Survivability Analysis of Networked Systems

S. Jha[1] J. Wing[2]

October 2000

CMU-CS-00-168

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This paper was submitted to the International Conference on Software Engineering 2001, Toronto, May 12-19, 2001.

[1]Computer Sciences Department, University of Wisconsin, Madison, WI 53706.
[2]Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213.

This research is sponsored in part by the Defense Advanced Research Projects Agency and the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, F33615-93-1-1330, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031 and in part by the National Science Foundation under Grant No. CCR-9523972. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory or the U.S. Government.

Keywords: survivability, model checking, reliability analysis, cost analysis, Markov Decision Processes, fault-tolerance, security

Abstract

Survivability is the ability of a system to continue operating despite the presence of abnormal events such as failures and intrusions. Ensuring system survivability has increased in importance as critical infrastructures have become heavily dependent on computers. In this paper we present a systematic method for performing survivability analysis of networked systems. An architect injects failure and intrusion events into a system model and then visualizes the effects of the injected events in the form of *scenario graphs*. Our method enables further global analyses, such as reliability, latency, and cost-benefit analyses, where mathematical techniques used in different domains are combined in a systematic manner. We illustrate our ideas on an abstract model of the United States Payment System.

1 Introduction and Motivation

Increasingly our critical infrastructures are becoming heavily dependent on computers. We see examples of such infrastructures in all domains, including medical, power, telecommunications, and finance. Whereas automation provides society with the advantages of efficient communication and information sharing, the pervasive, continuous use of computers exposes our critical infrastructures to a wider variety and higher likelihood of accidental failures and malicious attacks. Disruption of services caused by such undesired events can have catastrophic effects, including loss of human life.

Survivability is the ability of a system to continue operating in the presence of accidental failures or malicious attacks [7]. We use the term *fault* for both accidental failures (e.g., a disk crash) and malicious attacks (e.g., a denial-of-service attack). The precise semantics of *continuous operation* is application dependent; it is related to critical services that the system provides. For example, check clearing is a critical service of a banking system, and a survivable banking system will continue providing this service despite the presence of faults.

In this paper we present a method for analyzing a networked system for survivability. A *networked system* consists of nodes and links connecting the nodes. Communication between the nodes occurs by passing messages over the links. An *event* in the system can be either a user event (e.g., a user issues a check), an internal event (e.g, a user's account is debited), a communication event (e.g, sending a message between two banks), or a fault (e.g., a bank under a malicious attack). A *service* is associated with a *start event* (e.g., a user issues a check) and an *end event* (e.g., the check clears). The start event and the end event correspond respectively to when "a service is issued" and when a "service is finished."

Our main goal is to provide information to the system architect during the design phase, the early planning stage of the software lifecycle. With this information, the architect can weigh the pros and cons of decisions related to survivability. The method we present in this paper, however, is just as suitable for post facto analysis of existing systems.

Our method is general enough to support many different types of analysis. In this paper we focus on three specific kinds of questions.

Question 1: *What is the effect of a fault?*
Example: Imagine an architect is designing a power grid. He wants to know the effect of an outage of a power plant located in upstate New York on customers living hundreds of miles away in western Pennsylvania.
Answer (Fault-Effect Analysis): Using our method the architect can visualize the global effect of a local fault through a data structure that we call a *scenario graph*. In our method, we automatically generate scenario graphs using model checking.
Question 2: *What is the reliability and latency of a service?* Here, reliability is defined as the probability that a service that has been issued will finish. Latency measures the expected time it takes a service to finish.

Example: Suppose an architect designing a banking system wants to find out the probability that a check issued actually clears.
Answer (Reliability and Latency Analysis): To find the reliability of the banking system with respect to the check clearing service, we query an annotated scenario graph. The architect first identifies a set of "critical" elements in the network, i.e., nodes and links whose failures would have a severe effect on the provision of the service in question. He then assigns probabilities to each fault (i.e., the failure of each node or link). Then, using our method, he can automatically compute both the reliability and latency of the network.
Question 3: *Given cost constraints, which network nodes/links should be upgraded to maximize benefit (e.g., reliability)?*
Example: Suppose an architect is allowed to spend newly allocated funds to upgrade a fraction of the network's links to newer links that are faster and more reliable. Given the constraints imposed by his manager's limited budget, which links should he choose to upgrade to maximize the network's reliability?
Answer (Cost-Benefit Analysis): To perform a cost-benefit analysis, we further extend our annotated scenario graphs with additional cost information related to upgrading the links. We then can automatically compute how to maximize a given benefit given a set of cost constraints.

Survivability analysis is fundamentally different from analysis of properties found in other areas (e.g., algorithm analysis of fault-tolerant distributed systems, reliability analysis of hardware systems, and "security" analysis of computer systems). First, survivability analysis must handle a *broader range of faults* than any of these other areas; we must minimally handle both accidental failures and malicious attacks. To achieve this goal our method allows an architect to incorporate any arbitrary type of fault in the system model; however, we still allow distinctions among faults by assigning different weights (e.g., probability of occurring, cost to repair, etc.) to each fault.

