

SHARP: A Scalable Approach to Architecture-Level Reliability Prediction of Concurrent Systems

Leslie Cheung
Computer Science Dept
Univ of Southern California
lccheung@usc.edu

Leana Golubchik
Computer Science Dept
Univ of Southern California
leana@usc.edu

Nenad Medvidovic
Computer Science Dept
Univ of Southern California
nen@usc.edu

ABSTRACT

Early prediction of reliability is important in building dependable software. Existing approaches are unable to model concurrent systems in a scalable way. To address the scalability challenge, we propose a framework that is applicable at the architecture level. Our framework achieves scalability by approaching the system from the perspective of usage scenarios and by employing a hierarchical solution. Specifically, we solve lower granularity scenario-based submodels and a higher granularity system model; we then combine their results to obtain a system reliability estimate. Our evaluation indicates that (a) the proposed hierarchical framework is accurate, and (b) that it is more scalable than existing techniques.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—Reliability

General Terms

Reliability, Design

Keywords

Scalability, Concurrent Systems, Hierarchical Approach

1. INTRODUCTION

Early prediction of reliability is important in building dependable software systems. Problems discovered in later stages, such as during implementation, can be very costly to address. Major design decisions are made long before the software has been implemented, and correcting these problems may require redeveloping the system. Therefore, it is critical to analyze a software system's reliability in early stages of design, starting with the system's architecture. In this paper, we propose SHARP, a Scalable, Hierarchical, Architecture-level, Reliability Prediction framework.

A number of recent approaches have begun to quantify system reliability by analyzing a system's architectural models [3, 5, 7, 8, 9, 14, 15, 16, 20, 21]. Within the class of such efforts, SHARP is

distinct as it focuses on predicting reliability of *concurrent* software systems in a *scalable* yet accurate manner. The improved scalability is achieved through a *hierarchical approach*. We first overview existing techniques' shortcomings, which motivate our work.

Most existing techniques assume a sequential system. Typically, in such approaches a reliability model keeps track of which component is running. This is inadequate when modeling systems in which many components may be running concurrently. In predicting reliability of concurrent systems, one typically needs to keep track of the status of all components, i.e., a state in the reliability model consists of multiple variables, where the i^{th} variable corresponds to the state of Component i . Such an approach is taken, for instance, in [5, 20], and we refer to it as a "flat model". Such approaches essentially generate reliability models in a brute-force manner, and suffer from scalability (i.e., "state explosion") problems, making them often prohibitively costly to generate and solve.

To address the problem of *scalable* reliability prediction of concurrent systems, in this paper we propose SHARP, a hierarchical reliability prediction framework (see Section 3). That is, we approach the scalability problem by generating and solving submodels that each capture a part of the system's functionality. In doing so, SHARP leverages system use-case scenario models, such as UML sequence diagrams. Specifically, we solve finer granularity scenario-based submodels and a coarser granularity system model; we then combine their results to obtain a system reliability estimate. The motivation here is that the submodels are expected to be relatively small, and that solving a number of smaller submodels (rather than one huge model) results in space and computational savings. We note that the use of scenario-based models is also explored in [8, 15, 21]. However, these works differ from SHARP in that [8, 21] assume a sequential system, while [15] considers a "flat model" and thus suffers from the very scalability problems we are striving to address.

Lastly, we note that SHARP is an approximation of the "flat model" (i.e., one that keeps track of the status of all components) used in other techniques. However, we argue that its potential scalability benefits are achieved at the cost of fairly small losses in its accuracy. We evaluate our approach for accuracy and complexity. We validate the accuracy of SHARP, using the "flat model" as the ground truth. We illustrate that in practice significant space and computational benefits are obtained through our approach.

In summary, our main contribution is an architecture-level, hierarchical framework that can model *concurrent* systems in a *scalable* manner (see Section 3). Our initial assessment (see Section 4) illustrates that the practical space and computational cost benefits are achieved at the cost of small losses in accuracy. We believe that a number of interesting research directions remain open, as detailed in Section 5.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

QUOVADIS '10, May 3 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-972-5/10/05 ...\$10.00.

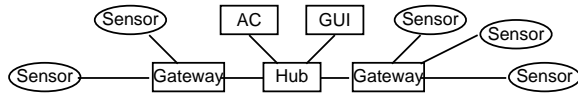


Figure 1: MIDAS' high-level architecture

2. BACKGROUND AND RELATED WORK

2.1 Running Example

For ease of exposition of our framework and for its subsequent evaluation, we use a sensor network application, built using the MIDAS framework [13], as our running example. This system monitors room temperature and turns the air conditioner (AC) on or off in order to satisfy user-specified temperature levels. We refer to this example system as the MIDAS system. The MIDAS system consists of five different types of components: a *Sensor* measures temperature and sends the measured data to a *Gateway*. The *Gateway* aggregates and translates the data and sends it to a *Hub*, which determines whether it should turn the AC on or off. Users can view the current temperature and change the thresholds using a *GUI* component, which then sends an update to the *Hub*.

Figure 1 gives a high-level depiction of our example system's architecture, with the state machines capturing the behavior of the corresponding components given in Figure 2. In a component state machine, an event E is either a sending event or a receiving event. Here we use the notation introduced by Yellin and Strom [22], in which sending and receiving of events is represented by “-” and “+”, respectively. For example, $+E1$ represents a component receiving event $E1$, while $-E2$ represents a component sending event $E2$. Note that some of the state machines include failure states (states labeled by a negative number in Figure 2) which represent erroneous behavior that is triggered by a failure event F . We discuss how we generate failure states below.

There are three possible scenarios in our example MIDAS system: (*Scenario 1*) the *Process Sensor Data* scenario that includes processing measurements from *Sensor* components; (*Scenario 2*) the *GUI Request* scenario that includes updating the temperature readings and changing temperature thresholds; and (*Scenario 3*) the *Control AC* scenario that includes turning AC on or off according to the temperature readings. In the remainder of this paper, for ease of illustration, we focus mainly on Scenario 1, which is depicted using a sequence diagram in Figure 3.

