(\mathcal{M} \models \phi)$ is true"); Φ_s ("Accept $P_{\geqslant\theta_s}(\mathcal{D}_s \models \phi_s)$ is true"); $\neg\Phi$ ("Accept $P_{\geqslant\theta}(\mathcal{M} \models \phi)$ is false"); $\neg\Phi_s$ ("Accept $P_{\geqslant\theta_s}(\mathcal{D}_s \models \phi_s)$ is false"); Ψ ("In fact, $P_{\geqslant\theta}(\mathcal{M} \models \phi)$ is true"); Ψ_s ("In fact, $P_{\geqslant\theta_s}(\mathcal{D}_s \models \phi_s)$ is true"); $\neg\Psi$ ("In fact, $P_{\geqslant\theta}(\mathcal{M} \models \phi)$ is false"); and $\neg\Psi_s$ ("In fact, $P_{\geqslant\theta_s}(\mathcal{D}_s \models \phi_s)$ is true"). Let (α, β) be the error bounds for verifying $P_{\geqslant\theta}(\mathcal{M} \models \phi)$. By definition, α is $\Pr(\Psi \mid \neg\Phi)$, i.e., the probability of false negative.

$$\begin{aligned}\alpha &= \Pr(\Psi \mid \exists\, c \in \mathcal{C}.\neg\Phi_c) && -\star\\ &\leqslant max\{\Pr(\Psi \mid \neg\Phi_s) \mid c \in \mathcal{C}\}\\ &\leqslant max\{\Pr(\Psi_c \mid \neg\Phi_s) \mid c \in \mathcal{C}\}\\ &\leqslant max\{\alpha_c \mid c \in \mathcal{C}\}\end{aligned}$$

$(\star)$ holds because "accepting that $P_{\geqslant\theta}(\mathcal{M} \models \phi)$ false" is equivalent to "accepting that there exists a $\mathcal{D}_s$ such that $P_{\geqslant\theta_s}(\mathcal{D}_s \models \phi_s)$ is false".

Similarly, β is $\Pr(\neg\Psi \mid \Phi)$, i.e., the probability of false positive.

$$\begin{aligned}\beta &= \Pr(\neg\Psi \mid \Phi) = \Pr(\neg\Psi \mid \forall\, c.\ \Phi_c)\\ &= \Pr(\exists\, c \in \mathcal{C}.\ \neg\Psi_c \mid \forall\, c \in S.\ \Phi_c)\\ &\leqslant \textstyle\sum_{i\in\mathcal{C}} \Pr(\neg\Psi_c \mid \forall\, c \in \mathcal{C}.\ \Phi_c)\\ &= \textstyle\sum_{c\in\mathcal{C}} \Pr(\neg\Psi_c \mid \Phi_c) = \sum_{c\in\mathcal{C}} \beta_c && -\square\end{aligned}$$

The setting above is beneficial in terms of alleviating state space explosion problem and easily handling system nondeterminism. However, the general form above depends on a few assumptions which are not easy to be satisfied in general. We assume verification of a global property ϕ can be divided into the verification against local properties $\phi_0, \phi_1, \ldots, \phi_n$; and the probability of satisfaction in the system is related to that of components by a function f. In general, such function is hard to attain, not to mention its inverse function f^{-1}.

Nonetheless, f can be obtained in some cases. A special case is when all local properties are the same as a global property, i.e., $\phi_i = \phi$ for the component $\mathcal{D}_i$. The function f is an evaluation of the MDP model which can be done by numerical methods e.g., value iteration. In the following, we show that the problem of reliability analysis is exactly such a special case and thus can be solved efficiently.

4. RELIABILITY ANALYSIS

The special case can readily apply to the software reliability analysis, i.e., reliability prediction and reliability distribution, by setting local properties and modeling function "f" explicitly. The property of interest in reliability analysis is the probability that a software has no failure. This is a global property, denoted by $P(\mathcal{M} \models \Box\neg\mathit{failure})$. The global property is actually composed by a set of local properties, i.e., probability of each module running successfully without any failure. An MDP model is built from the system architecture and users environment. Each module in the system can be treated as a component $\mathcal{D}$ in the MDP. The transition probability between components, e.g., P_{ij}, is the probability from component i to component j, given that component i does not fail, i.e., P_{ij} is conditional on $P(\mathcal{D}_i \models \Box\neg\mathit{failure})$. Therefore, the transition probability of the MDP, e.g., from component i to component j, is $P(\mathcal{D}_i \models \Box\neg\mathit{failure})P_{ij}$. Here, conditional probability p_{ij} can be estimated from usage profile [29, 18]. By reachability checking on the MDP (e.g., via value iterations), the relationship between $P(\mathcal{M} \models \Box\neg\mathit{failure})$ and $P(\mathcal{D}_i \models \Box\neg\mathit{failure})$ for each component i can be established.