Second, *events may be dependent on each other*, especially fault events. In contrast, for ease of analysis, most work in the fault-tolerant literature makes the *independence assumption*: assume that abnormal events are independent. We cannot make this assumption in analyzing systems for survivability. For example, if a server crashes, then it is easier for a malicious intruder to spoof the crashed server; the chance that an intruder will succeed in spoofing a server depends on the event that the server crashes. Or, if an attacker learns how to compromise one disk of a replicated server, then he can easily compromise the replicas too; the chance of bringing down an entire service depends on the likelihood of success of the original attack. In our method we allow users to express such dependencies. Representing dependence between events allows us to model phenomena such as *correlated attacks*, where local attacks might not succeed, but when they occur in tandem or in succession they can have a severe effect on the system. Distributed denial-of-service attacks is an example of a correlated attack (see CERT advisory `CA-2000-0`). Representing dependence also allows us to handle *cascading effects*, where one fault triggers another, which then triggers another, and so on. While it is cleaner to design a system to avoid cascading effects (e.g., by using a strict locking protocol to avoid cascading aborts in a transactional database), in practice it may be impossible to anticipate faults induced by a system's environment that violates the assumptions made by the system's original designer. Since survivability is of particular concern to those building systems of systems, system architects will have to face the possibility of cascading effects in their analysis.

Third, survivability analysis should also be *service dependent*. For example, the architect for a banking system might choose to focus on the check clearing service as being critical, although the banking system provides other services such as accounting, auditing, and cash distribution; for a different analysis, cash distribution might be the critical service to focus on. Taking into consideration the specific service a system is to provide enables more targeted analysis, which is often amenable to fully automated support. Also a method that focuses the architect's attention on specific services rather than the general system design is likely to be more appreciated and better understood by the end customer (who cares about the reliability of the applications' services). The analyses in our method are all driven by the properties that the architect specifies as they relate to a critical service.

Finally, survivability analysis deals with *multiple dimensions*. It simultaneously deals with functional correctness (modeling the service itself), fault-tolerance (modeling the effects of accidental failures), security (modeling the effects of malicious attacks), reliability (the likelihood of a service finishing), performance (network latency), and cost. To achieve this goal, the analytical approach described in this paper combines several different kind of analysis techniques into one framework.

The next section introduces *constrained Markov Decision Processes* which form the basis for reliability, latency, and cost-benefit analysis. A general overview of our method appears in Section 3. We describe a small example based on the United States Payment System in Section 4, which we use as a running example throughout the remainder of the paper. Section 5 provides additional details related to each step in our method. Section 6 briefly describes a prototype tool *Trishul* that we have implemented based on our method, and briefly describes two case studies that we have performed. Sections 7 and 8 discuss related work and conclusions respectively.

2 Model of Computation

Our formal model is based on *constrained Markov Decision Processes* or simply *CMDP*s. CMPDs are a generalization of Markov chains, where the transition probabilities depend on the past history. CMDPs enable us to model history dependent transition probabilities and provide a framework to perform cost-benefit analysis. Our exposition of CMDPs is based on Altman [2]. A CMDP is 5-tuple $\langle S, A, P, c, d\rangle$ where

- S is a finite state space.

- A is a finite set of actions. For a state $s \in S$, $A(s) \subseteq A$ is the set of actions available at state s.
- P are transition probabilities, where $P_{sas'}$ is the probability of moving from state s to s' if action a is chosen.
- $c : (S \times A) \rightarrow \Re$ is the immediate cost, i.e., $c(s, a)$ denotes the cost of choosing action a at state s. This cost will be related to the value function to be minimized.
- $d : (S \times A) \rightarrow \Re^k$ is a k-dimensional vector of immediate costs. This will be related to cost constraints.

A *Markov Decision Process* (*MDP*) is a CMDP without the last component d.

History at time t (denoted by h_t) is the sequence of states encountered and actions taken up to time t. A *policy* u takes into account the history h_t and determines the next action at time t. Specifically, $u_t(a|h_t)$ is the probability of taking action a given history h_t. A policy u defines a *value function* $V^u : S \rightarrow \Re$, where $V^u(s)$ is the expected cost of the actions taken if the CMPD uses policy u and starts in state s (the cost c is used to define expected cost). The technical definition of V^u can be found in [2]. Analogously, starting in state s let the expected value of the immediate costs d under the policy u be denoted by $D^u(s)$. Since the result of d is a k-dimensional vector, $D^u(s)$ is also a k-dimensional vector of real numbers. Assume that we are also given a k-dimensional vector $C = (c_1, \cdots, c_k)$, where c_i is the cost constraint on the i-th component of $D^u(s)$. Our aim is to find a policy that minimizes the value function V^u given the constraint imposed by the vector C, or

> Given an initial state $s_0 \in S$, find a policy u that minimizes $V^u(s_0)$ subject to $D^u(s_0) \leq C$.

Remark: Do not confuse a Markov process with a Markov policy, which is a policy where the probability of an action depends only on the current state of the CMDP and not the entire history.

Example 2.1 Imagine a bakery where there can be at most 10 customers waiting at any time. At each time the bakery manager has the option of having one or two servers behind the counter. The state of the CMDP corresponds to the number of servers behind the counter and the number of customers waiting. The action at each state is to decide on how many servers should be behind the counter. In Figure 1 we show a few transitions. Consider the transition from state (S=1, C=m) to (S=2, C=m-1). The action label $a = 2$ on the transition indicates that the manager decided to switch to two servers behind the counter. The probability that a waiting customer leaves with his/her order is 0.5 or 0.75 depending on whether there are one or two servers behind the counter. Notice that the probability that a customer gets serviced is higher when there are two servers behind the counter. Therefore, the transition from state (S=1, C=m) to (S=2, C=m-1) has probability 0.75. The rest of the transitions have a similar