2.2 Background

We focus on analyzing the effect on software system reliability of *architectural defects*, such as signature mismatches and mismatches between interaction protocols of components. When a defect is triggered, a *failure* may occur, in which a component produces a result that is incorrect (as defined in the system's requirements specification). We note that a *recovery* from failure is possible (as explained below). Lastly, we define *system reliability* as the probability that a user does not experience a failure caused by architectural defects.

In system reliability analysis, a system's failure is typically defined in terms of the failures of its components. As detailed in our previous work [2], we analyze an individual system component's architectural model by applying a defect classification technique, e.g., [17], to determine the failure states, and add a transition from a state that may trigger an identified defect to a failure state. Using the technique in [17] on the MIDAS system, we identify a defect in the *Sensor* component: a *Sensor* is unable to notify the *Gateway* when it is running out of battery due to a mismatch between the components' interaction protocols. Failures caused by this defect

are represented as the failure state -1 in the model in Figure 2(a). We extend the event send/receive nomenclature to failure and recovery events by viewing a component as sending failure (recovery) events when it fails (recovers).¹

Further details on the generation of component reliability models can be found in [2], where we discuss how we leverage information available at the architectural level to estimate the model's parameters (transition rates, failure rates, and recovery rates).

2.3 Related Work

Current literature includes a number of software reliability prediction techniques that are applicable at the architectural level [3, 5, 7, 8, 9, 14, 15, 16, 20, 21]. A comprehensive treatment of these is given in three surveys on the topic [6, 10, 11]. Many of these approaches are influenced by [3], which is one of the earliest works on reliability prediction that considers a system's internal structure using Markov chains. In [3], the states in the reliability model represent components, while the transitions represent transfer of control between components. These transitions are assumed to follow the Markov property (i.e., a transition to the next state is determined only by the current state). The work in [3] assumes a sequential system, and most existing approaches, with the notable exception of [5, 15, 20], make the same assumption. Since our work focuses on *concurrent* systems, we restrict the remaining discussion mostly to works that address concurrency. We also comment on approaches that make use of scenario-based models, as well as approaches based on formalisms other than Markov chains.

As noted earlier, in modeling a concurrent system one typically needs to keep track of the status of all components. [5, 15, 20] have taken this approach, in which a state S in a model of a concurrent system is described by C variables, where C is the number of components in the system, i.e., $S = (S^1, S^2, \dots, S^C)$. In [5, 20], components are modeled as black-boxes, which are either active or idle, i.e., $S^i = 0$ when Component i is idle and $S^i = 1$ when Component i is active. In addition to scalability problems, this is also a shortcoming since representing the internal structure of components facilitates more accurate models. For example, some defects may only be triggered when the component performs certain functions. To address this, instead of modeling the status of a component as either active or idle, one can use a finer-granularity component model; this would result in the type of model used in [15], where S^i represents the state of Component i . Specifically, [15] generates component models from scenario models and then generates a system model by combining the component models using the parallel composition technique developed in [4]. We note that, as in [15], our work models systems at a finer granularity level through the use of scenarios. However, unlike [15], we do not need to generate and solve an entire system model; instead, we employ a hierarchical approach, which is intended to yield better scalability.

Existing approaches are also inflexible with respect to different notions of system failure, e.g., in [3, 8, 9, 14, 20], failures are represented by transitions to a failure state in the (Markov-chain based) reliability model. In these models (which assume a single-threaded system), being in state S_i indicates that Component i is active, while all other components are idle. A transition from a state S_i to a failure state indicates that Component i has failed. This means that if any active component has failed, the entire system is considered to have failed. This is also the case in [15], where the system transitions to a failure state when any active compo-

¹We assume that the failures are recoverable, but we can model irrecoverable failures by considering transient analysis without significantly changing SHARP. For brevity, we omit details of irrecoverable failure analysis.

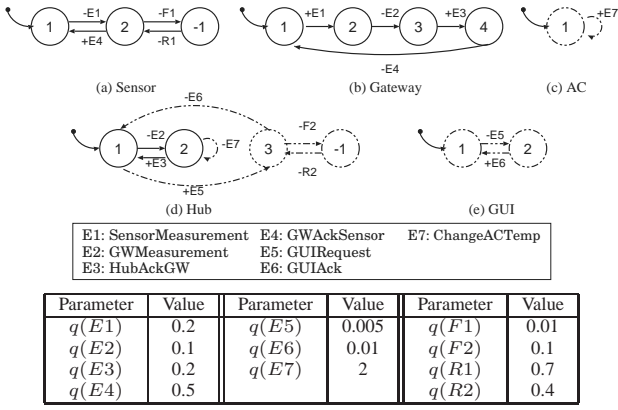


Figure 2: Components' state diagrams

nent fails. The work in [5, 7] does not include failure states explicitly; rather essentially a reward is assigned to each state (with the value of the reward representing the probability of the system failing in that state), where the system's reliability is computed as a Markov reward function [18]. However, the system failure description is still limited, assuming that the system fails when any (active) component fails. [20] provides a somewhat richer description of system failures, where a reliability model includes backup components that can provide services when the primary component fails; the system fails when the primary component and all backup components fail. However, this approach is not capable (without significant changes) of describing other notions of system failure, e.g., an OR-type relationship (the system fails when Component A or Component B fails). Such notions of system failure can be described within SHARP.

Some existing approaches make use of scenario models [8, 15, 21], but they assume a sequential system, with the exception of [15] as described above. For example, in [21] system reliability is defined as the weighted sum of scenario reliabilities. The weights represent the probabilities that each scenario occurs, with the assumption that one scenario is active at a time. This is not the case in our work: in a concurrent system, it is possible to have concurrency within a scenario, as well as multiple scenarios and/or multiple instances of the same scenario running simultaneously. Moreover, [21] assumes that the probabilities of each scenario occurring are known, which is also not the case in our work.