In the following, we present the details of applying the combination of hypothesis testing with probabilistic model checking to software reliability analysis. We first set up assumptions of our reliability analysis, followed by the construction of an MDP from system architecture and operational environment. The methodologies for reliability prediction and distribution are then introduced, respectively.

4.1 Assumptions and Threads to Validity

Considering reliability analysis as a special application, there are some underlying assumptions in terms of system reliability model and component failure behavior.

We use an MDP to model a system. Each state represents the execution of a single component of the application. Same as other Markov models, our model relies on the assumption of Markovian transfer of control among components, i.e., the probability distribution of future executing components depends only upon the present components.

Similar to [7, 18, 14, 13], our model also assumes that there is statistical independence among failures of the components. More specifically, the failure occurring within one component is neither the result of a failure occurring within another component, nor able to cause any other component to fail. However, in practice, different component may be heavily dependent, which may be a result of data exchange occurring through parameters or messages passing. In this work, we limit ourself to the applications whose components are failure independent. It appears to be a strict condition. However, the present assumption can be well satisfied considering many up-to-date large systems (e.g. CCS), within which the components are designed, implemented and tested independently. If any failure dependent components exist and can be grouped into one, the model would still work.

Moreover, in our reliability model, we assume that failure of any component will eventually lead to the failure of the system. For a system consisting of self-recovery or self-correction mechanism, there are some executions that end successfully after recovery. These scenarios are not considered as failure cases. Nonetheless they can be modeled in an MDP.

4.2 System Level Modeling

When to use nondeterministic choices? Compared to DTMCs, MDPs allow us to capture both probabilistic and nondeterministic behavior. A central issue is: when to use nondeterministic choices and when to use probabilistic choices. In general, probabilistic choices can be viewed as informed nondeterministic choices. That is, we use a nondeterministic choice when we have no definitive information on how the choice is resolved. For instance, if all we know is that there are two different outgoing transitions after executing a component C, we model the two transitions using a nondeterministic choice. If the choice is made locally, after testing C systematically, we learn the frequency of each outgoing transition and we can model C with a probabilistic choice. However, if the result of executing C is correlated to its inputs, there are two cases. If the inputs are the result of executing some other component K in the system, we may either model it as a nondeterministic choice conservatively; or we calculate the probability distribution of C's results based on the probability distribution of K's results. Notice that if we systematically test C and K as a whole, we may obtain a probability distribution of C's results. However, if the inputs of C are from an external environment which is difficult to predict (e.g., like the traffic of stock transactions), a nondeterministic choice would deliver a "safer" model.

In a nutshell, testing helps to turn nondeterministic choices into probabilistic choices. Ideally, we would like to learn probability distribution of all actions in the system. Nonetheless, due to the limited resources for testing or knowledge of the external environment, we often have to employ nondeterministic choices.

Example In the following, we illustrate the difference between nondeterministic choices and probabilistic choices using a simple example. Figure 3 presents a simplified fragment of the CCS system. There are two components $S1$ and $S2$, with reliability 0.8 and 0.9, respectively. The components execute simultaneously and independently. Assume that $S1$ is chosen 30% of the time. The corresponding DTMC model is shown on the left of the figure. The system reliability is then estimated as $0.3 \times 0.8 + 0.7 \times 0.9$, which is 0.87. There are two potential problems with the above prediction. First, the two components are running in parallel and hence a DTMC cannot truthfully model the system. Second, the transition probability is decided through user profiles, which can only be obtained within limited tests in a testing environment before the system is deployed. The transition probability is hardly accurate since it is determined by the dynamic tasks loading at run-time. If an MDP is used to model the system, as shown on the right, the reliability is calculated as 0.8 if $S1$ is chosen and 0.9 if $S2$ is chosen. The result shows that in the worst case, the system is only as reliable as $S1$. We can see that the result based on the MDP model is less dependent on the external environment. □

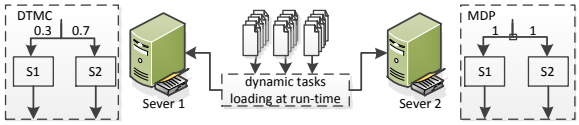

Figure 3: A system with run-time tasks distribution

In the CCS example, depending on the run-time traffic of the transaction, a sophisticated dynamic load balancing algorithm is used to distribute transactions to the three CCS servers. There are 15 different combinations of choices, i.e., any of the three is chosen; two of them are chosen in a particular order; or all three of them are chosen in a particular order. It is challenging, if not impossible, to predict the probability distribution of the choices. Modeled as nondeterministic choices, these choices can be distinguished by different schedulers and the system reliability for each choice can be calculated and compared during value iteration.