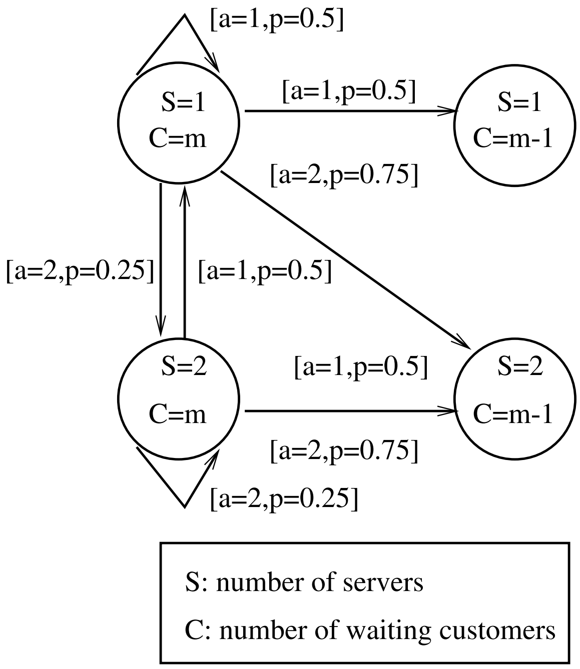

Figure 1: A Bakery

explanation. Given a state and an action, the probability that a customer is serviced in the next time period determines the cost function c. For example, the cost of the state action pair $\langle$ (S=1, C=m), a=1 $\rangle$ is -0.5 because if an action a=1 is chosen from the state the expected number of customers that are serviced during the next time step is 0.5. Notice that the negative of the cost determines the throughput, i.e., the expected number of customers that are serviced in the next time period. The number of servers behind the counter determines the cost function d, i.e., two servers cost more than one. The aim of the manager is to maximize expected throughput (or minimize expected cost related to c) given a constraint on the wages of the servers. Achieving this goal can be easily seen as a problem of value maximization under cost constraints and naturally fits the CMDP framework. The optimal policy for this CMDP will indicate to the bakery manager when to change the number of servers behind the counter.

3 The General Method

In this section we provide a brief overview of our method; Section 5 gives more details about the techniques we use and our implementation. In steps 1, 2, and 3 we model the network, inject faults into our model, and specify survivability-related properties. Then in steps 4, 5, and 6 we analyze the effects of faults, perform reliability and latency analysis, and do cost-benefit analysis—to parallel answering the three kinds of questions posed in the introduction.

3.1 *Step 1:* Model the Network

First, the architect models a networked system, which can be done using one of many formalisms. We choose to use *state machines* and we use them to model both network nodes and links. We use shared variables to represent communication between the state machines.

3.2 *Step 2:* Inject Faults

Both links and nodes may be faulty. With our state machine model of the networked system, we need not make a distinction between nodes and links when considering faults. That is, a link is simply a node that passes data between two other nodes. Injecting a fault then requires first representing that a fault has occurred and then determining the behavior of the faulty node for each kind of fault that may occur. The exact behavior of a faulty node, specified by the architect, depends on the application.

To represent faults in our method, for each state machine representing a node, we introduce a special variable called `fault`, which can range over a user-specified set of symbolic values. For example, the following declaration states that there are three modes of operation for a node, representing whether it is in the normal mode of operation, failed, or compromised by an intruder.

`fault:` { `normal, failed, intruded` }

Given this simple representation, we can then choose to specify the precise behavior of the node in each mode of operation. For example, for any given state we can specify that the machine makes a transition from the normal mode of operation to one of the abnormal modes (`failed` or `intruded`) and further specify what state the machine is in once such a transition occurs. We also have the option of leaving state transitions completely nondeterministic.

3.3 *Step 3:* Specify Survivability Properties

The architect specifies properties related to survivability using some kind of formal logic. In our method, we use a temporal logic called *Computation Tree Logic* (*CTL*), but other temporal logics such as *Linear Time Logic* [15] would also be appropriate.

In this paper, we focus on two classes of survivability properties: *fault* and *service* related. The first class captures properties of the networked system under scrutiny when it enters a faulty state. The second class captures properties specific to the system's services.

3.4 *Step 4:* Generate Scenario Graphs

Given a state machine model, M, of the networked system (with injected faults) and a survivability property, P, we then generate a *scenario graph*, which is a concise representation of a set of traces of M with respect to P. For fault properties, a *fault scenario graph* represents all system traces that end in a faulty state; for service properties, a *service success (fail) scenario graph* represents all system traces in which an issued service successfully finishes (fails to finish). An architect can use scenario graphs to visualize the effects of injected faults on a certain service. (In the operational security literature, scenario graphs are similar to *attack state graphs* [13].)

3.5 *Step 5:* Reliability and Latency Analysis

Once we have a scenario graph, we can perform further analyses, such as reliability and latency analysis. First, the architect specifies the probabilities of certain events of interest, such as faults, in the system. Since we do not assume independence of events, we use a formalism based on *Bayesian networks* [14] to specify the conditional probabilities of the events. We combine the specified probabilities with the scenario graph to obtain an MDP. We can then readily compute reliability and latency by solving for optimal policies using the relevant cost functions c, i.e., for reliability analysis the cost function is identically zero; for latency analysis, it is a function of the times associated with making state transitions.

An advantage of our method is that an architect need not specify probabilities for all events; an MDP can have both probabilistic and nondeterministic transitions.

3.6 *Step 6:* Cost-Benefit Analysis

In this step we transform the MDP from Step 5 into a CMDP. First we enhance the MDP's set of actions A with actions corresponding to decisions that an architect has to make. For example, these additional actions might correspond to upgrading links to produce a more reliable/faster system, and the architect must decide which links to upgrade. Each added action has a cost; the architect wants to simultaneously minimize cost and maximize some benefit (e.g., reliability). Thus, we also associate costs with these actions and provide constraints on these costs (i.e., specify the function d in the definition of CMDPs). The optimal policy corresponding to the CMDP so constructed provides the architect with the optimal decision under the specified cost constraints.