Other state-based approaches, such as those based on stochastic Petri nets (SPNs), suffer from the same state explosion problem. Existing SPN-based approaches focus on performance analysis based on UML models (see [1] for a survey); such models can be used in reliability analysis as well. However, solving the SPN requires generating the SPN's reachability graph, which has the same state explosion problem described above.

While non-state-based approaches, such as [12, 16], may not have the state explosion problem and (implicitly) consider concurrency, they are not as descriptive as state-based approaches, and hence may not give accurate estimates. For instance, the work in [12] computes system reliability as a weighted average of the reliabilities of all execution paths, and the reliability of each path is the product of component reliabilities. In addition, our own previous work [16] explored the use of Bayesian Networks (BNs) to model reliability of concurrent systems. States in the component models are interpreted as nodes in a BN, and transitions are interpreted as dependencies between nodes. We solve the BN for reliability given these dependencies and component reliabilities. However, the notion of concurrency in these approaches is limited as they do not describe flow of control. For example, the two approaches above

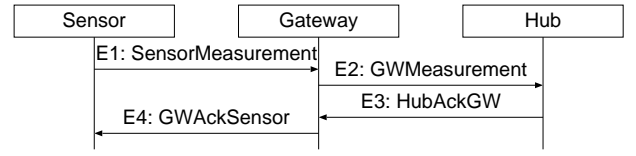


Figure 3: Sequence diagram of Scenario 1

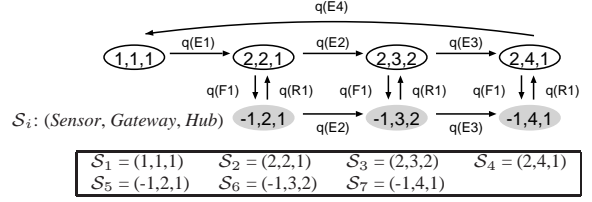


Figure 4: Scenario-level model of Scenario 1

are not able to model the time spent in each component, so that a lightly-used component has the same effect on reliability as a heavily-used component.

3. THE PROPOSED FRAMEWORK

To address scalability problems, we constructed SHARP as a hierarchical technique. At a lower level, we use finer-grained submodels, each representing a subset of system functionality. We refer to these as "scenario-level" models (for reasons made clearer below). At a higher level, we model concurrency aspects of the system using coarse-grained models, which only capture what is running in the system and not the corresponding details. We refer to these as "concurrency-level" models. This approach is motivated by the fact that it is typically more efficient to solve many small submodels rather than one large model.

In particular, we generate the submodels using the system's components and use-case scenarios, which provide a meaningful way to divide the system into smaller parts. We build scenario-level models (as detailed in Section 3.1) that each capture a subset of the system's functionality. These scenario-level models are evaluated for their reliability and performance characteristics. The performance characteristics are needed in building the concurrency-level models (as detailed in Sections 3.1 and 3.2), while the reliability characteristics are needed in computing the reliability of combinations of system scenarios (as detailed in Section 3.3) based on the system designer's definition of system failure. After solving the scenario-level models for scenario reliabilities, we need a way of combining the results appropriately to compute overall system reliability (as detailed in Section 3.4).

We model concurrency in SHARP as simultaneously executing instances of scenarios, in which each instance can be considered as a thread in the system. As an example, in MIDAS, a user may interact with the *GUI* while two sensors are taking measurements and interacting with other components. This can be represented by having one instance of the *GUIRequest* scenario and two instances of the *SensorMeasurement* scenario running simultaneously.

3.1 Step 1: Scenario-Level Models

3.1.1 Building scenario-level models

When building a scenario-level model, we extract the states and transitions in the involved components' state machine models that correspond to events in the scenario of interest. For example, in Scenario 1 from Figure 3, only the following events may occur: $E1$, $E2$, $E3$, and $E4$. Consider the *Hub* state machine in Figure 2(d); the events $E5$, $E6$, and $E7$ do not occur in Scenario 1, hence we eliminate the states and transitions associated with these events (marked

as dashed lines in Figure 2). The *GUI* and *AC* components do not participate in Scenario 1 at all, hence they are not used in the generation of the Scenario 1 model.

We then include all states in the state machine models which are not eliminated above. Moreover, if the state machine models contain transitions to a failure state from one of the included states, then that failure state is included as well — e.g., the failure state in the *Sensor* state machine is included because its State 2 is included.

Once all component models for a particular scenario are derived, we build the scenario-level model by applying the parallel composition technique developed in [4]. In our example, this would result in the model depicted in Figure 4. States corresponding to scenario failures are determined according to the system’s definition of failure. In the simplest case, a state in a scenario-level model is a failure state if any component is in a failure state.² Failure states in Scenario 1 model are marked in grey in Figure 4.

Once we have determined the states and transitions, we need to determine the rate at which each of those transitions occurs. Let us define the rate at which event E occurs to be $q(E)$. The transition rates may correspond to user behavior (e.g., a user issues a *GUIRequest* every X minutes on the average), system requirements (e.g., a *Sensor* sends a measurement to a *Gateway* every X seconds), predictions of system performance (e.g., on the average, a *Hub* will take X milliseconds to process data from a *Gateway*), and so on. As explored in our prior work [2], system architects can estimate the rates from functionally similar components (e.g., an older version of the same component), from requirements documents, from expert knowledge, or from a combination of these sources of information. Note that these sources of information may be unreliable, i.e., they may be poor estimates of the system’s actual usage patterns. In [2], we provide an extensive analysis of a reliability model’s sensitivity to them. We leverage these results in the remainder of this paper and assume that transition rates are included in the component models. Transition rates associated with the same event are assumed to be the same among all components. For example, in both *Sensor*’s and *Gateway*’s component models, $E1$ is assumed to occur with rate $q(E1)$.