Reliability Modeling The model of a system in our setting is an MDP $\mathcal{M} = (S, init, Act, Pr)$. For each system component C (i.e., a self-contained piece of codes that can be independently designed, implemented, and tested), there is a pair of states, C and xC, in S which represent the state of C executing and the state right after C terminates, respectively. Further, S contains two absorbing states: a state of *Success* and a state of *Failure*. Here, the probability of a system always not getting *failure* state is the same as the probability of a system eventually reaching *success* state, denoted as $P(\mathcal{M} \models \diamond Success)$. If the reliability of C is R_C, there is one probability distribution from C such that there is probability R_C to reach xC and probability $1 - R_C$ to reach *Failure*. Notice that if there is certain failure handling mechanism in the system (like WPF in the CCS system), the transition with probability $1 - R_C$ leads to a failure recovering state instead of the *Failure* state.

A simplified model of the CCS system is shown in Figure 4. The data in this figure are obtained based on test results on an early version of the system. Only the nondeterministic choices among the three CSS servers are shown and we further ignore the ordering among the three servers so as to save space. For compact presentation, we skip the *Failure* state. Instead, a node labelled as $C(R)$ is used to denote that the name of the component is C and the probability of reaching *Failure* is $1 - R$ and probability of R to transit to the successive components. The transition probability at each edge represents the usage information. Taking $Server1$ as an example, its reliability can be read off from the graph as 0.9972 and it has three outgoing transitions labeled with action η. If $Server1$ terminates successfully, it has a probability 0.584 of going to *Exit*, and a probability 0.416 of going to *DB*. If $Server1$ fails, it goes to $Server2$, which serves as a backup server for $Server1$. A backup transition is denoted by a dash line in the figure. In the corresponding MDP, the three transitions are labeled with $(\eta, 0.9972 \times 0.584)$, $(\eta, 0.9972 \times 0.416)$, and $(\eta, 1 - 0.9972)$, respectively. Note that if action $(\epsilon, 1)$ is chosen at *Enter*, there is a backup server available for $Server1_a$ or $Server2_a$. If action $(\sigma, 1)$ or $(\omega, 1)$ is selected instead, there are no backup servers available, i.e., both servers are failed or the three servers are all running to finish a job.

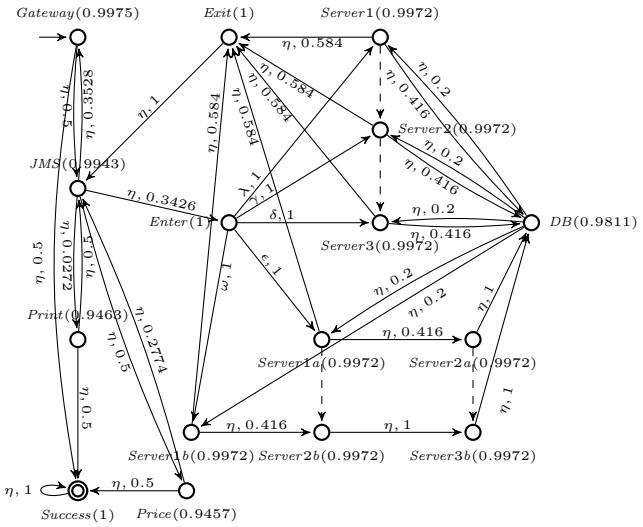

Figure 4: A simplified model for the CCS system

4.3 Reliability Prediction

Given the hypothesis testing results of each component and an MDP, we then calculate the overall system reliability.

Based on Lemma 3.1, if all components are tested with the same error bounds (α_c, β_c), the error bounds of the overall system are $(\alpha_c, \beta_c \times N)$, where N is the number of components. Notice that system-level false positive β could be N times larger than the one at component level. This implies that the confidence of system level measurement is lower than that of the components.

The maximum and minimum system reliabilities can by calculated by $P^{max}(\mathcal{M} \models \diamond Success)$ and $P^{min}(\mathcal{M} \models \diamond Success)$, respectively, and can be calculated using the value iteration method. Given the CCS model[1] in Figure 4, we obtain that the system reliability ranges from 0.95505 to 0.95729. The worst reliability is obtained with a scheduler such that three servers are running together, whereas the best reliability is obtained when only one is running. Detailed analysis is discussed in Section 5.

4.4 Reliability Distribution

Our approach on distributing the overall system reliability requires two inputs: (1) a reliability requirement R on the overall system and a pair of error bounds (α, β); (2) a system model in the form of an MDP. The goal is to find a reliability requirement on some components so that the overall reliability requirement is satisfied. The resultant requirement on the components (e.g. component c), is in the form of a reliability probability R_c and a pair (α_c, β_c), which can be established using hypothesis testing on the smaller-scale component. In the following, we first show how to identify R_c and then how to obtain (α_c, β_c).

Given an MDP $\mathcal{M}$ and a scheduler δ, we can obtain a DTMC $\mathcal{M}_\delta$. The probability of reaching the *Success* state, $P(\mathcal{M}_\delta \models \diamond Success)$, is a polynomial function constituted by multiple variables (i.e., R_c for all relevant components). The constraint $P_{\geqslant R}(\mathcal{M}_\delta \models \diamond Success)$ then gives us the reliability requirement on each component, under the scheduling of δ. However, such a constraint is hardly useful in practice as the reliability of the components constraint each other. For simplification and making the results useful in practice, we assign different weights for the components participating in reliability distribution, by considering testing costs, e.g., testing time, and effort. In practice, the software can make use of some readily developed components. The components whose reliability is already known and rarely changes (e.g., a legacy component), will not participate in reliability distribution.