4 Example

We consider a simplified model of the United States Payment System, depicted in Figure 2. There are three levels of institutions: *Federal Reserve Banks* at the top, *money centers* in the middle, and small *banks* at the bottom. If two banks are connected to the same money center, then transactions between them are handled by the money center; there is no need to go through the Federal Reserve Banks. For a detailed description of the system see [11].

To illustrate the architecture, suppose a customer A writes a \$50 check to customer C so that the check has a source address Bank-A and destination address Bank-C. The following steps occur for the issued check to clear:

1· Bank-A and Bank-C are not connected through a money center, so the check is then sent to a money center connected to Bank-A. In this case, let's choose money center MC-1.
2· The check is then transferred to the Federal Reserve Bank closest to MC-1, in this case FRB-2.
3· The check is then transferred to the Federal Reserve Bank that has jurisdiction over Bank-C, in this case FRB-3.
4· The check finally makes it way to Bank-C through the money center MC-3.

In Figure 2 the path of the check is shown using dot-dashed lines.

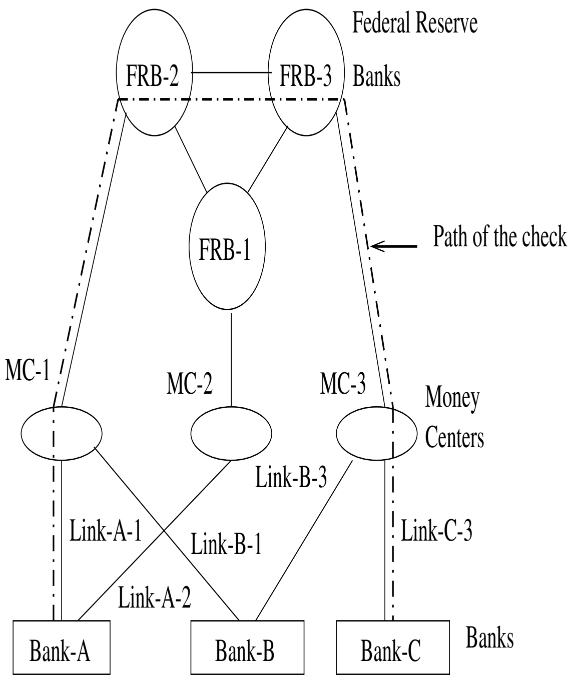

Figure 2: United States Payment System

5 Detailed Description

We now present the details of each step in our method in more detail, illustrating them with the check clearing example.

5.1 *Step 1:* Model the Network

We model each node and link in the system as a finite state machine, and the entire networked system as the composition of these machines. In our implementation, we use the model checker `NuSMV` [1], and hence we use `NuSMV`'s input language to describe the state machines representing a given system. Using this off-the-shelf model checker makes it convenient for us at later steps in our method to perform further global analyses; `NuSMV`'s output lets us automatically derive information that we would otherwise have to reconstruct.

In our banking example, we use state machines to model the banks, the money centers, the Federal Reserve Banks, and the links. Each element in the banking infrastructure corresponds to a `MODULE` description in `NuSMV` and communication is achieved by parameter passing. We make some simplifying assumptions in the model of our system: (1) There is just one user who issues checks; the source and destination address of these checks are decided nondeterministically, i.e., the source address can be banks A, B, or C, and similarly for the destination; (2) There is only one check active at any time, and the exact amount of the check is irrelevant.

5.2 *Step 2:* Inject Faults

Next we inject faults in our model by including a special state variable (`fault`) with each state machine to indicate the mode of operation. We modify the specification of each state machine to take into consideration its faulty modes of operation.

In our banking example, what faults we inject and how we handle them in our model are based on the following assumptions:

- The only network elements that can be faulty are (1) links between the banks and the money centers; and (2) small banks, representing that penetration by a malicious intruder has occurred (i.e., `fault = intruded`). No other links or institutions may become faulty and banks cannot fail accidentally.
- When a link is faulty, it blocks all messages and consequently no message ever reaches the recipient.
- Links may become faulty at any time. Thus, in our finite state machine model of a link, we allow a nondeterministic transition to the state where `fault` is equal to `failed`. The third value `intruded` for the variable `fault` is not used in this case.
- Banks can sense a faulty link and route the checks accordingly.

These assumptions show how we take into consideration the semantics of the application; e.g., we are implicitly assuming that Federal Reserve Banks are impenetrable and links between them are highly reliable and secure.

Our model reflects the following behavior. Under the normal mode of operation, a bank receives a check (nondeterministically issued by the user) with its source address. Depending on the destination address of the issued check, the bank either clears it locally or routes it to the appropriate money center. For example, if a check with source address A and destination address B is issued, then it is sent to the money center MC-1 and then sent to bank B. On the other hand, a check with source address A and destination address C has to clear through the Federal Reserve Banks (as in Figure 2). If a bank is faulty, then checks are routed arbitrarily by the intruder (thereby ignoring the check's destination address). A bank can then at any time nondeterministically transition from the normal mode (`fault=normal`) to the intruded mode (`fault=intruded`). Once the bank is faulty it stays in that state forever.

The precise behavior of a faulty node depends on the application, but two types of behaviors under failure conditions are common. In the case of a *stuck-at fault* the node becomes stuck, i.e., it accepts no input on its channel and consequently produces no output. A node with a *Byzantine fault* exhibits completely nondeterministic behavior, i.e., accepts any inputs and produces arbitrary outputs. A Byzantine fault can also be used to model an intruded node.