Once the states, the transitions, and the corresponding rates are determined, we can represent a scenario-level model by a continuous time Markov chain (CTMC)³ with a corresponding transition rate matrix, \mathcal{Q} , where the $(i, j)^{th}$ entry represents the transition rate from state \mathcal{S}_i to state \mathcal{S}_j in the scenario-level model. Note that since we are using a CTMC model, all the diagonal entries in \mathcal{Q} must be set such that each row sums to 0 [18]. For instance, using the parameters in Figure 2, the transition rate matrix, \mathcal{Q} , corresponding to MIDAS’s Scenario 1 model, is given in Equation 1.

$$\begin{matrix} \mathcal{S}_1 & \mathcal{S}_2 & \mathcal{S}_3 & \mathcal{S}_4 & \mathcal{S}_5 & \mathcal{S}_6 & \mathcal{S}_7 \\ \begin{matrix} \mathcal{S}_1 \\ \mathcal{S}_2 \\ \mathcal{S}_3 \\ \mathcal{S}_4 \\ \mathcal{S}_5 \\ \mathcal{S}_6 \\ \mathcal{S}_7 \end{matrix} \end{matrix} \begin{pmatrix} -0.2 & 0.2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.11 & 0.1 & 0 & 0.01 & 0 & 0 \\ 0 & 0 & -0.21 & 0.2 & 0 & 0.01 & 0 \\ 0.5 & 0 & 0 & -0.51 & 0 & 0 & 0.01 \\ 0 & 0.7 & 0 & 0 & -0.8 & 0.1 & 0 \\ 0 & 0 & 0.7 & 0 & 0 & -0.8 & 0.1 \\ 0 & 0 & 0 & 0.7 & 0 & 0 & -0.7 \end{pmatrix} \quad (1)$$

3.1.2 Scenario Reliability

Given the scenario-level model constructed above, we can now compute that scenario’s reliability. We define reliability of a sce-

²SHARP is flexible enough to allow designers to specify more complex failure rules. This is done using the same technique at the scenario level as described in Section 3.3 at the system level.

³The choice of CTMC is in no way constraining, as it can be mathematically converted to a discrete-time Markov chain (DTMC), as long as the rates are bounded [18].

nario as the probability that it is not in any of the failure states, given that the scenario is active (i.e., it is not in the idle state). This involves solving the CTMC for the steady state probability of being in state \mathcal{S}_i , $\pi(\mathcal{S}_i)$, using standard techniques [18]. We can then compute the reliability of Scenario i as:

$$r_i = \frac{1 - \sum_i \pi(\mathcal{F}_i) - \pi(\mathcal{S}_1)}{1 - \pi(\mathcal{S}_1)} \quad (2)$$

where \mathcal{F}_i is a failure state of the scenario-level model.

In our MIDAS example, solution of the Scenario 1 model from Figure 4 results in Scenario 1’s reliability being 0.9859. When applying the same approach to the generation and solution of the other scenario-level models in MIDAS, the reliabilities of Scenarios 2 and 3 are determined to be 0.8 and 1, respectively.

3.1.3 Scenario Completion Rates

An important ingredient in building a concurrency-level model (described next in Section 3.2) is the determination of a scenario’s average completion time, T , which can be defined as the average time it takes for a scenario-level model to return to the idle state \mathcal{S}_1 .⁴ To compute T , we use the same scenario-level model as in Section 3.1.2, but this time use it in computing a performance metric (namely, the scenario completion time).

Let us define the average scenario completion rate μ as $\mu = \frac{1}{T}$. We note that T includes time spent in normal operation as well as time spent in recovering from failures, because the definition of an “active” scenario (in the concurrency level model introduced below) includes all of the scenario’s behavior (i.e., everything other than the idle state \mathcal{S}_1).

Let $T(\mathcal{S}_i)$ be the scenario completion time, conditioned on the fact that the scenario begins its operation in \mathcal{S}_i , the first state to which it transitions from the idle state. Since there may be more than one transition out of the idle state, we need to first find $T(\mathcal{S}_i)$ for all \mathcal{S}_i . We can then uncondition on \mathcal{S}_i , to compute T . We compute $T(\mathcal{S}_i)$ for all \mathcal{S}_i by performing transient analysis that corresponds to solving Equation 3 [19]:

$$-v(\mathcal{S}_i)T(\mathcal{S}_i) + \sum_{\substack{\mathcal{S}_j \neq \mathcal{S}_1 \\ \mathcal{S}_j \neq \mathcal{S}_i}} \mathcal{Q}(\mathcal{S}_i, \mathcal{S}_j)T(\mathcal{S}_j) = -1 \quad (3)$$

where $v(\mathcal{S}_i)$ is the sum of the rates corresponding to transitions out of state \mathcal{S}_i . i.e.,

$$v(\mathcal{S}_i) = \sum_{\mathcal{S}_j \neq \mathcal{S}_i} \mathcal{Q}(\mathcal{S}_i, \mathcal{S}_j) \quad (4)$$

Let $P(\mathcal{S}_1, \mathcal{S}_i)$ be the probability of going from \mathcal{S}_1 to \mathcal{S}_i . Then,

$$T = \sum_i T(\mathcal{S}_i)P(\mathcal{S}_1, \mathcal{S}_i) = \sum_i T(\mathcal{S}_i) \frac{\mathcal{Q}(\mathcal{S}_1, \mathcal{S}_i)}{v(\mathcal{S}_1)} \quad (5)$$

Applying Equations 3 and 5 to the matrix in Equation 1 describing the Scenario 1 model, we compute the average completion time of Scenario 1 to be 17.0358 time units. We then compute the average completion rate of Scenario 1 as $\mu = \frac{1}{T} = 0.0587$. We compute the completion rates of the other scenarios in our example system using the same method: the completion rates of Scenarios 2 and 3 are 0.08 and 0.002, respectively.

3.2 Step 2: Concurrency-Level Model

Next, we describe how to generate a concurrency-level model. We do this by keeping track of the number of active instances of

⁴The scenario completion time has a direct effect on the mix of possible scenarios that can run in the system simultaneously. This will become clearer in Section 3.2.

