Architecture-Level Reliability Prediction of Concurrent Systems

[†]Leslie Cheung, [*]Ivo Krka, [*]Leana Golubchik, [*]Nenad Medvidovic
[†]NetApp, 495 E Java Dr, Sunnyvale, CA 94089
[*]Computer Science Department, Univ of Southern California, Los Angeles, CA 90089
leslie.cheung@netapp.com, {krka, leana, neno}@usc.edu

ABSTRACT

Stringent requirements on modern software systems dictate evaluation of dependability qualities, such as reliability, as early as possible in a system's life cycle. A primary shortcoming of the existing design-time reliability prediction approaches is their lack of support for modeling and analyzing concurrency in a scalable way. To address the scalability challenge, we propose SHARP, an architecture-level reliability prediction framework that analyzes a hierarchical scenario-based specification of system behavior. It achieves scalability by utilizing the scenario relations embodied in this hierarchy. SHARP first constructs and solves models of basic scenarios, and combines the obtained results based on the defined scenario dependencies; the dependencies we handle are sequential and parallel execution of multiple scenarios. This process iteratively continues through the scenario hierarchy until finally obtaining the system reliability estimate. Our evaluations performed on real-world specifications indicate that SHARP is (a) almost as accurate as a traditional non-hierarchical method, and (b) more scalable than other existing techniques.

Categories and Subject Descriptors

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

Keywords

Scalability, Concurrent Systems, Hierarchical Approach

1. INTRODUCTION

The success of most software systems is directly related to their dependability. One of the challenges in developing dependable software is that problems discovered during system implementation or operation can be prohibitively costly to address [2]: the principal design decisions that critically affect system dependability are made long before [26]. This suggests that analyzing a system's dependability at design time is of critical importance. In this paper, we focus on estimating the reliability of a system under design, and propose SHARP, a Scalable, Hierarchical, Architecture-level, Reliability Prediction framework, which improves upon the current state-of-the-art. To motivate the need for SHARP, we first highlight the shortcomings of the existing approaches.

A number of existing approaches quantify the reliability of a system by analyzing its architectural models (recently surveyed in [8, 12, 13, 15]). These existing techniques generate a stochastic reliability model from software architectural models, and most of them assume a sequential system for which a reliability model only needs to keep track of the currently running component. This is inadequate when modeling realistic systems in which many components are running and communicating concurrently. A straightforward approach to predicting the reliability of a concurrent system is to build a model that keeps track of the internal states of all system components. Such an approach is taken, e.g., in [7, 22, 28]; in this paper we refer to a model produced using such approaches as a "flat model". Flat-model approaches suffer from scalability (i.e., "state explosion") problems: generating and solving the reliability models is prohibitively costly, and even infeasible, with growing system sizes.

By contrast, the framework we propose in this paper, SHARP, explicitly focuses on predicting reliability of concurrent software systems in a scalable and accurate manner. At a high level, we generate continuous-time Markov chain (CTMC)-based reliability models from software architectural models — each describing a different part of the system's behavior. Reliability can then be computed by appropriately combining the results of the models. The improved scalability is achieved through an iterative hierarchical approach that takes advantage of the popular scenario-based behavioral specifications. For example, SHARP can assess reliability of a system whose behavior is described using high-level Message Sequence Charts (hMSCs) [27], Interaction Overview Diagrams (IODs) [29], or UML Sequence Diagrams with fragments [19]. Note that these specification languages allow an engineer to specify how smaller behavioral sequences form more complex behaviors. To ensure the accuracy of the reliability estimates, SHARP takes into account the contention for system resources while combining different parts of a system's behavioral description.

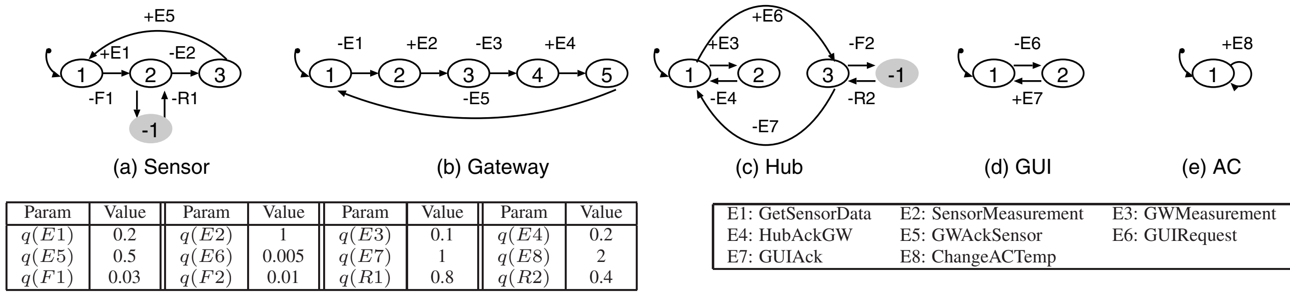
To use SHARP, an engineer needs to provide a system's behavioral specification, help to define the failure states, and specify the operational profile. Our previous work [3, 15] provides an in-depth treatment of the information sources from which these inputs are derivable; note that SHARP does not require more input information than other existing techniques. Based on the inputs, SHARP produces a system reliability estimate, as well as the reliability estimates for the smaller scenario sequences.

The input behavioral specification should consist of a system-level scenario-based specification (e.g., hMSC or IOD) and compo-

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.

ICPE'12, April 22-25, 2012, Boston, Massachusetts, USA

Copyright 2012 ACM 978-1-4503-1202-8/12/04 ...$10.00.



Param	Value	Param	Value	Param	Value	Param	Value
$q(E1)$	0.2	$q(E2)$	1	$q(E3)$	0.1	$q(E4)$	0.2
$q(E5)$	0.5	$q(E6)$	0.005	$q(E7)$	1	$q(E8)$	2
$q(F1)$	0.03	$q(F2)$	0.01	$q(R1)$	0.8	$q(R2)$	0.4

E1: GetSensorData	E2: SensorMeasurement	E3: GWMeasurement
E4: HubAckGW	E5: GWAckSensor	E6: GUIRequest
E7: GUIAck	E8: ChangeACTemp	

Figure 1: Components' state diagrams

nent-level state machine behavior models.[1] The scenario specification has two parts: (a) the description of the desirable event sequences (basic scenarios) and (b) the characterization of how different scenarios relate to one another (complex scenarios). SHARP accounts for different types of scenario dependencies: sequential (scenario A is followed by scenario B), conditional (scenario A is followed by scenario B or scenario C), and concurrent (scenario A and scenario B run concurrently). The expected operational profile should define the frequencies of event occurrence, the probabilities of the conditional scenario continuations, and the number of scenario instances that can be running concurrently. In our work, we identify failure states by analyzing architectural defects using our previously published technique [3, 24]; note that SHARP is not dependent on these specific techniques and any other existing technique can be used to obtain the failure information.

The main idea behind SHARP is that rather than considering a concurrent system as having simultaneously running components (e.g., [22]), we view it as having different *scenarios* executing concurrently. To generate a system model, SHARP first generates and solves reliability models of the basic scenarios, and then incorporates these results into higher-level models according to the defined inter-scenario relationships. We have developed efficient algorithms for (a) estimating the reliability of a complex scenario comprising a sequence of dependent lower-level scenarios and (b) estimating the reliability of a complex scenario comprising multiple concurrently running lower-level scenarios. SHARP iteratively goes through the behavioral specification until finally calculating the reliability of the highest-level scenario. Intuitively, a model of a basic scenario is expected to be relatively small, and solving a number of smaller submodels (rather than one very large, possibly intractable model) while intelligently combining them results in both space and computational savings.

As mentioned above, an important consideration distinguishing SHARP from related approaches is that software components in a concurrent system share and compete for resources such as services provided by other components. To ensure that the reliability estimates are accurate although submodels are analyzed separately, SHARP analyzes the contention between software components and incorporates the results into the reliability models.

We evaluate SHARP's scalability and accuracy on a set of real-world system specifications. Since SHARP is an approximation of the "flat model" used in other techniques, we examine the extent to which SHARP's scalability benefits are achieved at the cost of prediction accuracy. Our results demonstrate that SHARP provides accurate estimates when compared to the "flat model", while significantly reducing computational cost in practice.

In Section 2, we describe a running example used in this paper, and discuss additional background of the presented work. We overview the different parts of the SHARP framework in Section 3 and describe the specifics of the reliability computation algorithms in Section 4. We follow with our evaluations in Section 5, and describe other architecture-based reliability estimation techniques related to SHARP in Section 6. Finally, we conclude in Section 7.

[1]These can be automatically obtained from a scenario specification using existing techniques (e.g., [14, 29]).

2. BACKGROUND

In this section we introduce the running example (Section 2.1), summarize our prior work used to populate the initial software models with reliability related information (Section 2.2).

2.1 Running Example

The running example we use in this paper is a version of MIDAS, a sensor network application [17]. The application monitors the room temperature and controls the air conditioner (AC). MIDAS 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* unit 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*.