5.3 *Step 3:* Specify Survivability Properties

In this step, we specify survivability properties in *CTL*, a logic chosen for convenience since the model checker we use accepts *CTL* specifications. Although *CTL* is a rich logic and allows us to express a variety of properties, we focus on two classes of survivability properties: *fault* and *service* related.

Fault Related Properties
Suppose we want to express the property that *it is not possible for a node N to reach a certain* unsafe *state if the network starts from one of the initial states.* The precise semantics of an unsafe state depends on the application. Let the atomic proposition *unsafe* represent the property that node N is in an unsafe state. We can then express the desired property in *CTL* as follows:

$$\mathbf{AG}(\neg unsafe)$$

which says that *for all states reachable from the set of initial states it is true that we never reach a state where unsafe is true.* The negation of the property is

$$\mathbf{EF}(unsafe)$$

which is true if there exists a state reachable from the initial state where *unsafe* is true; in other words if the network starts in one of the initial states it is possible to reach an unsafe state. The atomic proposition *unsafe* can stand for a property as complex as we desire. It could mean that a certain critical node has entered an undesirable state (e.g., a critical valve is open in a nuclear power plant), or it could mean that a certain unauthorized operation occurred at a critical node. For example, if a node represents a computer protecting a critical resource, it could represent the fact that somebody without the appropriate authority has logged onto the computer. The precise nature of a faulty state depends on the example at hand.

Service Related Properties

Many networked systems are built for distributed applications. For these cases we want to make sure that if a node N issues a service, then the service eventually finishes executing. Let the atomic proposition *start* express that a service was started, and *finished* express that the transaction is finished. The temporal logic formula given below expresses that *for all states where a service starts and all paths starting from that state there exists a state where the service always finishes*, or in other words a *service issued always eventually finishes.*

$$\mathbf{AG}(\mathit{start} \rightarrow \mathbf{AF}(\mathit{finished}))$$

For the banking example, we would like to verify that a check issued is always eventually cleared. This can be expressed in *CTL* as

$$\mathbf{AG}(\mathit{checkIssued} \rightarrow \mathbf{AF}(\mathit{checkCleared}))$$

We can also analyze the effect of a compromised node (say N). Suppose we have modeled the effect of a malicious attack on node N (see discussion on injecting faults). Now we can check whether the desired properties are true in the modified networked system. If the property turns out to be true, the network is resistant to the malicious attack on the node N. This type of analysis is useful in determining vulnerable or critical nodes of a network with respect to a certain service. Using this analysis, if a node is found to be vulnerable or critical for a given service to complete, then the system administrator can deploy sophisticated intrusion detection algorithms for that node or bolster the security infrastructure around it. Thus our analysis can help identify the critical nodes in a networked system and therefore help determine whether it is survivable with respect to desired properties of a given service.

5.4 *Step 4:* Generate Scenario Graphs

We automatically construct scenario graphs via model checking. When a specified property is not true in a given model, a model checker will produce a counterexample, i.e., a trace or a scenario that leads to a final state that does not satisfy the property. (Details of model checking, e.g., see [5], are not needed to understand our method.) We exploit this functionality of model checkers to generate scenario graphs; i.e., a scenario graph is a compact representation of all the traces that are counterexamples of a given property [1]. For example, suppose we want to check whether during the execution of a networked system a certain event (e.g., buffer overflow) never happens. If the property is not true (i.e., buffer overflow can happen), the scenario graph encapsulates all sequences of states and transitions that lead the system to a state where a buffer overflow occurs.

Scenario graphs depict ways in which a network can enter an unsafe state or ways in which a service can fail to finish. Scenario graphs encapsulate the

[1] Identifying the fragment of CTL such that all counterexamples to the formulas in this fragment form a finite graph is not a trivial problem. Fortunately, the two types of formulas we consider have this property.

effect of local faults on the global behavior of the network. If the architect models malicious attacks, the scenario graph is a compact representation of all the threat scenarios of the network, i.e., a set of sequences of intruder actions that lead the network to an unsafe state.

Fault Scenario Graphs
Recall that we can express the property of the absence of an unsafe reachable state as:

$$\mathbf{AG}(\neg \mathit{unsafe})$$

If this formula is not true, it means that there are states that are reachable from the initial state that are faulty.

We briefly describe the construction of a scenario graph. Assume that we are trying to verify using model checking whether the specification of the network satisfies $\mathbf{AG}(\neg \mathit{unsafe})$. Usually, the first step in model checking is to determine the set of states S_r that are reachable from the initial state. After having determined the set of reachable states, the algorithm determines the set of reachable states S_{unsafe} that have a path to an unsafe state. The set of states S_{unsafe} is computed using fix-point equations [5]. Let R be the transition relation of the network, i.e., $(s, s') \in R$ iff there is a transition from state s to s' in the network. By restricting the domain and range of R to S_{unsafe} we obtain a transition relation R_f that encapsulates the edges of the scenario graph. Therefore, the scenario graph is $G = (S_{unsafe}, R_f)$, where S_{unsafe} and R_f represent the nodes and edges of the graph respectively. In symbolic model checkers, like `NuSMV`, the transition relation and sets of states are represented using *binary decision diagrams* (BDDs) [4], a compact representation for boolean functions. All the operations described above can be easily performed using BDDs. The BDD for the transition relation R_f is a succinct representation of the edges of the fault scenario graph. Since BDDs are capable of representing a large number of nodes, very large scenario graphs can be computed using our method.