Table 1: Values of $P_i(A_i)$ in MIDAS

Parameter	Value	Parameter	Value	Parameter	Value
$P_1(0)$	0.0514	$P_2(0)$	0.9412	$P_3(0)$	0.9804
$P_1(1)$	0.3507	$P_2(1)$	0.0588	$P_3(1)$	0.0196
$P_1(2)$	0.5979				

each scenario in a system. Recall that we model concurrency as simultaneously executing instances of scenarios, and an instance is considered active when it is in any state other than the idle state of its scenario-level model. For example, consider Scenario 1 in Figure 3. Before a *Sensor* sends *E1*, this instance of Scenario 1 is considered idle, as the execution of Scenario 1 has not started in any component. The components could, however, be executing other scenarios. Once the *Sensor* has sent *E1*, this instance is considered active, until the *Sensor* has received *E4*, in which case all three components have finished the execution of the scenario.

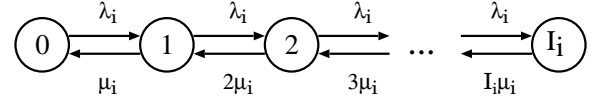
The state of the entire system can be described as a *combination of scenarios*. A combination of scenarios, C_j , is defined as $C_j = (A_1, A_2, \dots, A_S)$, where A_i represents the number of active instances of Scenario i ; here S is the total number of distinct scenarios in the system.⁵ Since it may not be realistic for the number of instances of a particular scenario to grow unbounded, we also define I_i to be the maximum possible number of active instances of Scenario i , and $I = \max(I_i)$ be the largest number of active scenario instances for all Scenarios i . We envision I_i being available at design-time, as such a limit is likely to be due to design decisions, system resource limitations, requirements, the system’s structure, and the components’ behaviors. In order to complete our reliability estimation (Sections 3.3 and 3.4), we need to compute the distribution of the possible active scenario combinations.

Since, in general, not all combinations of scenarios in a system may be possible, we allow a system architect to specify the combinations of scenarios that are not possible (or not allowed). For instance, in MIDAS, such a restriction may be put in place to avoid a race condition: when a user is setting a new threshold through the *GUI* (Scenario 2), the *Hub* may incorrectly control the *AC* using the old threshold (Scenario 3). In that case, an architect would specify that Scenarios 2 and 3 cannot run simultaneously. Hence, if we set $I_1 = 2$, $I_2 = 1$, and $I_3 = 1$, and include the restriction that Scenarios 2 and 3 cannot run simultaneously, then, the possible scenario combinations are those depicted in Table 2.

After we have determined the possible scenario combinations, we calculate the probability that each combination occurs. We define $P(C_j) = P(A_1, A_2, \dots, A_S)$ to be the probability of having A_i active instances of Scenario i for each i . To reduce the complexity of this computation we make the simplifying assumption that in a highly concurrent system, all instances of all scenarios run independently. If executions of two or more scenarios are dependent on one another, we treat them as a single, larger scenario. Intuitively, this assumption corresponds to the rates of all transitions in the scenario-level models being independent of A_i . We make this assumption to reduce the computational cost of this step, recognizing that it may not always hold in practice. Thus, our solution is an approximation, and we illustrate its accuracy in Section 4.2. Hence,

$$P(C_j) \simeq \frac{\prod_i P_i(A_i)}{W} \quad (6)$$

⁵Given that the system essentially experiences scenario “start” and “completion” events, we assume that the probability that more than one scenario starts and/or completes in the exact same instant in time is negligible. This is a standard assumption in Markov chain models which makes them more tractable without a significant loss in what is expressible with such models.

**Figure 5: A concurrency-level model**

where $W = \sum_i P(C_j)$ is a normalization factor⁶ that ensures that $P(C_j)$ sum to 1. In MIDAS, $P(A_1, A_2, A_3)$ is the probability of having A_1 , A_2 , and A_3 active instances of Scenarios 1, 2 and 3, respectively. According to Equation 6,

$$P(A_1, A_2, A_3) \simeq \frac{P_1(A_1) \times P_2(A_2) \times P_3(A_3)}{W}$$

To compute $P_i(A_i)$, we solve a concurrency-level model, depicted in Figure 5, for each Scenario i . Specifically, this is a CTMC model, representing the number of active instances of Scenario i . When there are A_i instances of Scenario i , $A_i < I_i$, a new instance of Scenario i starts at the rate of λ_i . Note that the starting of a new scenario instance corresponds to transitioning out of the idle state in the scenario-level model. Therefore, $\lambda_i = v(S_1)$, where S_1 is the idle state of Scenario i ’s model (recall Equation 4). When $A_i > 0$, an instance of Scenario i completes at the rate of $A_i \mu_i$, where μ_i , the scenario completion rate, is computed as described in Section 3.1.3.

Now that we have determined the transition rates of the (Markov chain) concurrency-level model, we can solve the model for $P_i(A_i)$ for all A_i , using standard techniques [18], which corresponds to solving for the probability of being in state A_i in Scenario i ’s concurrency model (recall Figure 5).

Table 1 gives the probability distribution of the number of active instances for each scenario in our MIDAS example; these are computed using concurrency-level models of each scenario (as described above). Table 2 gives the corresponding combination probabilities, computed using the data from Table 1 by applying Equation 6. Lastly, since the computation of the distribution of different scenario combinations is done in an approximate manner as described above, in Section 4 we evaluate the accuracy of this approximation, as well as the reduction in computational cost.

3.3 Step 3: Combining Results

Given that we now know how to compute the probabilities of having various combinations of scenarios as well as the reliabilities of individual scenarios, what remains is the computation of the reliabilities of the scenarios’ combinations (described next) followed by the computation of the overall system reliability (described in Section 3.4). We will use the combination (2,1,0) — two active instances of Scenario 1, one active instance of Scenario 2, and zero active instances of Scenario 3 — as an illustrative example in this section; the reliabilities of other combinations are calculated analogously. To compute the reliability of a scenario combination, we need to first examine how system failure is defined.