The state diagrams depicted in Figure 1 capture the behavior of the MIDAS components. In a component state diagram, an event E is either a sending event or a receiving event. Sending and receiving events is represented by "–" and "+", respectively. In SHARP, an event needs to have a specification of its arrival rate in states in which that event is enabled. Some of the state machines in Figure 1 include failure states (labeled with -1) that represent erroneous behavior triggered by a failure event F. Below, we outline how we derive a system's operational profile and the component failure states in our evaluations.

The system-level behavior of MIDAS is captured using five basic scenarios shown in Figure 2:

- the *SensorGW* scenario processes measurements from a *Sensor* by a *Gateway* (Figure 2(a));
- the *GWHub* scenario processes aggregated measurements from a *Gateway* to the *Hub* (Figure 2(b));
- the *GWAck* scenario acknowledges the *Sensor*'s measurement (Figure 2(c));
- the *GUIRequest* scenario updates the temperature readings and changes thresholds (Figure 2(d)); and
- the *ChangeACTemp* scenario turns *AC* on or off according to the temperature readings (Figure 2(e)).

The five basic scenarios are in turn combined to describe the overall system behavior as shown in Figure 3. This higher-level behavioral description consists of relations between basic and complex scenarios that together form a scenario hierarchy. The scenar-

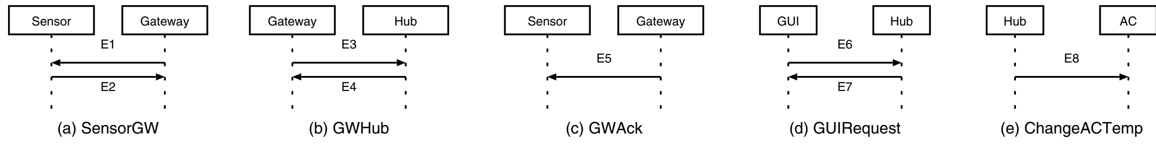


Figure 2: Sequence diagrams

ios can run concurrently or sequentially one after the other. In MIDAS, complex scenario *Sensors_PAR* represents the parallel execution of multiple *Sensor*s running the *SensorGW* scenario (Figure 3 describes a system variant with two gateways, each connects to two sensors). Furthermore, the complex scenario *SensorMeasurement* specifies a longer sequence that summarizes how a sensor measurement is propagated from *Sensor*s to *Gateway* to *Hub* and back. In SHARP, an application engineer also needs to annotate the scenario hierarchy with (1) branching probabilities when one scenario can be sequentially followed by multiple other scenarios, and (2) the number of scenario instances that run in parallel.

2.2 Prior Work

The focus of our previous research has been on (1) classifying and discovering *architectural defects* [24] and (2) analyzing and extracting *operational profile* information from various information sources [3, 15]. In SHARP, we analyze the impact of the discovered architectural defects (e.g., mismatches between components' operation protocols) on the system's reliability and enrich the initial component and scenario models with the extracted operational profile information. While we utilize our prior techniques to obtain the SHARP inputs, SHARP is in no way dependent on those techniques. In principle, any other existing technique can be used to extract the failure and operational profile operation. Note that defects that are introduced beyond the design stage, such as coding errors, are not considered in architecture-level reliability analysis, as our goal here is to evaluate the impact of different design decisions, rather than ensuring that system reliability meets certain requirements (e.g., five 9's rule). In the remainder of this section, we discuss how the failure and operational profile information is incorporated into the initial software models.

2.2.1 Failure information

To determine the possible system failure states, we first analyze the system components' architectural models by applying a defect classification technique [24]. The example defects detected using the technique in [24] include interface mismatches, specification incompleteness, and behavioral inconsistencies. Once defects are identified, we add a failure transition from each state in which a defect may be triggered. For example, applying the defect classification technique on MIDAS, we determine that a *Sensor* is unable to notify the *Gateway* when it is running out of battery. This defect was discovered as a mismatch between the two components' interaction protocols.[2] Failures caused by this defect are represented as the failure state -1 in Figure 1(a). As in most existing approaches [8, 12, 13, 15], we assume that time between failures is exponentially distributed.

In addition to modeling failures, SHARP supports modeling the recovery from a failure. For example, *Sensor* returns to State 2 as the *Sensor* recovers from the interaction protocol mismatch via a reset. In general, a component can return to any state designated by the system designer during defect analysis. In this paper, the running example and the evaluation systems have recoverable failures, and hence we apply steady state analysis (see Section 2.2.3 for details). However, SHARP can be adapted to apply transient analysis to handle irrecoverable failures with minimal modifications.

2.2.2 Operational profile

To parametrize the initial software models with operational profile information, we build on our previous work [3, 15] that estimates the parameters in component-level reliability models from the available information sources: domain knowledge, requirements documentation, system simulation, and execution logs of functionally similar systems. The specific value of the technique we introduced in [3] is that it allows an engineer to assess the quality of the different available information sources. For example, we can rely on domain knowledge to estimate the transition rates, but we have shown in [3] that such an estimate may be subjective and inaccurate for complex systems. Utilizing execution logs of an existing system is useful in predicting the operational profiles while designing a new version, but such information would be unavailable for estimating operational profiles regarding any new feature that is not present in the existing version.

The operational profile information is mapped into the model as transition rates for the desired behaviors as well as failure and recovery transition rates. For example, the event *GetSensorData* ($E1$) from Figure 1 is determined to occur every 5 seconds (rate of 0.2 events per second). At the system level, SHARP requires

[2]Note that the component models resulting in this interaction protocol mismatch [3] are elided for brevity.

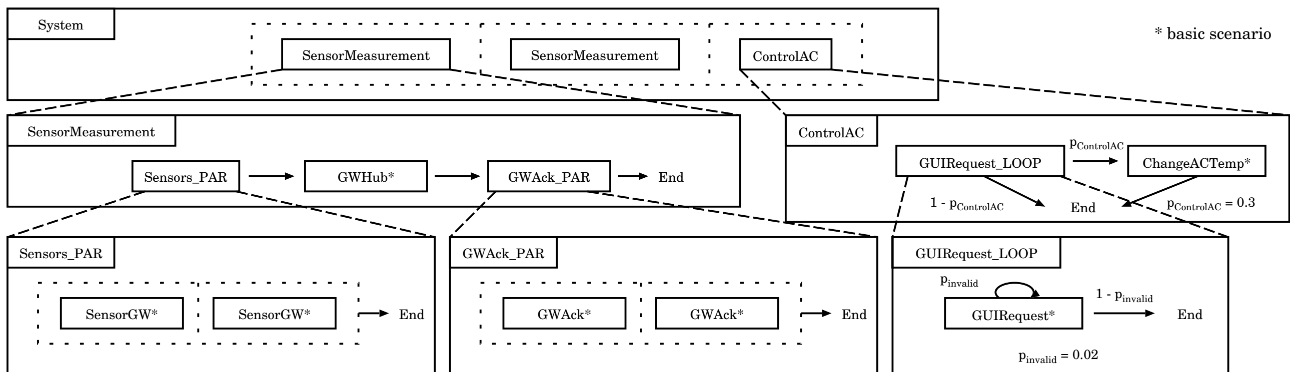
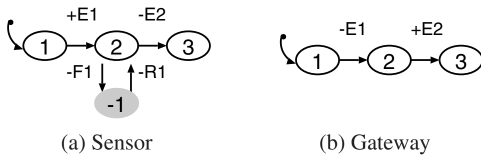


Figure 3: MIDAS scenarios organized in a hierarchy



(a) Sensor (b) Gateway

Figure 4: Component submodels of *SensorGW*

transition rates for the events in a scenario sequence, which are directly extrapolated from the component models. Furthermore, the initial scenario-level model is enriched in SHARP with information about the possible concurrently running scenario instances.

2.2.3 Generating Reliability Models of Basic Scenarios

We described how we generate a CTMC-based reliability model that we refer to as a scenario-based reliability model (SBM for short) in [4], and provide this information here for completeness.

To generate the SBM for a scenario, we first generate a component submodel for each component in each scenario, and then apply parallel composition, as in [22]. A component submodel of Component $Comp_c$ in Scenario $Scen_i$, $Comp_c_Scen_i$, is a state machine model describing the behavior of $Comp_c$ in $Scen_i$, in which a state transition represents the occurrence of an event in the corresponding sequence diagram. In our MIDAS example, the component submodels for the *SensorGW* scenario (recall Figure 2) are depicted in Figure 4.

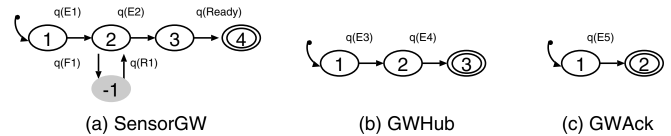
The next step is to add failure states to each component submodel. by leveraging the component model as described in Section 2.2.1. For example, to model the defect in the *Sensor*, represented by the failure state -1 in Figure 1(a), we add a failure state, State -1, in Figure 4, a failure transition from State 2 to State -1, and a recovery transition from State -1 to State 2.