As a result, $P_{\geqslant R}(\mathcal{M}_\delta \models \diamond Success)$ becomes a polynomial inequality constituted by a variable x only. Using numerical methods, we can obtain a lower bound on x, which is the reliability requirement we need. Multiplying x with assigned weights, the reliability requirement for the components participating in reliability distribution can be obtained. Take the model in Figure 4 for example. Assume R is 0.98 and the scheduler δ_1 resolves the nondeterministic choice at state *Enter* by selecting action η. We further assume a unit weight assigned to all components, the calculated polynomial using our algorithm above is $0.5x^1 + 0.16435x^3 + 0.05402x^5 + 0.11641x^7 - 0.09865x^8 \cdots \geq R$. When the iterations stop, the polynomial is accumulated up to the term of x^{160}. We omit the result of the terms here. By Newton's method [2], we obtain that the lower bound on x is 0.99601. This is the reliability requirement for every component since we assuming the same weight.

The above concerns only one scheduler. In general, there are multiple schedulers and we need to guarantee that the system reliability requirement is satisfied with any scheduler. Applying value iteration directly is very challenging as we need to compare polynomial functions representing the probability of reaching *Success* through different distributions from a state in each iteration. We thus adopt an alternative approach, i.e., we compute a lower bound on x for every scheduler and the maximum of the lower bounds gives us the minimum requirement on component reliability. Based on [5], only finitely many memoryless schedulers need to be considered. Our algorithm works as follows. First, an unvisited memoryless scheduler δ is selected. Next, we perform the value iteration method on $\mathcal{M}_\delta$. The following shows how the result vector V is updated. Assume scheduler δ chooses a distribution μ_s at state s: $V^{(n+1)}(s) = \sum_{t \in S} x \times \mu_s(t) \times V^{(n)}(t)$. Once a stopping condition is satisfied, we obtain a constraint $V(init) \geq R$ and solve the equation $V(init) - R = 0$ using Newton's method to obtain a lower bound on x so that $V(init) \geq R$ is true. The steps above are repeated for all memoryless schedulers.

The upper bound of memoryless schedulers equals to the product of the numbers of distributions for each state. If there are ten states and two of them both have 3 distributions and the rest has one, the number of schedulers is bounded by 9. Essentially, the more nondeterminism there is, the more schedulers are to be considered. In practice, the number of schedulers are manageable as we are dealing with a high-level system model. Further, since schedulers are independent, we can parallelize the computation.

Next, we distribute the system error bounds (α, β). We assume that error bounds of each component are the same,

[1]The reliability are relatively low as they are obtained from test environment before the software released. The data for released version is confidential from the company. We have demonstrated that our method can still work and is accurate to a certain level by assuming relatively high reliability of each component (e.g., 99.999%), which is often the case after software released, available at [1].

denoted as (α', β'). Based on Lemma 3.1, we can deduce the following constraints: $\alpha' \leqslant \alpha$ and $\beta' \leqslant \frac{\beta}{N}$. Therefore, the two error bounds for each component are α and $\frac{\beta}{N}$, respectively. Given the model in Figure 4, we assume the system error bounds are $(0.02, 0.04)$. Since N is 8, the error bounds for each component are $(0.02, 0.005)$. β' might be considerably smaller than β if N is large. A very small error bound for hypothesis testing may lead to a large number of tests. The practical implication is that one can estimate system reliability by testing either larger components with larger error bounds (which is harder to test but needs less tests) or smaller components with smaller error bounds (which is easier to test but needs more tests).

5. IMPLEMENTATION AND EVALUATION

The proposed approach has been realized in a toolkit named RaPiD (Reli*a*bility Pred*i*ction and Distribution). RaPiD is a self-contained toolkit for reliability prediction and distribution, and it is publicly available at [1], with all case studies. It provides a user friendly interface to draw MDP models as well as fully automated methods to solve the reliability prediction and distribution problems. RaPiD is implemented with 4K lines of C# code. It uses a number of MATLAB (version 2009a) libraries to support powerful mathematical calculations as well as graph plotting functions. In the following, we apply RaPiD to study two real-world systems and obtain interesting results.

5.1 Reliability Prediction for the CCS

The CCS system has more than 300K lines of code. To predict the reliability of the CCS system, we first build an MDP model (as partly shown in Figure 4) and then test each of the components. Next, we apply RaPiD with a stopping criterion of 1E-5 and obtain the minimum/maximum system reliability of 0.95505/0.95729, respectively.