Service Success/Fail Scenario Graphs
In the case of services, we are interested in verifying that every service started always eventually finishes. Recall that we express this property in *CTL* as

$$\mathbf{AG}(\mathit{start} \rightarrow \mathbf{AF}(\mathit{finished}))$$

Since we allow several nodes to be faulty, in our experience we find that most of the time this property fails to hold. Thus more interestingly, during the model checking procedure, we derive two graphs: a *service success scenario graph* and a *service fail scenario graph*. The success scenario graph captures all the traces in which the service finishes; the fail scenario graph, all the traces in which the service fails to finish. These scenario graphs are constructed using a procedure similar to the one described for the fault scenario graphs.

In our banking example, issuing a check corresponds to the start of a service. The scenario graph shown in Figure 3 shows the effect of link failures on the check clearing service for a check issued with source address Bank-A and destination address Bank-C (the start event is labeled as issueCheck(Bank-A,Bank-C) in the figure). The event corresponding to sending a check from location

L1 to L2 is denoted as sendCheck(L1,L2). The predicates up(Link-A-2) and down(Link-A-2) indicate whether Link-A-2 is up or down. Recall that we allow links to fail nondeterministically. Therefore, an event sendCheck(Bank-A,MC-2) is performed only if Link-A-2 is up, i.e., up(Link-A-2) is the pre-condition for event sendCheck(Bank-A,MC-2). If a pre-condition is not shown, it is assumed to be true. Note that a fault in a link can also be construed as an intruder taking over the link and shutting it down. From the graph it is easy to see that a check clears if Link-A-2 and Link-C-3 are up, or if Link-A-2 is down and Link-A-1 and Link-C-3 are up. We modified the model checker `NuSMV` to produce such scenario graphs automatically.

For realistic examples scenario graphs can be extremely large. Therefore, it is not feasible to enumerate all the scenarios or traces corresponding to a scenario graph. We developed a querying process by which an architect can select a subset of scenarios. First an architect identifies events of interest in the network; then, using these events as alphabet symbols, the architect provides a regular expression to specify the traces of interest. Consider the scenario graph shown in Figure 3 and this regular expression for the alphabet Σ:

$$\Sigma^\star \text{ sendCheck(FRB-2,FRB-3) } \Sigma^\star$$

This query captures the architect's interest in all traces where the check is transferred from FRB-2 to FRB-3, as denoted by the event sendCheck(FRB-2,FRB-3). A trace that satisfies the regular expression is shown by a dotted line in Figure 3.

5.5 *Step 5:* Reliability and Latency Analysis

Once we have generated scenario graphs, we can perform reliability and latency analysis. First, we need to incorporate probabilities of various events into a given scenario graph to produce an MDP; then using the MDP we compute reliability and latency by calculating the value function corresponding to the optimal policy.

We first explain this analysis using the banking example and then provide a formal explanation. Let the boolean state variable $A1$ indicate whether Link-A-1 is up, so $\overline{A1}$ corresponds to Link-A-1's being down. Analogously, $A2$ and $C3$ are the boolean variables corresponding to Link-A-2's and Link-C-3's being up. In general an event will be associated with a boolean variable and the negation of the variable will denote that the event did not occur; we will use the boolean variable and the event it represents synonymously, e.g., event $A1$ corresponds to Link-A-1's being up.

We now explain how we handle dependencies between events. Assume that event $A2$ is dependent on $A1$ and there are no other dependencies. Let $P(A1)$ and $P(C3)$ both be $\frac{1}{2}$ where $P(A1)$ and $P(C3)$ are the probabilities of Link-A-1 and Link-C-3 being up. The probability of event $A2$ depends on the event $A1$, and we give its conditional probability as

$$\begin{aligned} P(A2|A1) &= \tfrac{1}{2} \\ P(A2|\overline{A1}) &= \tfrac{1}{4} \end{aligned}$$

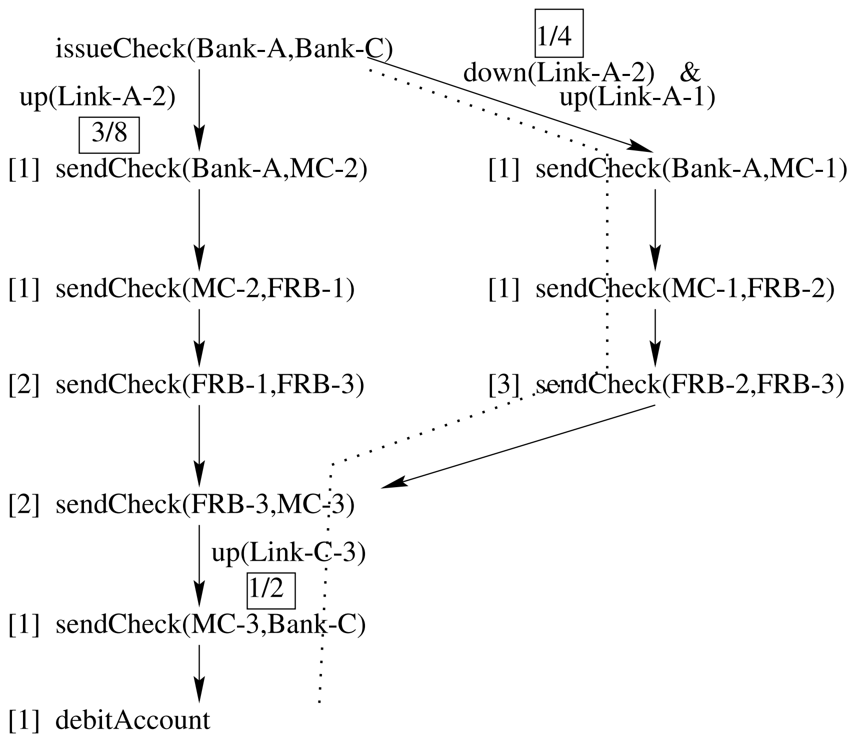

Figure 3: A Simple Scenario Graph

reflecting that if Link-A-1 is down, it is more likely that Link-A-2 will go down. In general, if an event A depends on the set of events $\{A_1, \cdots, A_k\}$, then the probability of A has to specified for each possible case in the set of events $\{A_1, \cdots, A_k\}$. For example, if A depends on $\{A_1, A_2\}$, then $P(A|A_1 \wedge A_2)$, $P(A|A_1 \wedge \overline{A_2})$, $P(A|\overline{A_1} \wedge A_2)$, and $P(A|\overline{A_1} \wedge \overline{A_2})$ have to be specified. This technique is the Bayesian network formalism.