We then generate an SBM for each basic scenario $Scen_i$ by applying parallel composition [16] to the component submodels $Comp_c_Scen_i$ for all $Comp_c$. Examples SBMs for the basic MIDAS scenarios as depicted in Figure 5. In our example, applying parallel composition to the component submodels in Figure 4 would result in the SBM for the *SensorGW* scenario depicted in Figure 5(a). (Note that State 3 in the *SensorGW* scenario (Figure 5(a)) corresponds to contention modeling, which we describe in Section 4.1).

Finally, we determine the transition rate between the states based on the operational profile estimated using the techniques described in Section 2.2.2. Formally, let Q_i be the transition rate matrix for $Scen_i$'s SBM. If the transition from State j to State k corresponds to the event E, the transition rate $Q_i(j,k) = q(E)$, where $q(E)$ is the rate that event E occurs. To complete the SBM, according to [25], we set the diagonal entries $Q(j,j)$ such that each row in Q sums to zeros.[3]

To solve for scenario reliability r_i, we redistribute the rate going to the SBM's End state to the Start state to analyze the system's long-term execution; the assumption here is that the particular scenario will eventually execute again. Once we have completed this step, we compute r_i by computing the probability of not being in the failure state when the system is in steady state. i.e., $r_i = 1 - \sum_{f \in F_i} \pi_i(f)$, where F_i is a set of failure states in $Scen_i$, and $\vec{\pi}_i$ is the steady state probability vector, which can be computed using standard methods [25].

[3]Note that self-loops in a component model (i.e., an event that causes a transition from a state to itself) have been implicitly accounted for here. Since self-loops do not cause any state transitions, they do not affect the probability distribution of being in a state in a CTMC, and are therefore dropped in an SBM.



(a) SensorGW (b) GWHub (c) GWAck

Figure 5: Example SBMs of MIDAS basic scenarios

3. AN OVERVIEW OF SHARP

In this section, we outline the envisioned benefits that motivated us to develop SHARP and provide a high-level description of the technical steps that comprise SHARP.

SHARP estimates reliability of a system based on (a) a non-probabilistic behavioral specification consisting of component-level state-based models and system-level scenario-based models, (b) the operational profile, and (c) the definition of failure states. At a high level, SHARP partitions the system behavior into smaller analyzable parts according to the scenario specification, generates a CTMC-based reliability model for each part, and computes system reliability by combining the results of the reliability models using a hierarchical approach we develop. With SHARP, we address two crucial obstacles that prevent the application of architecture-based reliability estimation techniques to complex, concurrent systems:

1. Efficiently solving a reliability model that captures complex system behaviors consisting of elaborate multi-scenario sequences.
2. Efficiently estimating system reliability when a system consists of a large number of concurrently running components and scenarios.

The first obstacle corresponds to the ability to deal with sequential scenario combinations without having to solve the corresponding non-scalable "flat" model used by other approaches [22]. The second obstacle relates to the need to handle multiple scenario instances running concurrently. For example, we want to be able to efficiently solve the *Sensors_PAR* scenario from Figure 3 even in situations when we have many concurrently running *Sensor*s. SHARP resolves these obstacles by first generating and solving the reliability models of smaller scenarios and then incorporating the results into reliability models of the complex scenarios; this is done in a bottom-up way throughout the specified scenario hierarchy.

SHARP consists of activities that determine (1) the reliability and completion times for the basic scenarios, and (2) the reliability and completion times for the complex, sequential (SEQ) and parallel (PAR) scenarios based exclusively on the reliability and completion times of the scenarios they reference. As an example, Figure 6 illustrates the steps that SHARP performs to analyze the reliability of the *SensorMeasurement* scenario from Figure 3. The process for obtaining the reliability information for *GWHub* and *GWAck_PAR* is identical to the process for *SensorGW* and *SensorsPAR* and is not shown. Intuitively, SHARP first analyzes the low-level, basic scenarios and incrementally incorporates the lower-level analysis results in the higher-level SEQ and PAR scenario models. As shown in Figure 6, analyzing a higher-level scenario model in SHARP involves reusing the results from the lower level without the need to recalculate or refine the calculations.

For a basic scenario, we generate a SBM as described in Section 2.2.3. The unique aspect of our approach is that we slice the component reliability models according to the basic scenario in order to determine the possible failure states related to that scenario's

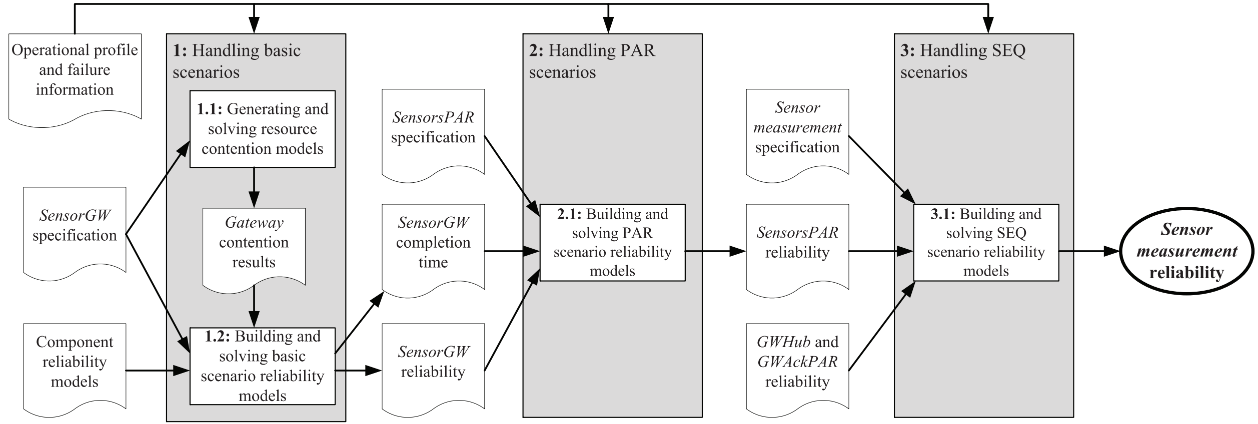


Figure 6: An illustration of SHARP applied on the complex *SensorMeasurement* scenario

execution. By doing so, we reuse information about architectural defects, thus making the reliability analysis more meaningful as opposed to having an engineer "guess" the failure states.

A software architect can choose to augment the SBM to model resource contention with special "queueing" states. Intuitively, a queueing state simulates a situation when an event may not be processed immediately because of resource constraints. For example, while a single-threaded *Gateway* is processing data from one *Sensor*, it may receive data from another; consequently, we augment the SBM by adding a state to represent queueing of the *Sensor*'s request (State 3 in the $SensorGW$ SBM from Figure 5). In Section 4.1, we describe how we obtain the contention-related parameters using queueing networks (QNs) [25]. The basic scenario SBMs are finally solved for scenario reliability and completion time using standard methods. Note that completion times are necessary for SHARP to handle PAR scenarios.

To solve for the reliability of a complex scenario with sequential dependencies (e.g., the *SensorMeasurement* from Figure 3) in which a number of basic scenarios may be running one after another, we propose a technique based on stochastic complementation [18]. Stochastic complementation is a standard technique for solving large Markov chains that relies on partitioning a large model into smaller analyzable parts. Our application of stochastic complementation utilizes the partitioning that is intrinsically present in a SEQ scenario where each sub-scenario has only one entry point. For example, when analyzing the MIDAS scenario hierarchy (Figure 3) SHARP utilizes the SEQ relations in the *SensorMeasurement* scenario to solve *Sensors_PAR*, *GWHub*, and *GWAck_PAR* first, and then incorporate the obtained results into a small, high-level *SensorMeasurement* model with only three states. Technically, this corresponds to lumping the states of each sub-scenario into one state, and then solving one "flat" CTMC that represents the three sub-scenarios. Note that stochastic complementation does not result in a less accurate solution than a "flat" model — the computational savings come for free. SHARP attains additional computational savings by computing the reliability of lower-level scenarios only once and then reusing the results in multiple SEQ scenarios that reference them.

Using the existing state-of-the-art, computing the reliability of a complex parallel (PAR) scenario would require keeping track of the internal states of all components during system execution. As discussed earlier, this approach is intractable for larger concurrent systems. To this end, we propose symbolic representation of the system execution state that utilizes a CTMC to keep track of the number of the currently running scenario instances. To abstract a scenario's execution state to either running or completed, SHARP uses the completion times that are calculated for the child scenarios. Each state of a PAR scenario SBM can be described as a *combination* of child scenarios. For example, we aggregate the overall behavior of *Sensors_PAR* from Figure 3 with an SBM depicted in Figure 9 that tracks whether there are zero, one, or two concurrently running instances of *SensorGW*. In Section 4.3, we detail the steps involved in solving a PAR scenario's reliability and we evaluate the accuracy of the obtained results obtained in Section 5.