To compare the effectiveness of different models, i.e., the effect of nondeterministic choices for system reliability prediction, we build three models in total. Model $M1$ is a DTMC model, assuming that the probability distribution of all transitions among system components are known; $M2$ is as shown in Figure 4 where nondeterministic choices are used to model the run-time choice of the CCS servers, and $M3$ further introduces nondeterminism by modeling the choices of going back to JMS or DB from a CCS server nondeterministically. To investigate the usefulness of the back-up servers in terms of system reliability, we modify these three models to incorporate transitions leading to a backup server. The resultant models are denoted as $M1b$, $M2b$ and $M3b$, respectively.

As presented, if we assume all components have the same reliability x, we can obtain system reliability as a polynomial function of x for each scheduler. Using RaPiD, we can plot the functions, as shown in Figure 5. Different from $M1$, $M2$ has 5 schedulers, and $M3$ has 160 schedulers in total since it has a state with 5 nondeterministic choices and another five states each with 2 nondeterministic choices. The dash lines are the corresponding functions for $M1b$, $M2b$ and $M3b$, respectively. Notice that many of functions are identical (e.g., for $M3$) and their plots overlap with each other.

The following observations can be made based on the results. First, the difference between maximum reliability and minimum reliability becomes larger when the number of nondeterministic choices increases. For instance, Table 5.1 shows the differences for the three models.

Table 1: Reliability prediction for the three models

Name (#Schedulers)	M1 (1)	M2 (5)	M3 (160)
Min. Reliability	0.95568	0.95401	0.73149
Max. Reliability	0.95568	0.95568	0.96257

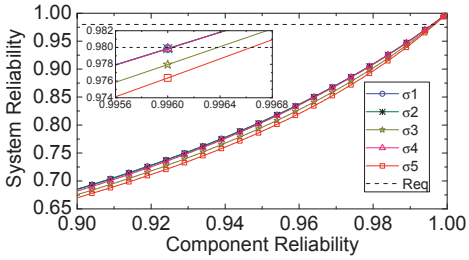

Figure 6: Reliability analysis result for the CCS

Assuming that we need to show the system's reliability is at least 0.95, the result based on $M3$ is not conclusive, which suggests further testing is necessary so that we can learn the probability distribution of the nondeterministic choice (i.e., the transitions from CCS servers to JMS or DB) and make more accurate prediction. The result based on $M2$ on the other hand shows that we can make fairly accurate prediction without making any assumption on the run-time dynamic loading decisions, and this serves as a strong argument that the system is robust in the open dynamic stock market. If we superimpose the results for $M1$, $M2$ and $M3$ (i.e., the solid curves in Figure 5 (a), (b), (c)), we can find that the curve in graph (a) resides between the curves in graph (b), all the curves in (a) and (b) reside between the curves in graph (c). Second, in all three graphs, the dashed curves are higher than the corresponding solid curves. It implies that the system reliability indeed becomes higher by introducing a backup server. Nonetheless, it should be noticed that with the increase of component reliability, the gain of system reliability by introducing the backup server decreases. The results confirm (and quantify) our intuition.

5.2 Reliability Distribution for the CCS

RaPiD solves the reliability distribution problem using the approach documented in Section 4.4. There are totally 15 different choices of servers operation modes for the CCS, which indicates 15 schedulers existing in the model. Figure 6 visualizes the results for those five schedulers for the model shown in Figure 4, where scheduler $\sigma_1/\sigma_2/\sigma_3$ chooses action $\eta/\lambda/\gamma$ (i.e., to run $Server1/Server2/Server3$ only, respectively) at state $Enter$; scheduler σ_4 leads to state $Server1_a$ and possibly state $Server2_a$ subsequently (i.e., to run $Server1$ and $Server2$ at the same time); and scheduler σ_5 leads to a state where all three servers are running.

Assume that the system reliability requirement is 0.98 (i.e., the horizontal dash line in Figure 6). When all curves are above the dash line, we have sufficient component reliability to guarantee the system is reliable with any scheduler. Given the same component reliability, a scheduler is "better" if its corresponding system reliability is higher. Similarly, given the same system reliability, a scheduler is "better" if its corresponding component reliability is lower. Notice that when component reliability is low, the less severs are chosen to work simultaneously (e.g., scheduler σ_1 and σ_2),

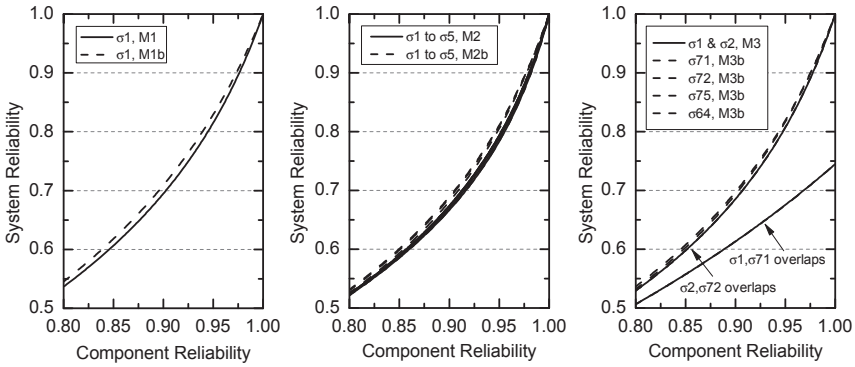

Figure 5: System reliability vs. component reliability for: (a)M1 and M1b; (b)M2 and M2b; (c)M3 and M3b

the higher the system reliability is achieved; and as component reliability becomes higher (e.g., > 0.95), the schedulers (e.g., σ_4) leading to more servers running outperform the others.