In SHARP, system designers can specify the conditions under which the system is considered to have failed as follows. If there are x_i or more failed instances of *any* Scenario i , the system is considered to have failed, i.e.,

$$(F_1 \geq x_1) \vee (F_2 \geq x_2) \vee \dots \vee (F_S \geq x_S) \quad (7)$$

where F_i is the number of failed instances of Scenario i (recall that S is the number of distinct scenarios).⁷As an example, in MIDAS,

⁶The normalization factor is needed because, in general, not all combinations of scenarios may be allowed, as described above.

⁷The OR-clauses are used for ease of presentation. SHARP can easily specify more general failure conditions, by using disjunctive normal form and modifying Equation 10 accordingly.

Table 2: Values of $P(C_j)$ and $R(C_j)$ in MIDAS

C_j	$P(C_j)$	$R(C_j)$	C_j	$P(C_j)$	$R(C_j)$
(0,0,0)	0.0475	1	(1,1,0)	0.0202	0.7887
(1,0,0)	0.3240	0.9859	(1,0,1)	0.0065	0.9859
(0,1,0)	0.0030	0.8	(2,1,0)	0.0345	0.7998
(0,0,1)	0.0010	1	(2,0,1)	0.0110	0.9998
(2,0,0)	0.5523	0.9998			

the system is considered reliable if it can control the temperature appropriately and display the current room temperature to the user. This requires that (1) at least one sensor correctly measures and sends data to the *Hub*, via a *Gateway*; (2) the *GUI* displays the current temperature obtained from the *Hub*; and (3) the *Hub* controls the *AC* appropriately. Therefore, we define the system to be unreliable when two or more instances of Scenario 1, or one or more instance of Scenario 2, or one or more instance of Scenario 3 have failed, respectively. Thus, $x_1 = 2$, $x_2 = 1$, and $x_3 = 1$.

To compute the probability that combination C_j satisfies the failure condition in Equation 7, we consider the clauses one at a time, and compute the probability of satisfying a clause as follows:

$$P(F_i \geq x_i) = \sum_{f=x_i}^{A_i} P(F_i = f) \quad (8)$$

Note that F_i is a binomial random variable [18]: F_i is the number of failed instances of Scenario i , out of A_i active instances, and an instance is either failed (with probability $1 - r_i$, where r_i is the reliability of Scenario i as defined in Equation 2), or non-failed (with probability r_i). Therefore, according to [18],

$$P(F_i = f) = \binom{A_i}{f} (1 - r_i)^f (r_i)^{(A_i - f)} \quad (9)$$

In the MIDAS example, based on Equations 8 and 9, we compute $P(F_1 \geq 2)$ as follows:

$$\begin{aligned} P(F_1 \geq 2) &= \sum_{f=2}^2 P(F_i = f) \\ &= \binom{2}{2} (1 - 0.9859)^2 (0.9859)^0 = 1.9881 \times 10^{-4} \end{aligned}$$

Similarly, $P(F_2 \geq 1) = 0.2$, and $P(F_3 \geq 1) = 0$.

Since the system is considered to have failed when any clause in Equation 7 is satisfied, the reliability of a combination, $R(C_j)$, can be defined as:

$$R(C_j) = 1 - \sum_{i=1}^S P(F_i \geq x_i) \quad (10)$$

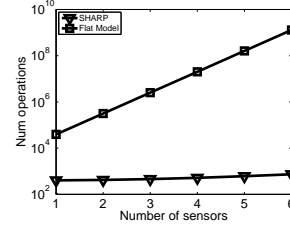
To complete our example, we combine the above results according to Equation 10, i.e.,

$$\begin{aligned} R((2, 1, 0)) &= 1 - \sum_{i=1}^S P(F_i \geq x_i) \\ &= 1 - (1.9881 \times 10^{-4} + 0.2 + 0) = 0.7998 \end{aligned}$$

We repeat this calculation for each combination; Table 2 gives reliabilities of all scenario combinations for the MIDAS example under the above given failure conditions.

3.4 Step 4: Computing System Reliability

As a final step in our framework, we compute system reliability by combining the results of the previous steps. System reliability, $SysRel$, is defined as the sum of the scenario combinations' reliabilities (as computed in Section 3.3), weighted by the probability that the combination occurs (as computed in Section 3.2), i.e.,

**Figure 6: Computational Cost in Practice**

$$SysRel = \sum_j P(C_j)R(C_j) \quad (11)$$

In our running example, the solution of Equation 11 gives the reliability of the MIDAS system as 0.9835, which, in this case, is within 1% of the ground truth of 0.9925, obtained by solving the “flat model” as detailed below.

4. EVALUATION

We evaluate SHARP along two dimensions: (1) the computational cost of generating and solving concurrent systems' reliability models as compared to those that can be derived from existing approaches (Section 4.1), and (2) the corresponding accuracy of SHARP (Section 4.2). More specifically, we compare SHARP against a *flat model*, which is used here as the “ground-truth”. Our flat model is essentially the same as [15], where a system reliability model is generated by applying the parallel composition technique developed in [4].⁸ We note that the flat model does not make the independence assumption in Section 3.2 as SHARP does. Thus, comparison to the flat model also allows us to evaluate the inaccuracy due to this assumption. We also note that the difference between our application of [4] (in Section 3.1) and that in [15] is that we use the technique from [4] to generate a scenario-level model (which, as argued below, is expected to be relatively small) while [15] uses it to generate a model of the entire system at once.

To also establish whether a much simpler and more efficient approximation than the one performed in SHARP would suffice, we additionally compare SHARP against a *coarse approximation* in which the system reliability is computed simply as a product of scenario reliabilities. Specifically, in this coarse approach we solve for scenario reliabilities as in Section 3.1. Then, instead of building and solving concurrency models as in Section 3.2, we assume all scenario instances run all the time, i.e., $SysRel = \prod_i (r_i)^{A_i}$.

4.1 Computational Cost

We now explore the computational cost of SHARP as compared to the flat model and the coarse approximation discussed above. Due to lack of space, we omit the worst-case analysis and instead focus on the computational cost in practice.