Table 1: Definition of Variables used in SHARP

Variable	Definition
$Q(i,j)$	Transition rate from State i to State j in an SBM
$q(E)$	Avg rate that event E occurs
$C_i = (c_1, c_2, \ldots)$	A combination with c_j instances of $Scen_j$
$P_j(c_j)$	Probability of having c_j instances of $Scen_j$
I_j	Max number of instances of $Scen_j$
I	$max_j(I_j)$
d_j	Avg delay caused by $Scen_k, k \neq j$
$P(C_i)$	Probability that Combination C_i occurs
$R(C_i)$	Reliability of Combination C_i
r_i	Reliability of $Scen_i$
t_j	Completion time of $Scen_j$
S	Total num of unique scenarios
H_j	Num of child scenarios of $Scen_j$

4. DETAILS OF SHARP

For space reasons, we cannot provide a detailed treatment of every step in SHARP. Instead, we choose to "dive into" the parts of SHARP that are novel; we omit details of the parts that are computationally straightforward and identical to other existing approaches [3, 22, 30] (e.g., computation of reliability and completion times for a CTMC). Namely, we present the details of *contention modeling and analysis* at the level of basic scenarios (Section 4.1), *combining scenarios with sequential dependencies* (Section 4.2), and *scalable concurrent behavior modeling and analysis* (Section 4.3). A list of variables that would be used in this section are summarized in Table 1.

4.1 Contention Modeling

In Section 2.2.3, we discussed the comparatively straightforward steps taken to devise a scenario's corresponding CTMC. We solve

Table 2: r_i and t_i of the MIDAS scenarios

Scenario	r_i	t_i	Scenario	r_i	t_i
SensorMeasurement	0.9867	27.394	GUIRequest	0.9999	201.03
ChangeACTemp	1	0.5	GUI_LOOP	0.9999	205.13
ControlAC	0.9999	205.28	System	0.9940	287.46

these CTMCs for reliability and completion times using standard techniques [25]. For completeness, we provide the computed results for the MIDAS scenarios in Table 2.

When several components (callers) request services from a servicing component (callee), the callee needs to allocate its resources appropriately to serve a caller, while other callers would need to wait to obtain service.[4] Since the system behavior may be different when a request is waiting for service than when it is being processed, we add a *queueing* state to represent that a caller's request is queued. Formally, let E be an event with an arrival rate of $q(E)$ that triggers a transition from State j to State k in an SBM. If there is a component that may be servicing other requests upon receiving E, we add a queueing state q into the SBM, such that $Q(j, \mathsf{q}) = q(E)$, and $Q(\mathsf{q}, k) = q(Ready)$. $Ready$ is an event indicating that the callee is ready to process the request of the caller of interest, and hence $q(Ready)$ corresponds to the time waiting for the callee's service. Note that $q(Ready)$ excludes the callee's service time, as the service time has been accounted for in the caller's SBM. [5] As an example, in the *SensorMeasurement* scenario, the *Gateway* could be servicing a *Sensor*'s request when another *Sensor* sends a request. Therefore, we insert State 3 between States 2 and 4 to represent queueing, as in Figure 5(a). Any other points of contention would be modeled in a similar manner.

The next step is to determine $q(Ready)$, the outgoing rate of the queueing state. We define $q(Ready) = \frac{1}{T_{wait}}$, where T_{wait} is the average time a caller waits to receive service. To compute T_{wait}, we solve a *queueing network* (QN) [25], which describes the queueing behavior of the callers' requests. Here, we model the callee component as a single server in the QN. Since information about how a component is implemented and deployed may be unavailable during design, it is unclear how to incorporate other resources (e.g., CPU and memory) into the QN. Therefore, we model the callee component as a black box. This QN can be refined if such information is available.

To build such a QN, we utilize the following information: (a) the number of different types of callers (i.e., the different types of components where each type can request different services with different processing times), and the maximum number of each type of caller that may request a service; (b) how often a caller requests a service (arrival rate); (c) how long the callee takes to serve a request (service rate);[6] and (d) the callee's queueing discipline. Note that (a) is derivable from the system's requirements and architectural models; (b) is available from the operational profile (i.e., the rate of an event E); and (c) is the total rate leaving a state k also derivable from the operational profile. The operational profile information and the other model parameters can be determined using

[4]As the flat model results in callees serving the callers on a FCFS basis, we also use FCFS in our exposition. However, SHARP allows other queueing disciplines.

[5]By plugging $q(Ready)$ into a component's SBM (a CTMC), we are essentially assuming that the waiting time is exponentially distributed, which may not be the case in general. We make this approximation for modeling convenience.

[6]For ease of exposition, we assume both the inter-arrival time and service time are exponentially distributed in this paper. In general, these assumptions can be removed, but it may be computationally costly to solve for $q(Ready)$ if the resulting QN is not in product-form [1].

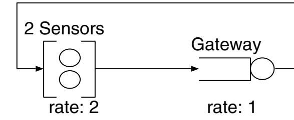


Figure 7: QN model of the *SensorMeasurement* scenario

our previous technique [3]. Lastly, (d) can be available from the system's requirements pertaining to architectural constraints (e.g., using a middleware that serves requests in a round-robin fashion). The constructed QN is finally solved for the average waiting time in the queue using standard methods [25]. Since (a)-(c) are readily available, once the contention points have been identified and (d) the queueing discipline has been specified, the process of generating and solving the QN for $q(Ready)$ can be automated.

The QN for the *SensorMeasurement* scenario is depicted in Figure 7. As an example, we are modeling the case when a *Sensor* sends measurements to the *Gateway* while the *Gateway* is processing another *Sensor* request. Hence, we have one class of callers: two *Sensors* may send measurements to the *Gateway* with the arrival rate of $2 \times q(E2) = 2$, processing rate of $q(E3) = 1$, and the *Gateway* is a FCFS callee. After solving the QN for the average waiting time at *Gateway*'s queue in Figure 7, the resulting rate of leaving the queueing state (state 3) in Figure 5(a) is estimated to be 5. Other points of contention in the SBMs are treated analogously.

4.2 Combining Scenarios with Sequential Dependencies

To analyze a complex scenarios with sequential dependencies, we apply stochastic complementation to generate a SEQ scenario's SBM by combining the SBMs of the child scenarios (Step 2.1). This is a novel use of an advanced stochastic method for analyzing a software system's quality attribute, and comprises an important contribution of this paper. Intuitively, stochastic complementation breaks a large Markov model into a number of submodels, solves the submodels separately, and reconstructs results of the original model. The special structure required for an efficient solution using stochastic complementation is that each submodel has only one start state. Notably, the generated basic scenario SBMs satisfy this requirement as they have a single starting state. We solve the resulting SEQ scenario SBM for scenario reliability (Step 2.2) in a similar manner to existing approaches [6, 22, 30].

4.2.1 Step 2.1: Generating SBM

We generate an SBM for a SEQ scenario as follows: we first generate the states of the model, and then compute the transition rates with respect to the applied stochastic complementation [18]. The states in a SEQ scenario's SBM correspond to the child scenarios. We determine the transitions according to the dependencies between the child scenarios. If a SEQ scenario $Scen_i$ has a child scenario $Scen_k$ executing after another child scenario $Scen_j$, we add a transition from state j to state k in $Scen_i$'s SBM. For example, the SBMs of the SEQ scenarios in MIDAS are depicted in Figure 8.[7]

The transition rates for each transition determined above are calculated as follows [18]:

$$Q_i(j,k) = (p_i(j,k))out_j \quad (1)$$

where $p_i(j,k)$ is the probability that $Scen_k$ executes after the execution of $Scen_j$, and out_j is defined in a similar manner to [18]:

[7]Note that the self-loop in State 1 of the *GUI_LOOP* scenario (depicted as dotted arrow in Figure 8), representing that the user's input is invalid, has been dropped, because a CTMC implicitly accounts for self-loops.

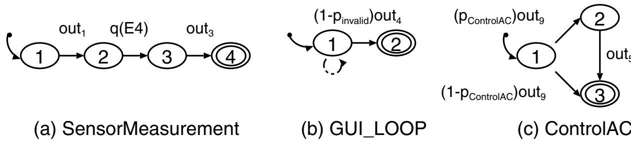


Figure 8: SBMs of the SEQ scenarios

$$out_j = \sum_{s \in S_j} (\pi_j(s)) Q_j(s, End) \tag{2}$$

where S_j is a set of states in $Scen_j$, $\pi_j(s)$ is the steady state probability of being in State s in the $Scen_j$'s model, and $Q_j(s, End)$ is the transition rate going from State s to the End state in $Scen_j$'s model. For example, in *GUI_LOOP* (Figure 8(b)), let $Scen_i = GUI_LOOP$ and $Scen_j = GUIRequest$, the transition rate from State 1 to State 2 in $Scen_i$ is

$$\begin{aligned} Q_i(1,2) &= p_i(1,2)(\sum_{s \in S_j} \pi_j(s) Q_j(s, End)) \\ &= (1 - p_{invalid})(\pi_j(2))(q(E7)) \\ &= (0.98)(0.005)(1) = 0.0049 \end{aligned}$$

Furthermore, when we move up a level in the hierarchy to *ControlAC* (Figure 3) the transition rate going from State 1 to 2 in *ControlAC*'s SBM in (Figure 8(c)) becomes

$$\begin{aligned} Q_i(1,2) &= p_i(1,2) \sum_{s \in S_j} \pi_j(s) Q_j(s, End) \\ &= (p_{ControlAC})(1) Q_j(s, End) \\ &= (0.3)(1)(0.0049) = 0.0015 \end{aligned}$$

4.2.2 Step 2.2: Computing Scenario Reliability

To solve for a sequential scenario's reliability, we redistribute the rate going to the End state of a SEQ scenario SBM and solve the model for its steady state probability vector $\vec{\pi}_i$, as in Section 2.2.3. Once we have computed $\vec{\pi}_i$ and r_j for all child scenarios $Scen_j$, we solve the scenario reliability using the equation

$$r_i = 1 - \sum_j \pi_i(j) r_j \tag{3}$$

Continuing with our example, to solve for $\vec{\pi}_i$ using the SBM of the *ControlAC* scenario, after redistributing the rate going to the End state, we have the following rate matrix:

$$\begin{bmatrix} -0.0015 & 0.0015 \\ 2 & -2 \end{bmatrix} \tag{4}$$

Solving this model gives us $\vec{\pi}_i = [0.9993, 0.0007]$. The reliability of the *ChangeACTemp* is 1, as there is no defect identified in that scenario. Hence, the reliability of the *ControlAC* scenario is $r_i = (0.9993)(0.9999) + (0.0007)(1) = 0.9999$. The reliabilities of other scenarios, and are depicted in Table 2, are similarly computed.