5.3 Reliability Distribution for the TCS

The Burr Proton Therapy Center is a radiation therapy facility associated with the Massachusetts General Hospital in Boston. Proton therapy is a treatment controlling the dose of radiation delivered to the patients. High precision radiation therapy enables reduced dose to healthy tissue. Reliability assurance on such system is of uttermost importance. One software component of the system, called the Therapy Control System (TCS), provides the users with all the control functions necessary. It is written primarily in 250K lines of C code. The TCS handles the storage and retrieval of patient data entry of prescriptions, scheduling of treatments, patient positioning and beam delivery.

A high-level view of the system is shown in Figure 7. The *Human/Computer Interface Layer* is a graphical user interface. The *Application Layer* is the core of the system. It consists of four modules: System Manager (SM), which controls operational modes, and event reporting; Beam Manager (BM), which handles allocation and operation of the proton beam transport; Treatment Manager (TM), which handles the patient treatment sequence from prescription to irradiation; and Database Manager (DM), which provides functions to allow the other modules to access to the database. The *Control Unit Layer* contains drivers for the physical devices, including Accelerator Control Unit (ACU), Energy Selection and Beam Transport Control Unit (ECU-BTCU), Positioning Control Unit (PCU), Treatment Control Unit (TCU), and Safety Control Unit (SCU). These are implemented in a table-driven fashion as low level state machines. RTServer is the information distribution server. It manages all communication among client processes, freeing all low-level network coding. RTH1 and DataDAQ are two data acquisition interfaces. RTH1 is in charge of ACU and SCU, while DataDAQ is in charge of the rest of control units. The service starts with any beam service requirements sent via Human/Computer Interface Layer, and completes after the BM generating the irradiation summary.

The TCS system serves as an excellent case study for our reliability distribution method as it is presently undergoing a software upgrade. Some components are to be revised or replaced. Given the requirement on system level reliability, it is desirable to generate concrete reliability requirement for newly developed components so that they are contracted

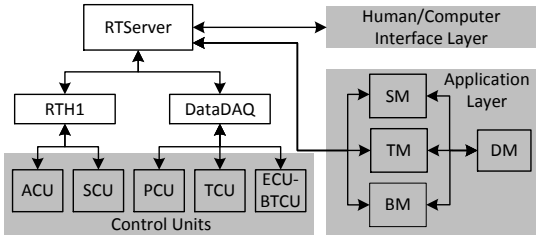

Figure 7: Architecture of a Therapy Control System

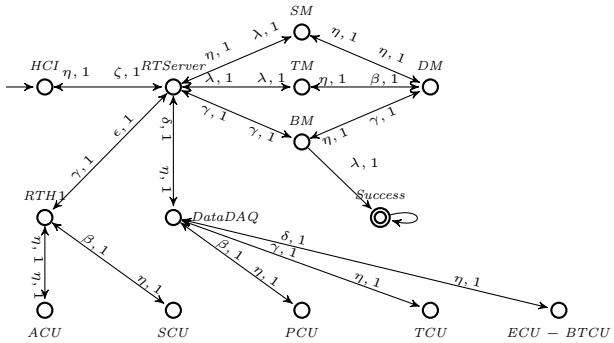

Figure 8: A reliability model for the TCS

properly. The challenge in applying RaPiD is that there is no precise information on transition probabilities. The reason is that testing the system is highly complicated, as there are 5 concurrent machines and many interrupting events generated by hardware control units. However less complex safety mechanisms are in place to mitigate any error. As a result, transition probability and the system are modeled by non-determnistic choices only. Nonetheless, we show that we can still obtain some useful results.

A simplified MDP model of the system is shown in Figure 8. As the reliability of each component is not available, they are omitted. Although there are 2,592 schedulers, only three different system reliability functions of component reliability exist. By further analyzing the corresponding schedulers reported by RaPiD, we can identify three typical workflows of the system that result in the three scenarios, respectively.