Figure 6 illustrates the computational costs in practice of SHARP and the flat model using the MIDAS example. Here, we vary the number of *Sensors* in the system (x-axis), and plot the number of addition/multiplication operations needed to solve the two resulting models on the y-axis. Otherwise, the system is the same as the example used throughout the paper (recall Figure 1). Note that the y-axis of Figure 6 is plotted on a logarithmic-scale. As can be seen from the figure, the computational cost of SHARP is much lower and grows significantly slower than that of the flat model. For example, it takes more than 10^{10} operations to compute the

⁸The only differences between our flat model and the one in [15] is that [15] assumes that failures are irrecoverable. SHARP can model systems with recoverable and irrecoverable failures.

reliability solution of the MIDAS system with 6 *Sensors* using the flat model, while it only takes about 1000 operations to compute the solution using SHARP.

Since the scenario-level models are likely to be smaller than the flat model, we argue that SHARP requires significantly less space in practice than the flat model. The savings are also due to the fact that we can generate and solve scenario-level models one at a time, and thus reuse the space.

Lastly, given that SHARP takes the approach of solving many smaller models rather than one large model, if parallel processing is available, we could solve our models in parallel.

4.2 Accuracy

Our goal is to provide evidence that SHARP is sufficiently accurate to be used in making design decisions. The goal of design-time approaches is to analyze the effect of different design decisions on reliability rather than obtain absolute reliability measurements.⁹ Therefore, we compare the *sensitivities* of SHARP and the corresponding flat model: if the differences in the changes in reliability estimates are reasonable small when the same parameter is varied in both SHARP and the flat model, then SHARP can be considered accurate. For completeness, we also include the coarse approximation results in our analysis.

To assess its sensitivity, we applied SHARP to a large variety of systems, with different numbers of components, scenarios, as well as numbers of instances of scenarios. We show representative results obtained from the following systems:

1. The MIDAS example system we used throughout the paper. This system has three scenarios, and may potentially have a large number of instances of a scenario (e.g., multiple sensors taking measurements). There are four *Sensors*, one *Gateway*, one *Hub*, one *GUI*, and one *AC* in the instantiation of MIDAS used in this evaluation.
2. A GPS system with route guidance, audio player, and bluetooth phone capabilities. This system has five major components: *RouteGuidance (RG)*, *EnergyMonitor (EM)*, *MediaPlayer (MP)*, *BluetoothPhone (BT)*, and *Database (DB)*. This system is modeled using 21 scenarios. Note that it is unlikely that there will be more than one instance of a scenario in this system because of the system's structure (e.g., it typically makes little sense to have two instances of a route guidance scenario to perform the same route guidance service).¹⁰ To evaluate SHARP in a controlled manner, we injected the following defects into this GPS system: (1) a defect in the *EM* component which may lead to failure to notify other system components when the battery is low, and (2) a defect in the *RG* component which may lead to failure in updating a user's location accurately.

First, we compare the sensitivities of the three approaches — SHARP, flat model, and coarse approximation — when model parameters change. We also evaluate the accuracy of SHARP as compared to the ground truth (i.e., flat model). We vary a parameter within a range (to be specified below), and observe how system reliability changes. Here, we present results corresponding to varying failure-related parameters in the MIDAS and GPS systems. We

⁹For example, at implementation time, it may be appropriate to evaluate a system's reliability using the five 9's standard. However, this is not typically meaningful at design time.

¹⁰One exception to this would be the situation when the system designers are concerned about service failures, and hence introduce redundancy. We do not consider such a variant of the GPS system.

performed similar experiments by varying other parameters and using several other systems' models. The results were qualitatively similar and are omitted here for brevity.

In Figures 7(a) - (d), we vary the failure rates of the *Sensor* and *Hub* components in MIDAS between 0 and 0.1, and the *EM* and *RG* components in GPS between 0 and 0.01. In Figures 7(e) - (h), we vary the recovery rates of the *Sensor* and *Hub* components in MIDAS between 0 and 0.5, and the *EM* and *RG* components in GPS between 0 and 0.15. In all cases, we vary the parameters one at a time, maintaining other parameters fixed at their default values (Default values of the MIDAS system are given in Figure 2).

In these experiments, we observe that results obtained from SHARP closely follow the flat model. This suggests that SHARP is accurate in predicting system reliability, while in practice it should result in much better scalability than the flat model approach. While the coarse approximation costs even less (i.e., it corresponds to performing only Step 1 in SHARP), its results exhibit very different trends in several cases (e.g., Figures 7(a), (b) and (e)) and are typically much more pessimistic. This can be seen, for instance, in Figure 7(b), where Scenario 2 triggers the defect in the *Hub*. In this case, the probability that an instance of Scenario 2 is active, $P_2(1)$, is quite low (0.0588 from Table 1). Thus, the coarse approximation's assumption that an instance of Scenario 2 is active all the time gives inaccurate results.

We also illustrate that SHARP is useful in determining which components are more critical to a system's reliability. We have verified this property of SHARP in a number of examples. For instance, in Figure 7, when we vary the failure rates of *Sensor* (Figure 7(a)) and *Hub* (Figure 7(b)) between 0 and 0.1, system reliabilities obtained from SHARP change by 40% and 10%, respectively. Since the system's reliability is affected more by the changes in *Sensor*'s failure rate than *Hub*'s, under these conditions *Sensor* is the more critical component. Note that the differences in the changes in reliability estimates between SHARP and the flat model are very small (within a few percent). We have not observed significant deviations from the flat model in any of our studies. We can thus conclude that SHARP is useful in this analysis.

5. FUTURE DIRECTIONS

A number of interesting research questions remain, and we highlight three ongoing research directions below.