4.3 Scalable Concurrency Modeling

This section describes the SHARP steps that are performed when calculating the reliability of a complex PAR scenario. Note that we refer to the symbolic model discussed in Section 3 as the concurrency-level model. SHARP first determines the feasible scenario combinations (Step 3.1), and constructs the concurrency-level model (Step 3.2). Next, SHARP calculates the probabilities (Step 3.4) and the reliabilities (Step 3.5) of the different scenario combinations. SHARP ultimately uses the obtained information to compute the overall PAR scenario reliability (Step 3.6). Noting that the concurrency model can get intractable when dealing with very large systems for which concurrent scenario instances may number in the thousands, SHARP employs model truncation [25] (Step 3.3).

Table 3: Values of $P(C_k)$ and $R(C_k)$ in the *System* scenario

C_i	$P(C_i)$	$R(C_i)$	C_i	$P(C_i)$	$R(C_i)$
(0,0)	0.0420	0.9605	(1,0)	0.0630	0.9735
(2,0)	0.1260	0.9866	(3,0)	0.7266	0.9999
(0,1)	0.0077	0.9606	(1,1)	0.0116	0.9736
(2,1)	0.0231	0.9867			

4.3.1 Step 3.1: Determining Scenario Combinations

Determining the possible combinations is the first step in solving for reliability and completion time of a PAR scenario. A combination, C_i, is defined as $C_i = (c_1, c_2, \ldots, c_{H_j})$, where c_j is the number of completed instances of $Scen_j$, and H_j is the number of child scenarios.[8] We also define I_j to be the number of instances of $Scen_j$ that needs to be completed, and $\mathbf{I} = max(I_j)$ to be the largest number of possible instances among all $Scen_j$. The execution of a PAR scenario is completed only when all child scenarios have completed their execution.

In order to find scenario reliability, we need to compute the distribution of the possible combinations. Since, in general, not all combinations of scenarios in a system may be possible, we allow a system architect to specify the combinations that are not possible (or allowed). For instance, in MIDAS, such a restriction exists to avoid exhausting the resources of the *Hub* by allowing no more than three *Hub* requests. Hence, in the *System* scenario from Figure 3, if we set $I_1 = 3$ and $I_2 = 1$, and include the restriction that $I_1 + I_2 \leq 3$, then the possible scenario combinations are those depicted in Table 3. Note that the I's are small in our example as we try to keep it simple.

4.3.2 Step 3.2: Concurrency-Level Model Generation

To compute the probability $P_j(c_j)$ that c_j instances of a child scenario $Scen_j$ have completed, we generate a concurrency-level model for each child scenario. A concurrency-level model is a CTMC, whose states correspond to the number of completed instances of $Scen_j$. We use the concurrency-level model to compute the probabilities $P_j(c_j)$, where c_j denotes the number of completed $Scen_j$ instances. The determination of the final state in a concurrency-level model depends on the number of concurrently running child scenarios.

When there is one child scenario, completing I_j instances of $Scen_j$ represents the completion of the whole PAR scenario. For example, the concurrency-level models corresponding to *Sensors_PAR* and *GWAck_PAR* in MIDAS are depicted in Figures 9(a) and (b), respectively. The final state in these models is state 2 because there can be two concurrently running instances of *Sensors_PAR* and *GWAck_PAR*.

When there is more than one child scenario, completing I_j instances of $Scen_j$ means that the execution of all instances has been completed. $Scen_j$ can only execute again when all other scenarios have been completed, and the parent scenario executes again. We

[8] We assume that the probability that more than one scenario 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.

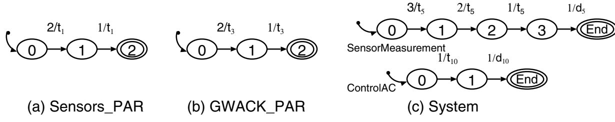


Figure 9: SBMs of the PAR scenarios

Table 4: Values of $P_j(c_j)$ in the *System* scenario

SensorMeasurement				*ControlAC*	
Parameter	Value	Parameter	Value	Parameter	Value
$P_1(0)$	0.0305	$P_1(2)$	0.0916	$P_2(0)$	0.8949
$P_1(1)$	0.0458	$P_1(3)$	0.8321	$P_2(1)$	0.1051

add a state End to model this behavior, and define State End to be the final state of each concurrency-level model. We also add a transition from state I_j, which corresponds to completing all instances of $Scen_j$, to State E. For example, the concurrency-level models corresponding to the *System* scenario are depicted in Figure 9(c).

The transition rate from state c_j ($c_j < I_j$), representing c_j completed instances of $Scen_j$, to state $c_j + 1$ is $(I_j - c_j)(1/t_j)$, where t_j is completion time of $Scen_j$, which is assumed to be exponentially distributed. The transition rate corresponds to the rate an instance of $Scen_j$ completes, when there are c_j completed instances. t_j can be computed using standard techniques, which involves solving for the average passage time from State 1 to any End state in the SBM using $Q'_j\vec{T_j} = -e$ [25], where Q'_j is the matrix after eliminating the row and column corresponds to the End state in Q_j, $-e$ is a column vector of -1 with the appropriate dimension, and $T_j(k)$ is the average passage time from State k to the End state. i.e., $t_j = T_j(1)$.

The transition from state I_j to state End has a rate of $1/d_j$, where d_j is the average of the total delay caused by other scenarios $Scen_k$, $k \neq j$. We set $d_j = \sum_{k\neq j} I_k t_k$.[9]

4.3.3 Step 3.3: Performing Model Truncation

To further reduce the computational cost, we eliminate the combinations in a PAR scenario that are rarely visited according to a model truncation technique from [25].

The steady probability distribution of c_j, the number of completed instances of $Scen_j$, depends on the values of t_j (scenario completion time), as well as the completion time of other scenarios t_k, $Scen_k \neq Scen_j$. $P_j(c_j)$ can be obtained by solving a concurrency-level model using standard techniques. As an illustration, we depict the steady-state probability distribution of c_j in Figure 10. We assume the completion rate of other scenarios are fixed, and $d_j = 1$. Also, we set $\mathbf{I} = 50$, and varied t_j at different values. For instance, when $t_j = 100$, 29 (out of 51) possible values of $P_j(c_j)$ become smaller than 1%.

In generating the scenario combinations, we can elide the values of the completed scenario instances c_j that occur rarely. Specifically, we consider x as a relevant value of c_j if $P_j(c_j = x)$ is larger than a threshold ϵ (thus, the case without using truncation has $\epsilon = 0$). For example, if $\epsilon = 0.01$ (depicted as a dotted line in Figure 10), when $t = 100$, $d_j = 1$, and $\mathbf{I} = 50$, we only consider 22 out of the 51 state in the concurrency-level model.

Note that there is a tradeoff between the number of states we elide and the loss in accuracy when applying model truncation. We evaluate this tradeoff in Section 5.2.

[9]Note that d_j is an approximation, which assumes that the average time to complete an instance of $Scen_k$, given that $Scen_j$ has completed, is still t_k. An exact computation of d_j involves transient analysis, which is computationally more expensive [25].

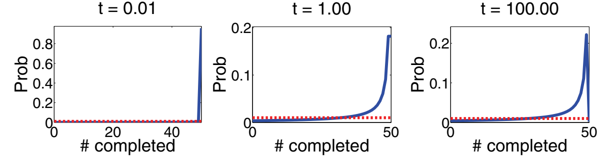


Figure 10: Probability distribution of the number of completed instances

4.3.4 Step 3.4: Computing Combination Probability

SHARP solves the concurrency-level model for the probability distribution of each combination. We define $P(C_i)$ to be the probability that C_i occurs, i.e., $P(C_i) = P(c_1, c_2, \ldots, c_{H_j})$ is the probability that there are c_j completed instances of $Scen_j$ for each $j = 1 \ldots H_j$. Since we assume that all instances of all child scenarios run independently, $P(C_i) \simeq (\prod_j P_j(c_j))/W$, where $P_j(c_j)$ is the probability that c_j instances of $Scen_j$ have completed, and $W = \sum_j P(C_j)$ is a normalization factor[10] that ensures that $P(C_j)$ sum to 1. In the *System* scenario from Figure 3, $P(c_1, c_2)$ is the probability that there are c_1 and c_2 completed instances of *SensorMeasurement* and *ControlAC*, respectively. Hence, $P(c_1, c_2) \simeq (P_1(c_1) \times P_2(c_2))/W$.