In our example, first we have to compute the probability of the two events $A2$ and $\overline{A2} \wedge A1$. These events correspond to events up(Link-A-2) and down(Link-A-2) & up(Link-A-1) in the scenario graph. The probabilities for these events are derived below.

$$
\begin{aligned}
P(A2) &= P(A2|\overline{A1})P(\overline{A1}) + P(A2|A1)P(A1) \\
&= \frac{1}{4}(1 - \frac{1}{2}) + \frac{1}{2} \cdot \frac{1}{2} \\
&= \frac{3}{8} \\
P(\overline{A2} \wedge A1) &= P(\overline{A2}|A1)P(A1) \\
&= (1 - P(A2|A1))P(A1) \\
&= \frac{1}{4}
\end{aligned}
$$

We add these probabilities (shown inside little boxes) to the relevant edges of the scenario graph in Figure 3. Since we might assign probabilities to only some events (typically faults) and not others, we obtain a structure that has a combination of purely nondeterministic and probabilistic transitions. In our banking example, the architect might assign probabilities only to events corresponding to faults; the user of the banking system still nondeterministically issues checks. Intuitively, nondeterministic transitions are actions of the environment or the user, and probabilistic transitions correspond to moves of the adversary. If we view nondeterministic transitions as actions, the structure obtained after incorporating probabilities into the scenario graph is an MDP. (In the distributed algorithms literature [12], structures that have a combination of nondeterministic and probabilistic transitions are called *concurrent probabilistic systems.*)

We now explain the algorithm to compute reliability and latency by first considering a property about services. Recall that we are interested in the following property:

$$\mathbf{AG}(\mathit{start} \rightarrow \mathbf{AF}(\mathit{finished}))$$

Let G be the service success scenario graph corresponding to this property. Suppose each edge $s \rightarrow s'$ in G has a cost $c(s \rightarrow s')$ associated with it. Now the goal of the environment, which is assumed to be malicious, is to devise an optimal policy or equivalently choose nondeterministic transitions in order to minimize reliability or maximize latency. A value function V assigns a value $V(s)$ for each state s in the scenario graph. Next we describe an algorithm to compute the value function $V^\star$ corresponding to this optimal policy. This algorithm is called *policy iteration* in the MDP literature. (Later we explain how the value function can be interpreted as worst case reliability or latency.) In the initial step, $V(s) = 1$ for all the states that satisfy the property *finished*, and for all other states s we assume that $V(s) = 0$. A state s is called *probabilistic* if transitions from that state are probabilistic. A state is called *nondeterministic* if it is not probabilistic. For all states s that satisfy *finished* the value $V(s)$ is always 1; and for all other states the value function is updated as follows:

- If s is nondeterministic then

$$V(s) \;=\; \min_{s' \in succ(s)} c(s \rightarrow s') + V(s')$$

- If s is probabilistic then

$$V(s) \;=\; \sum_{s' \in succ(s)} p(s,s')(c(s \rightarrow s') + V(s'))$$

In the equations given above, $succ(s)$ is the set of successors of state s and $p(s,s')$ is the probability of a transition from state s to s'. Intuitively speaking, a nondeterministic move corresponds to the environment choosing an action to minimize the value. The value of a probabilistic state is the expected value of the value of its successors. Starting from the initial state, the value function V is updated according to the equations given above until convergence.

After the above algorithm converges, we end up with the desired value function $V^\star$. Let s_0 be the initial state of the scenario graph.

- If the cost, c, associated with the edges is *zero*, then $V^\star(s_0)$ is the *worst case reliability metric* corresponding to the given property, i.e., the worst case probability that if a service is issued it will eventually finish.
- If the cost, c, associated with the edges correspond to negative of the latency, then the value $-V^\star(s_0)$ corresponds to the *worst case latency* of the service, i.e., the worst case expected finishing time of a service. Notice that in this setting minimizing cost corresponds to maximizing latency.

Consider the scenario graph shown in Figure 3. The worst case reliability using our algorithm is $(\frac{1}{2} \cdot \frac{3}{8}) + (\frac{1}{2} \cdot \frac{1}{4}) = \frac{5}{16}$. That is, the worst case probability that a check issued by Bank-A on Bank-C is cleared is $\frac{5}{16}$. Latency in days for all the events is shown in Figure 3 inside square brackets, e.g., latency of the event sendCheck(FRB-3,MC-3) is 2 days. The worst case latency using our algorithm computes to be 4 days.