1. We are relaxing the assumption that scenarios run independently. We note again that we made this assumption for tractability of model solution, rather than because real systems necessarily behave in this manner. Although we illustrated accuracy of our approach, we believe that it can be improved further as follows. One of the aspects of dependency we are exploring is *resource contention* between software components. If more than one caller component makes a request to the same callee component, the callee's service time changes, which in turn changes the scenario completion rates (recall Section 3.1.3). We hypothesize that we can augment the scenario-level models to include resource contention using queueing networks, and that this modification will result in further improvements in the accuracy of SHARP.
2. Another important facet of our future work is further reduction in the complexity of the concurrency-level model solution. We envision *model truncation* techniques [18] as one useful direction. For example, we can restrict our models to only include scenarios that may trigger a defect in a component. Another example of the use of truncation is dropping states (in the concurrency-level model) that are highly

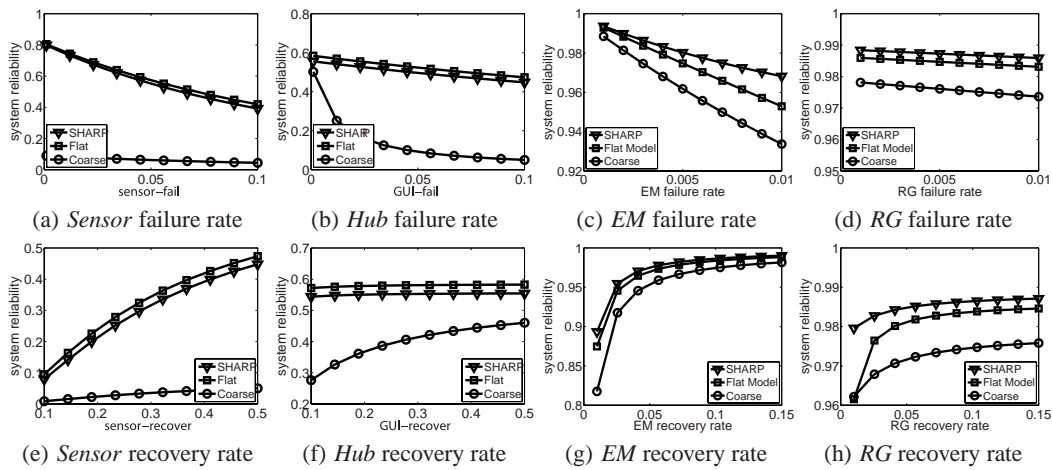


Figure 7: Sensitivity analysis

unlikely to be visited. For instance, states corresponding to many simultaneously active scenario instances can be truncated, if the probability of that occurring is low, i.e., when $\lambda \ll \mu$ (recall Section 3.2). This would reduce the number of scenario-level and concurrency-level models we need to generate, thus reducing the number of scenario combinations. We hypothesize that the resulting penalty in terms of accuracy will be minimal.

3. We are extending our evaluation to more complex systems. More specifically, we plan to apply SHARP on systems with more complex sequence diagrams, and with more interactions among scenarios. Thus far, our experience with applying SHARP to more complex systems indicates that one needs to be careful in how scenario-level models are built, e.g., to insure connectivity.

6. CONCLUSIONS

We presented SHARP, a scalable framework for predicting reliability of concurrent systems. Our main idea in modeling concurrency is to allow multiple instances of system scenarios to run simultaneously. We overcame inherent scalability problems by leveraging scenario models and using an (approximate) hierarchical technique which allowed generation and solution of smaller parts of the overall model at a time. Our experimental evaluation showed that SHARP is more scalable than existing approaches in practice, and its scalability is achieved without significant degradation in the accuracy of system reliability predictions. Lastly, we indicated a number of future research directions.

7. ACKNOWLEDGMENTS

This work is supported by the NSF (award numbers 0417274, 0509539, and 0720612).

8. REFERENCES

- [1] S. Balsamo et al. Model-based performance prediction in software development: A survey. *IEEE TSE*, 30(5), 2004.
- [2] L. Cheung et al. Early prediction of software component reliability. In *ICSE 2008*.
- [3] R.C. Cheung. A user-oriented software reliability model. *IEEE TSE*, 6(2), 1980.
- [4] P. D'Argenio et al. On generative parallel composition. *Electronic Notes in Theoretical Computer Science*, 22, 2000.
- [5] R. El-Kharboutly et al. UML-based methodology for reliability analysis of concurrent software applications. *I. J. Comput. Appl.*, 14(4):250–259, 2007.
- [6] S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE TDSC*, 4(1), Jan 2007.
- [7] S. Gokhale and K. Trivedi. Reliability prediction and sensitivity analysis based on software architecture. *ISSRE'02*.
- [8] K. Goseva-Popstojanova et al. Architectural-level risk analysis using UML. *IEEE TSE*, 29(3), Oct 2003.
- [9] K. Goseva-Popstojanova and S. Kamavaram. Software reliability estimation under uncertainty: Generalization of the method of moments. In *Proc. of HASE 2004*.
- [10] K. Goseva-Popstojanova and K. Trivedi. Architecture-based approaches to software reliability prediction. *Intl. J. Comp. & Math. with Applications*, 46(7), Oct 2003.
- [11] A. Immonen and E. Niemela. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Sys. Modeling*, Jan 2007.
- [12] S. Krishnamurthy and A. Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *ISSRE 97*.
- [13] S. Malek et al. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. *ICSE'07*.
- [14] R. Reussner et al. Reliability prediction for component-based software architectures. *J. of Sys. and Software*, 66(3), 2003.
- [15] G. Rodrigues et al. Using scenarios to predict the reliability of concurrent component-based software systems. *FASE'05*.
- [16] R. Roshandel et al. A Bayesian model for predicting reliability of software systems at the architectural level. *QoSA'07*.
- [17] R. Roshandel et al. Understanding tradeoffs among different architectural modeling approaches. In *WICSA 2004*.
- [18] W. Stewart. *Probability, Markov Chains, Queues, and Simulation*. Princeton University Press, 2009.
- [19] H.C. Tijms. *Stochastic Models*. John Wiley and Sons, 1994.
- [20] W. Wang, D. Pan, and M. Chen. Architecture-based software reliability modeling. *J. of Systems and Software*, 79(1), 2006.
- [21] S. Yacoub et al. Scenario-based reliability analysis of component-based software. In *Proc. 10th ISSRE*, Nov 1999.
- [22] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM TOPLAS*, 19(2):292–333, 1997.