In Figure 9, the plot of the worst scenarios is a horizontal line of zeros. It implies that for any component reliability, the system level reliability is zero, i.e., the system cannot reach the *Success* state. This set of schedulers always chooses *RTH*1 or *DataDAQ* from *RTServer* and hence

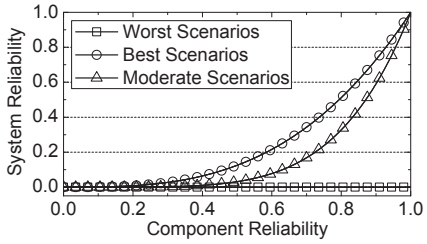

Figure 9: Reliability analysis result for the TCS

state *Success* is never reached. The best scenarios include the cases within which the transition directly goes from *RTServer* to *BM* and then reaches *Success*. In real situation, this is an extreme case that the system sends beam treatments request directly to the *BM*, which completes the job, and then the whole transition finishes successfully. Moderate scenarios contain cases within which *RTServer* goes to *SM*, *TM*, or *BM* and then goes to *DM*; and afterwards, *DM* reports data back to *BM* and lastly the *Success* state is reached. As we can see, without any testing results on the system, we are able to find out the worst/best system reliability in respect to components reliability. This information is particularly helpful in the early stage of software development, as the system developers can use the results as a guideline on how to test the system or how to improve the system reliability, e.g., by improving the feedback communication from control unit layer to RTServer.

5.4 Scalability

RaPiD is efficient in our case studies. The reliability prediction took 0.03 seconds for the CCS, and the reliability distriution took 42 seconds for the CCS (with 160 schedulers) and 628 seconds for the TCS (with 2,592 schedulers). To further test the scalability of RaPiD, we evaluate RaPiD's reliability prediction and distribution using 5 benchmark MDP models from [21] as well as randomly generated models (with 1K to 50K states and the number of states for having multiple transitions are sampled from a uniform distribution). The results show that RaPiD is able to handle 14K states per second on average (with termination threshold as relative difference 1.0E-6) in calculating reachability probability. Reliability distribution (with a bound 600 on the number of terms in the obtained polynomial) is slightly slower due to maintaining/updating/solving the polynomial functions. The data is obtained using a PC with Intel(R) Core(TM) i7CPU at 2.80 GHz and 8 GB of RAM.

6. RELATED WORK AND CONCLUSION

Hypothesis testing has gained its popularity in probabilistic model checking [8, 40, 24] as it can overcome state space explosion problem. Its applications were limited to deterministic systems in the early stage. In recent years, it has been extended to nondeterministic systems [6, 16, 23]. [6] provides an approach that only limits to spurious nondeterminism that introduced by the commutativity of concurrently executed transition in compositional setting. [16, 23] applies learning techniques to search for near optimal schedulers so as to convert MDP to an induced Markov Chain. The effectiveness in searching for the near-optimal schedulers is decided by several parameters for controlling the maximum number of schedulers to evaluate each time and the effectiveness of learning process. All those parameters shall be tuned by users. Instead, our approach lifts up nondeterminism to a level that exact methods can be used.

Our work can also be viewed as performance analysis of programs with probabilities. Geldenhuys et al. [12] have provided an approach to calculate the probabilities of code executions quantitatively. This work can be seen as a form of profiling. In the context of weakest preconditions, McIver and Morgan have several work in studying probabilities and nondeterminism in programs, e.g., in [28]. In our work, the probability of transition is known as a priori.

Our framework has been applied to reliability prediction and distribution. For reliability prediction, it is related to software architecture based reliability evaluation [7, 18, 14, 13]. Compared to the above work, our approach handles systems with model parameters which are hard to obtain. Furthermore, it can quantify the accuracy of component reliability with the help of hypothesis testing. Some recent studies focus on dynamically changing parameters in reliability models and updating parameters based on run-time data [27, 11, 10]. These are not applicable until the software is released. Our reliability model tackles the issue on missing run-time information before system deployment. In addition, our remedy relies on modeling hard-to-predict run-time behaviors as nondeterministic choices so as to obtain reliability measurement which is independent of the dynamic environment. Our reliability distribution problem is similar but slightly different from the reliability allocation problem by solving an optimization problem, e.g., in [31, 17, 26]. The optimization goals are to minimize the amount of testing time while ensuring that a system is sufficiently reliable. [26] also discusses a way to minimize the number of remaining faults given a fixed amount of testing efforts. Our method on reliability distribution focuses on the minimization of component reliability requirement. To the best of authors' knowledge, our work is the first on applying the combination of probabilistic model checking with hypothesis testing to reliability prediction and distribution. Moreover, we have established the system error bounds from components error bounds and vice versa.

In future work, we will extend our combined setting to the verification of general properties, which can support quantitative measurement like "the probability of warning messages failing to send before failure occurs is at least 0.99". This is related to the decomposition of global property into local properties, which is complicated in general. We are currently studying conditions under which such decomposition is sound (i.e., if the components satisfy the local properties, then the global property is guaranteed) or complete or both. A possible approach is to start with coarse sound decomposition and refine the decomposition through a method similar to counter-example guided abstraction refinement.

7. ACKNOWLEDGEMENT

This work has been supported in part by research grant ZJURP1100105 and IDD11100102 (SUTD).