Assuming that once the whole PAR scenario has completed its execution it will be executed again, we merge the final state F and the initial state 0. We solve the resulting model for $P_j(c_j)$ using standard techniques [25]. Table 4 gives the probability distribution of $P_j(c_j)$ in the two PAR scenarios of our MIDAS example; these are computed using the concurrency-level models from Figure 9. Furthermore, Table 3 gives the *System* scenario combination probabilities, computed using the data from Table 4. Since the computation of the distribution of different scenario combinations is done in an approximate manner, in Section 5 we evaluate the accuracy of this approximation, as well as the reduction in computational cost.

At the level of the entire system, the assumption that a PAR scenario is completed only when all child scenarios have finished executing may be restrictive. In some systems that continuously run, child scenarios start and complete independently. For example, after the *Sensor*s finish taking measurements, they do not necessarily have to wait for the *GUI* to update the data before taking additional measurements. In [4], we handle these independent concurrent scenarios with a modified concurrency-level model. Specifically, this model has additional transitions that represent the spawning of new scenario instances and are directed opposite to those depicted in Figure 9.

4.3.5 Step 3.5: Computing Combination Reliability

Here, we make a simplifying assumption that the system fails if any scenario instance has failed. SHARP can accommodate more complex failure conditions as discussed in [4]. Given this assumption, the reliability of a combination C_i is $R(C_i) = \prod_{j=1}^{H_j} r_j^{(I_j - c_j)}$. We repeat this calculation for each combination. Table 3 gives reliabilities of all scenario combinations for MIDAS.

4.3.6 Step 3.6: Computing Scenario Reliability

We compute scenario reliability by combining the results of the previous steps. Reliability of a PAR scenario is defined as the sum of the scenario combinations' reliabilities, weighted by the probability that the combination occurs, i.e., $r_i = \sum_k P(C_k)R(C_k)$.

In our running example, the reliability of the *System* scenario, and hence system reliability, is 0.9940, which, in this case, is within

[10]The normalization factor is needed because, in general, not all combinations of scenarios may be allowed.

Table 5: Definition of Variables used in Complexity Analysis

$\mathbf{U}$	Total num of unique components
$\mathbf{C}$	Total num of components
$\mathbf{E}$	Total num of events
$\mathbf{S}$	Total num of unique scenarios
$\mathbf{S_B}$	Total num of basic scenarios
$\mathbf{S_I}$	Total num of intermediary scenarios
I_j	Max number of instances of $Scen_j$
$\mathbf{I}$	$max_j(I_j)$
M_j	Max number of states in $Comp_j$
$\mathbf{M}$	$max_j(M_j)$

0.05% of the ground truth of 0.9935, obtained by solving the "flat model", as detailed below.

5. EVALUATION

We evaluate SHARP along two dimensions: *complexity* of generating and solving concurrent systems' reliability models as compared to those derived from existing techniques (Section 5.1), and *accuracy* (Section 5.2). We compare SHARP against a flat model, which is used as the "ground-truth". Flat model is essentially used by Rodrigues et al. [22], where a system reliability model is generated by applying parallel composition to component models.[11]

We applied SHARP to a variety of systems, with different numbers of components, scenarios, and numbers of scenario instances. Note that the system we used in evaluating SHARP is relatively simple, because the flat models of larger systems quickly become too large to solve, and we would lack an objective baseline for comparison. We show representative results obtained from the following systems:

1. An instantiation of MIDAS with twelve *Sensors*, six *Gateways*, one *Hub*, one *GUI*, and one *AC*.

2. A GPS system with route guidance, audio player, and bluetooth phone capabilities. This system has five major components. The system's behavior is captured with 21 basic scenarios. The GPS system has limited concurrency due to the system's purpose. For example, it typically makes little sense to have two instances of a route guidance scenario to perform the same route guidance service.

   To evaluate SHARP in a controlled manner, we injected the following defects into this GPS system: (a) a defect in the *EnergyMonitor (EM)* component which may lead to failure to notify other system components when the battery is low, and (b) a defect in the *RouteGuidance (RG)* component which may lead to failure in updating a user's location accurately.

3. A client-server system (CS) with possibly many clients and a single server that provides remote file access. The system behavior is described with one basic scenario. The server processes the client requests in a FCFS fashion with an assumption of infinite buffer space. We primarily leverage CS to illustrate the effect of contention modeling when there are many clients competing for the same resource. We consider a defect in the server that may lead to failure to reply to the client when a requested file cannot be retrieved.

[11] [22] assumes irrecoverable failures. As we discussed earlier, SHARP can model irrecoverable failures with minor modifications.

Table 6: Worst-case complexity

Complexity	SHARP	Flat Model
Time	$O(\mathbf{S_I}\ \max(\mathbf{S^3}, \mathbf{SI^3} + (\mathbf{S}+\mathbf{2})\mathbf{I^S})+$ $\mathbf{S_B}(\mathbf{M^{3C}} + \mathbf{M^C EI^E}))$	$O(\mathbf{M^{3C}})$
Space	$O(\mathbf{S} +\ \max(\mathbf{M^{2C}}, \mathbf{S^2}, \mathbf{SI}))$	$O(\mathbf{M^{2C}})$

5.1 Complexity Analysis

We now explore the complexity of SHARP as compared to the flat model. We first describe the theoretical worst-case complexity of each approach in Section 5.1.1, and then discuss the computational cost that is likely to arise in practice in Section 5.1.2.

5.1.1 Worst Case Complexity

Let $\mathbf{U}$ be the number of unique components, $\mathbf{C}$ be the total number of components, $\mathbf{E}$ be the number of events, $\mathbf{S}$ be the number of scenarios (basic and intermediary), $\mathbf{S_B}$ be the number of basic scenarios, $\mathbf{S_I}$ be the number of intermediary scenarios (i.e., $\mathbf{S} = \mathbf{S_B} + \mathbf{S_I}$), I_j be the number of instances of $Scen_j$, $\mathbf{I} = max(I_j)$ for all $Scen_j$, M_j be the number of states of $Comp_j$, and $\mathbf{M} = max(M_j)$ for all $Comp_j$. These definitions are summarized in Table 5, and the resulting complexities are summarized in Table 6. Let us first analyze the complexity of SHARP:

Basic scenarios: In the worst case, every state in every component participate in a basic scenario, and hence the SBM may have as many as $O(\mathbf{M^C})$ states. Once we have determined the states in the SBM, we need to determine the transitions between each pair of states. Therefore, the complexity of the generation of a SBM is $O(\mathbf{M}^{2\mathbf{C}})$. The complexity of solving a SBM [12] is $O(\mathbf{M}^{3\mathbf{C}})$. Thus, the time complexity of generating and solving the SBM for a basic scenario is $O((\mathbf{M^{2C}} + \mathbf{M^{3C}})) = O(\mathbf{M^{3C}})$. The space complexity of generating and solving a basic scenario's SBM is $O(\mathbf{M^{2C}})$ — once we have solved a SBM, we can reuse its space as we generate SBMs one at a time.

Contention Modeling: In the worst case, there is contention in every state in the SBM of a basic scenario. If, as a result, we add a queueing state corresponding to each state, we double the size of every SBM of each basic scenarios, which does not affect the worst case complexity of solving it ($O((2\mathbf{M^C})^3 = O(8\mathbf{M}^{3\mathbf{C}}) = O(\mathbf{M}^{3\mathbf{C}})$). Thus, in the worst case, we have $O(\mathbf{M^C})$ QNs to solve. Since we assume product-form QN, the worst case complexity of solving one such QN is $O(\mathbf{EI^E})$ [20]. Thus, the worst case time complexity of solving all QNs would be $O(\mathbf{M^C EI^E})$.

SEQ scenarios: Since there are at most $\mathbf{S}$ scenarios in the system, there are at most $\mathbf{S}$ states in the SBM of a SEQ scenario, because each scenario is represented by a state in the SBM of a SEQ scenario. Therefore, the complexities of generating and solving the SBM of a SEQ scenario are $O(\mathbf{S}^2)$ and $O(\mathbf{S}^3)$, respectively, and the space complexity is $O(\mathbf{S}^2)$, as discussed above.