8. REFERENCES

[1] RaPiD. `http://www.comp.nus.edu.sg/~pat/rel`.
[2] M. Avriel. *Nonlinear Programming: Analysis and Methods*. Dover Publishing, 2003.
[3] A.Wald. Sequential tests of statistical hypotheses. *Annal of mathematical statistics 16*, 2:117–186, 1945.

[4] A.Wald. *Sequential Analysis*. Wiley, 1947.

[5] C. Baier and J. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[6] J. Bogdoll, L. M. F. Fioriti, A. Hartmanns, and H. Hermanns. Partial order methods for statistical model checking and simulation. In *Formal Techniques for Distributed Systems*, pages 59–74. Springer, 2011.

[7] R. C. Cheung. A user-oriented software reliability model. *IEEE Trans. Software Engineering*, SE-6(2):118–125, 1980.

[8] E. Clarke, A. Donzé, and A. Legay. Statistical model checking of mixed-analog circuits with an application to a third order δ- σ modulator. In *Hardware and Software: Verification and Testing*, pages 149–163. Springer, 2009.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[10] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE*, pages 111–121.

[11] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *ICSE*, pages 341–350. ACM, 2011.

[12] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *ISSTA*, pages 166–176. ACM, 2012.

[13] S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Trans. Dependable and Secure Computing*, 4(1):32–40, 2007.

[14] K. Goševa-Popstojanova and K. S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2-3):179–204, 2001.

[15] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. In *Formal Aspects of Computing 6(5)*, pages 512–535, 1994.

[16] D. Henriques, J. G. Martins, P. Zuliani, A. Platzer, and E. M. Clarke. Statistical model checking for markov decision processes. In *QEST*, pages 84–93. IEEE, 2012.

[17] C. Y. Huang and M. R. Lyu. Optimal testing resource allocation, and sensitivity analysis in software development. *IEEE Trans. Reliability*, 54(4):592–603, 2005.

[18] A. Immonen and E. Niemel. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008.

[19] Z. Jelinski and P. Moranda. Software reliability research. *Freiberger, W.(ed.): Statistical Computer Performance Evaluation*, pages 465 – 484, 1972.

[20] P. Kubat. Assessing reliability of modular software. *Operations Research Letters*, 8(1):35–41, 1989.

[21] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, pages 585–591, 2011.

[22] J. C. Laprie and K. Kanoun. *Handbook of software Reliability Enginerring*, chapter Software Reliability and System Reliability, pages 27–69. McGraw-Hill, New York, NY, 1996.

[23] R. Lassaigne and S. Peyronnet. Approximate planning and verification for large markov decision processes. In *SAC*, pages 1314–1319. ACM, 2012.

[24] A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *RV*, pages 122–135, 2010.

[25] B. Littlewood and J. L. Verrall. A bayesian reliability growth model for computer science. *Journal of the Royal Statistical Society, Ser. A (Applied Statistics)*, pages 332 – 346, 1973.

[26] M. R. Lyu, S. Rangarajan, and A. P. A. van Moorsel. Optimal allocation of test resources for software reliability growth modeling in software development. *IEEE Trans. Reliability*, 51(2):183–192, 2001.

[27] I. Meedeniya and L. Grunske. An efficient method for architecture-based reliability evaluation for evolving systems with changing parameters. In *ISSRE*, pages 229–238.

[28] C. Morgan and A. McIver. pgcl: Formal reasoning for random algorithms. *South African Computer Journal*, pages 14–27, 1999.

[29] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Trans. Software Engineering*, 10(2):14–32, 1993.

[30] J. D. Musa and K. Okumoto. A logarithmic poisson execution time model for software reliability measurement. *Malaiya, Y. K.; Srimani, P. K. (ed.): Software Reliability Models - Theoretical Developments, Evaluation & Applications*, pages 23 – 31, 1990.

[31] R. Pietrantuono, S. Russo, and K. S. Trivedi. Software reliability and testing time allocation: An architecture-based approach. *IEEE Trans. Software Engineering*, 36:323–337, 2010.

[32] A. Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57. IEEE, 1977.

[33] H. Sandoh. Reliability demonstration testing for software. *IEEE Trans. Reliability*, 40(1):117–119, 1991.

[34] K. Sharma, R. Garg, C. K. Nagpal, and R. K. Garg. Selection of optimal software reliability growth models using a distance based approach. *IEEE Trans. Reliability*, 59(2):266–276, 2010.

[35] W. J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[36] J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In *CAV*, pages 709–714. Springer, 2009.

[37] O. Tal, C. McCollin, and T. Bendell. Reliability demonstration for safety-critical systems. *IEEE Trans. Reliability*, 50(2):194–203, 2001.

[38] D. M. Woit. Estimating software reliability with hypothesis testing. Technical report, CRL Report No.263, Monterey, 1996.

[39] H. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon, 2005.

[40] H. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *CAV*, pages 223–235. Springer, 2002.