PAR scenarios: In the worst case, all $\mathbf{S}$ scenarios run in parallel. In Step 2.2, since each concurrency-level model has at most $O(\mathbf{I})$ states, the complexity of solving for all $\mathbf{S}$ of them is $O(\mathbf{SI^3})$. Step 2.4 requires computing $P(C_j)$ for each $Comb_j$, therefore the complexity is $O(\mathbf{I^S})$ as we have I^S combinations. In computing the reliability of a combination in Step 2.5, we need to multiply the reliabilities for each child scenario $Scen_i$, and hence the complexity of this step is $O(\mathbf{SI^S})$. Finally, in Step 2.6, we compute scenario reliability by multiplying $R(C_j)$ and $P(C_j)$ for each scenario, so the complexity is $O(\mathbf{I^S})$. Therefore, the complexity of solving for reliability of a PAR scenario is $O(\mathbf{S_I}(\mathbf{SI^3} + \mathbf{2I^S} + \mathbf{SI^S})) = O(\mathbf{S_I}(\mathbf{SI^3} + (\mathbf{S}+\mathbf{2})\mathbf{I^S}))$. The space complexity of solving the

[12] The time complexity of solving a Markov chain with $\mathbf{N}$ states is $O(\mathbf{N^3})$, and the space complexity for storing the corresponding rate matrix is $\mathbf{N^2}$.

Table 7: Summary of computational costs in practice

	$\mathbf{U}$	$\mathbf{C}$	$\mathbf{M}$	$\mathbf{S_B}$	$\mathbf{I}$	$\mathbf{E}$	Flat Model	SHARP
MIDAS	5	12	5	5	6	8	1.52×10^{12}	692
GPS	5	5	17	21	1	43	1.17×10^{11}	1331
CS	2	9	2	1	8	2	1.34×10^{8}	737

PAR scenarios is $O(\mathbf{SI})$, as we need to store the results of the concurrency-level models.

Note that we have not considered the computational cost savings of model truncation (Step 2.3) in this complexity analysis, as model truncation does not reduce the worst-case complexity.

Overall Complexity: First, since there are $\mathbf{S_B}$ basic scenarios, the complexity of generating and solving all $\mathbf{S_B}$ SBMs of the basic scenarios is $O(\mathbf{S_B}(\mathbf{M^{3C}} + \mathbf{M^C EI^E}))$. There are $\mathbf{S_I}$ intermediary scenarios, which each of them could either be s SEQ or PAR scenario. As we do not know which of SEQ or PAR scenario is more expensive to solve in the worst case (it depends on the values of $\mathbf{S}$ and $\mathbf{I}$), we describe the complexity of solve an intermediary scenario to be $O(max(\mathbf{S^3}, \mathbf{SI^3} + (\mathbf{S}+\mathbf{2})\mathbf{I^S}))$. Therefore, the overall time complexity is $O(\mathbf{S_B}(\mathbf{M^{3C}} + \mathbf{M^C EI^E})) + \mathbf{S_I} \max(\mathbf{S^3}, \mathbf{SI^3} + (\mathbf{S}+\mathbf{2})\mathbf{I^S}))$.

In analyzing the overall space complexity, we need to consider the space needed to store the results of the scenarios that have been processed, in addition to the space needed to store the SBM of the scenario that is being processed. Since we store the r_i and t_i of each $\mathbf{S}$ scenario in the worst case, the space needed to store the results of $\mathbf{S}$ scenarios is $O(2\mathbf{S})$. The "last" scenario could be a basic, SEQ, or a PAR scenario, so the space complexity is the maximum space needed among the three types of scenarios. Thus, the overall space complexity is $O(\mathbf{S} + max(\mathbf{M^{2C}}, \mathbf{S^2}, \mathbf{SI}))$.

In the flat model, we first apply parallel composition using all components, for which the complexity is $O(\mathbf{M^{2C}})$. The time complexity of solving the flat model is $O(\mathbf{M^{3C}})$. Therefore, the overall time complexity of the flat model approach is $O(\mathbf{M^{2C}} + \mathbf{M^{3C}}) = O(\mathbf{M^{3C}})$. Since the flat model has as many as $O(\mathbf{M^C})$ states, its space complexity is $O(\mathbf{M^{2C}})$.

5.1.2 Computational Cost Analysis

Table 7 summarizes the computational costs to solve for system reliability using the flat model and SHARP. We denote the number of unique components with $\mathbf{U}$; $\mathbf{C}$ is the total number of components; $\mathbf{S_B}$ is the number of basic scenarios; and $\mathbf{M} = max(\mathbf{M}_j)$, where $\mathbf{M}_j$ is the number of states of $Comp_j$. Additionally, $\mathbf{I}_i$ is the number of instances of $Scen_i$, and $\mathbf{I} = max(\mathbf{I}_i)$ for all $Scen_i$. The computational cost savings using SHARP are significant for all three systems' evaluation.

Comparing the computational costs of the three systems yields some interesting observations. We noticed that it is more expensive to solve the GPS system model than the MIDAS model, since the GPS system is modeled with 21 basic scenarios and we generate and solve an SBM for each scenario. Although systems with more basic scenarios are more expensive to solve in SHARP, the computational costs in practice are still significantly lower as compared to the flat model. While CS is simpler than MIDAS, it costs more to solve it for reliability. This is because the number of parallel scenario instances is larger in CS ($\mathbf{I} = 8$) than MIDAS ($\mathbf{I} = 3$), which results in a larger PAR scenario model.

Since SBMs are likely to be smaller than the flat model, we argue that SHARP in practice requires significantly less space than the flat model. The savings are also due to the fact that we can generate and solve SBMs one at a time, and thus reuse the space. Furthermore, given that SHARP takes the approach of solving many smaller models rather than one large model, it is possible to solve the different branches of the hierarchy in parallel. The results discussed above were confirmed by a number of other example systems.

5.2 Accuracy

Our goal is to provide evidence that SHARP is sufficiently accurate to be used in making design decisions. Therefore, we compare the *sensitivities* of SHARP and the corresponding flat model: if the differences in the changes of reliability estimates are reasonably small when the same parameter is varied in the two models, then SHARP can be considered accurate.

5.2.1 Sensitivity Analysis

First, we compared the sensitivities of SHARP and the flat model when model parameters change. 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 performed similar experiments by varying other parameters and using other systems' models. The results were qualitatively similar.

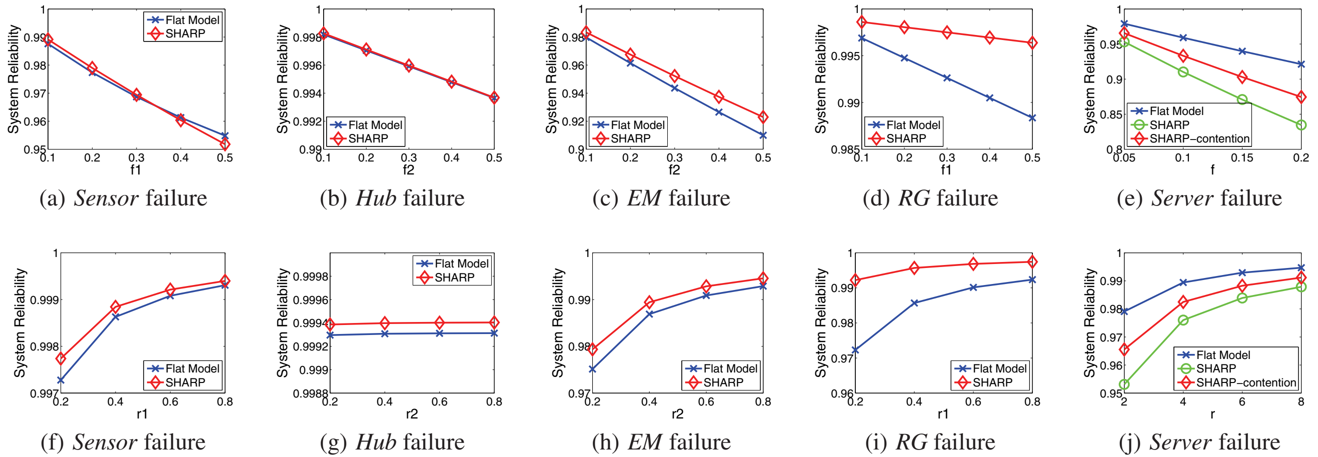
The inaccuracies in our estimates come from the solution of the PAR scenarios, because of the approximations we made (recall Section 4.3). We generate the SBM of the basic scenarios using the same technique as in existing work, therefore the results are the same. The solution of the SEQ scenarios is exact: the steady state probability using our stochastic complementation-based approach is the same as if solved directly (with a flat model) [18].

Next, we study how the inaccuracies propagate to the system level. In Figures 11(a) - (d), we vary the failure rates of the *Sensor* and *Hub* components in MIDAS, and the *EM* and *RG* components in GPS between 0.1 and 0.5. In Figures 11(f) - (i), we vary the recovery rates of *Sensor* and *Hub* in MIDAS, and *EM* and *RG* in GPS between 0.2 and 0.8. We observe that results obtained from SHARP closely follow the flat model in these experiments.

We also illustrate that SHARP is useful in highlighting components that are more critical to a system's reliability. For instance, in Figure 11, when we vary the failure rates of *Sensor* and *Hub* between 0.1 and 0.5, system reliabilities obtained from SHARP change by 4% and 0.4%, suggesting that under these conditions *Sensor* is the more critical component. This is corroborated by the flat model.

5.2.2 Effect of Contention Modeling

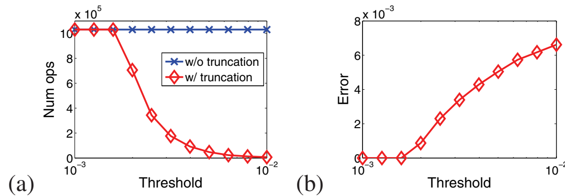
To illustrate the importance of modeling contention in SHARP, we use CS with a single scenario. By increasing the number of clients, we can model a highly-contended system. For example, our results with one server and 8 clients are depicted in Figures 11(e) and (j), where we present the results of using SHARP without contention modeling, SHARP with contention modeling, as well as results from the flat model (which includes contention) as a baseline for comparison. The differences between the results obtained from SHARP without contention modeling and the flat model can be as large as 12% (when the failure rate is 0.2), while the results with contention modeling are much more accurate (the error is generally about 2%, and no larger than 5%, when the failure rate is 0.2). This occurs because, without contention modeling SHARP includes the time spent waiting to be served as processing time, thus overestimating the processing time. In turn, this lowers the reliability because processing a request may trigger a defect in the server that waiting for service does not. Results obtained using other systems are qualitatively similar, and are omitted for brevity.



(a) *Sensor* failure (b) *Hub* failure (c) *EM* failure (d) *RG* failure (e) *Server* failure

(f) *Sensor* failure (g) *Hub* failure (h) *EM* failure (i) *RG* failure (j) *Server* failure

Figure 11: Sensitivity analysis at the system level



(a) (b)

Figure 12: (a) Computational cost of SHARP with and without truncation, (b) Errors caused by model truncation

5.2.3 *Effect of Model Truncation*

In evaluating the effect of truncation (recall Section 4.3.3), we first study the computational cost savings. As a representative illustration, in Figure 12(a), we plot the number of operations needed to solve for the reliability of MIDAS with one *GUI*, *AC*, and *Hub*, and 100 *Gateways*, with each *Gateways* connects to two *Sensors*. i.e., there are 100 instances of the *SensorMeasurement* scenario. In Figure 12(a), we varied the threshold (x-axis, plotted in log-scale), and plotted the number of operations needed to solve SHARP with truncation. We fixed the scenario reliability of *SensorMeasurement* at 0.99, and the completion time at 1. The cost without truncation is our baseline, and can be considered as having a threshold of 0. As we can see from Figure 12(a), the computational cost savings can be significant.

Next, we study the error in reliability estimates when truncation is used. The results are depicted in Figure 12(b). We varied the threshold in the x-axis, and plotted the error in reliability estimates as compared to the results without truncation (y-axis). The error is expectedly smaller for smaller thresholds, The largest error is 0.8% for the threshold of 10^{-2}. Our experiments with model truncation suggest it as an effective way of reducing the analysis space for concurrent systems with minimal losses in accuracy.

6. RELATED WORK

Current literature includes a number of software reliability prediction techniques that are applicable at the architectural level [5, 7, 9, 10, 11, 21, 22, 23, 28, 30]. A comprehensive treatment of these is given in several surveys on the topic [15, 8, 12, 13]. Many of these approaches are influenced by [5], which is one of the earliest works on reliability prediction that considers a system's internal structure using Markov chains. In [5], 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 [5] assumes a sequential system, and most existing approaches, with the notable exception of [7, 22, 28], 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. [7, 22, 28] have taken this approach, in which a state S in a model of a concurrent system is described by $\mathbf{C}$ variables, where $\mathbf{C}$ is the number of components in the system, i.e., $S = (S^1, S^2, \ldots, S^{\mathbf{C}})$. In [7, 28], components are modeled as black-boxes, which are either active or idle, i.e., $S^i = 0$ when $Comp_i$ is idle and $S^i = 1$ when $Comp_i$ 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 [22], where S^i represents the state of $Comp_i$. Specifically, [22] generates component models from scenario models and then generates a system model by combining the component models using parallel composition.

In our earlier work in [4], we estimate the reliability of concurrency systems, modeled as independent scenarios running in parallel. The major differences between this paper and our earlier work are (1) we model the dependencies between scenarios in this paper, while in [4] we assume scenarios are independent; (2) we allow scenarios that are running in parallel synchronize (recall Section 4.3.4); and (3) we explore contention modeling in details (Section 2.2.3), which is first hypothesized as future direction in [4].

Existing approaches are also inflexible with respect to different notions of system failure, e.g., in [5, 10, 11, 21, 28], 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 $Comp_i$ is active, while all other components are idle. A transition from a state S_i to a failure state indicates that $Comp_i$ 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 [22], where the system transitions to a failure state when any active component fails. The work in [7, 9]

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 [25]. However, the system failure description is still limited, assuming that the system fails when any (active) component fails. [28] 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 $Comp_A$ or $Comp_B$ fails). Such notions of system failure can be described within SHARP, by changing the way we compute combination reliability in Section 4.3.5.

Some existing approaches make use of scenario models [10, 22, 30], but they assume a sequential system, with the exception of [22] as described above. For example, in [30] 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, [30] assumes that the probabilities of each scenario occurring are known, which is also not the case in our work.

7. CONCLUSIONS

We presented SHARP, a scalable framework for predicting reliability of concurrent systems. SHARP models concurrency by allowing multiple instances of system scenarios to run simultaneously. We overcame inherent scalability problems by leveraging scenario models and using an approximate hierarchical technique that allowed generation and solution of smaller parts of the overall model at a given time. Our experimental evaluation showed that SHARP's scalability, which is missing from existing techniques, is achieved without significant degradation in the prediction accuracy.

8. ACKNOWLEDGEMENTS

This work is supported by the NSF (award numbers 0509539, 0920612, and 0905665).

9. REFERENCES

[1] F. Baskett et al. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 22(2), 1975.

[2] B. Boehm. Software engineering economics. *IEEE TSE*, 10(1), 1984.

[3] L. Cheung et al. Early prediction of software component reliability. In *ICSE'08*.

[4] L. Cheung et al. SHARP: A scalable approach to architecture-level reliability prediction of concurrent systems. In *QUOVADIS'10*.

[5] R.C. Cheung. A user-oriented software reliability model. *IEEE TSE*, 6(2), 1980.

[6] V. Cortellessa et al. Early reliability assessment of uml based software models. In *WOSP'02*.

[7] R. El-Kharboutly et al. UML-based methodology for reliability analysis of concurrent software applications. *I. J. Comput. Appl.*, 14(4), 2007.

[8] S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE TDSC*, 4(1), 2007.

[9] S. Gokhale and K. Trivedi. Reliability prediction and sensitivity analysis based on software architecture. In *ISSRE 2002*.

[10] K. Goseva-Popstojanova et al. Architectural-level risk analysis using UML. *IEEE TSE*, 29(3), 2003.

[11] K. Goseva-Popstojanova and S. Kamavaram. Software reliability estimation under uncertainty: Generalization of the method of moments. In *HASE 2004*.

[12] K. Goseva-Popstojanova and K. Trivedi. Architecture-based approaches to software reliability prediction. *Intl J. Computer & Mathematics with Applications*, 46(7), 2003.

[13] A. Immonen and E. Niemela. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, Jan 2007.

[14] I. Krka et al. Synthesizing partial component-level behavior models from system specifications. In *ESEC/FSE 2009*.

[15] I. Krka et al. A comprehensive exploration of challenges in architecture-based reliability estimation. *Architecting Dependable Systems*, 6, 2009.

[16] J. Magee and J. Kramer. *Concurrency: State Models And Java Programs*. John Wiley & Sons, 2006.

[17] S. Malek et al. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In *ICSE'07*.

[18] C. D. Meyer. Stochastic complementation, uncoupling Markov chains, and the theory of nearly reducible systems. *SIAM Review*, 31(2), 1989.

[19] OMG. UML 2.2 specification, 2009.

[20] M. Reiser and S. S. Lavenberg. Mean value analysis of closed multichain queueing networks. *J. ACM*, 27(2), 1980.

[21] R. Reussner et al. Reliability prediction for component-based software architectures. *J. of Systems and Software*, 66(3), 2003.

[22] G. Rodrigues et al. Using scenarios to predict the reliability of concurrent component-based software systems. In *FASE 2005*.

[23] R. Roshandel et al. A Bayesian model for predicting reliability of software systems at the architectural level. In *QoSA 2007*.

[24] R. Roshandel, B. Schmerl N. Medvidovic, D. Garlan, and D. Zhang. Understanding tradeoffs among different architectural modeling approaches. In *WICSA 2004*.

[25] W. Stewart. *Probability, Markov Chains, Queues, and Simulation*. Princeton University Press, 2009.

[26] R. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[27] S. Uchitel et al. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM TOSEM*, 13(1), 2004.

[28] W. Wang et al. Architecture-based software reliability modeling. *J. of Systems and Software*, 79(1), 2006.

[29] J. Whittle and P.K. Jayaraman. Synthesizing hierarchical state machines from expressive scenario descriptions. *ACM TOSEM*, 19(3), 2010.

[30] S. Yacoub et al. Scenario-based reliability analysis of component-based software. In *ISSRE'99*.