5.6 *Step 6:* Cost-Benefit Analysis

Finally, we add more cost information and extend our MDP to a CMDP. Again, we will explain this analysis using the running example first. Suppose an architect wants to upgrade some links to improve the overall robustness of the system. Three links Link-A-1, Link-A-2, and Link-C-3 are candidates for being upgraded. Assume that if Link-A-1 and Link-C-3 are upgraded then the probabilities $P(A1)$ and $P(C3)$ increase to $\frac{3}{4}$ respectively. If Link-A-2 is upgraded then the probability of Link-A-2 being up is given below.

$$\begin{aligned} P(A2|A1) &= \tfrac{3}{4} \\ P(A2|\overline{A1}) &= \tfrac{3}{8} \end{aligned}$$

If the links are not upgraded, then the probabilities do not change. In addition to the actions corresponding to the nondeterministic transitions, three extra actions (corresponding to upgrading Link-A-1, Link-A-2, and Link-C-3) are added to the action set, A, of the MDP that was constructed previously. Moreover, assume that the architect has a cost constraint so that only two links can be upgraded. Therefore, in this case we obtain a CMDP, where the cost of upgrading the links is expressed by the cost function d (Section 2). Algorithms for finding optimal policies in the case of *CMDP*s exist but are complicated [2]. Fortunately, our problem is easier because the decisions to upgrade the links are *static*, i.e., do not depend on the state of the system. In this case the optimal decision can be found by solving an auxiliary integer programming problem. With each of the three links Link-A-1, Link-A-2, and Link-C-3 we associate 0-1 variables x_{A1}, x_{A2} and x_{C3}. Intuitively, $x_{A1} = 1$ indicates that Link-A-1 has been upgraded. Now the worst case reliability is a function of x_{A1}, x_{A2}, and x_{C3}. We denote this by $Rel(x_{A1}, x_{A2}, x_{C3})$. Our aim is to maximize the worst case reliability $Rel(x_{A1}, x_{A2}, x_{C3})$ subject to the constraint that at most two links can be upgraded, i.e.,

$$x_{A1} + x_{A2} + x_{C3} \leq 2$$

This is a non-linear integer programming problem. Although the problem in its full generality is hard, several heuristics for solving these class of problems have been studied [16]. For our example, Figure 4 lists the worst-case reliability for the three possible cases. It is clear that the best option is to upgrade Link-A-1 and Link-C-3.

$x_{A1} = 1$ and $x_{A2} = 1$	$\frac{7}{16}$
$x_{A1} = 1$ and $x_{C3} = 1$	$\frac{39}{64}$
$x_{A2} = 1$ and $x_{C3} = 1$	$\frac{9}{16}$

Figure 4: Table of Three Cases

6 Status

We built a tool *Trishul* based on the ideas presented in this paper. We implemented all the basic algorithms. We are finishing the graph visualization component and a customized editor.

We also finished two major case studies: an extended banking system and a bond trading floor. Our model of the banking system is much more complicated than the simplified example presented in this paper. For example, we handle protocols such as *Fedwire* and *SWIFT* (used for transfer of funds and transmitting financial messages respectively) that we did not show here[2]. The entire banking system model is about 2,000 lines of `NuSMV` code. The scenario graph has about 25,000 nodes and computing reliability and latency takes only a few minutes.

We also modeled and analyzed the system architecture of a bond trading floor of a major investment company in New York [3]. The model is about 10,000 lines of `NuSMV` code and has about 100 state variables. Our tool found several attacks. Two of these attacks were considered serious by the architects. One attack enabled a junior trader to acquire a head trader's password. The second attack enabled a junior trader to obtain sensitive information from the company's database, i.e., a junior trader could find out the nature of the pending trades. Not surprisingly, we gained valuable experience during this case study. The most cumbersome part of the modeling process was the fault injection phase because the nature of the faults injected was heavily dependent on the security policies and technologies deployed at that node. We plan to automate the fault injection process in the near future.

7 Related Work

Survivability is a fairly new discipline, and viewed by many as distinct from the traditional areas of security and fault-tolerance [7]. The Software Engineering Institute uses a method for analyzing the survivability of network architectures (called SNA) and conducted a case study on a system for medical information management [8]. The SNA methodology is informal and meant to provide general recommendations of "best practices" to an organization on how to make their systems more secure or more reliable. In contrast, our method is formal and leverages off automatic verification techniques such as model checking. Other papers on survivability can be found in the *Proceedings of the Information Survivability Workshop* [10].

Research on *operational security* by Ortolo, Deswarte, and Kaaniche [13] is closest to Step 4 of our method. Their attack state graphs are similar to our scenario graphs. However, since we use symbolic model checking to generate scenario graphs, represented by BDDs, we can handle extremely large graphs. Moreover, in our method a scenario graph corresponds to a particular service; in contrast their graph corresponds to a global model of the entire system. We are currently investigating how to incorporate concepts and analysis techniques presented in their paper [13]. into our method.

Fault injection is a well-known technique in the fault tolerance community.

[2] We thank Joe Ahearn of CSFB for clarifying the details of these two protocols.

[3] Due to the propriety nature of the case study we are in the process of "sanitizing" the model so we can publish the results at a later date.

We allow the designer to specify any kind of fault, and thus we can consider a wider class of faults. Moreover, we allow fault events to be dependent and thus can model correlated attacks. Computing reliability is also not new. There is a vast amount of literature on verifying probabilistic systems and our algorithm for computing reliability draws on this work [6]. The novelty in our work is the systematic combination of different techniques into one method.

8 Summary of Contributions and Future Work

Survivability has become increasingly important with society's increased dependence on critical infrastructures run by computers. In this paper, we presented in a single framework a systematic method for analyzing a networked system for survivability. A fundamental contribution of our work is to use constrained Markov Decision Processes as the sole underlying mathematical model for this framework. A second contribution is the natural integration of a set of analysis techniques from disparate communities into this framework: model checking (popular in computer-aided verification) , Bayesian network analysis (popular in artificial intelligence), probabilistic analysis (popular in hybrid systems and queueing systems), and cost-benefit analysis (popular in decision theory). In combination, these techniques let us provide a multi-faceted view of the networked system. This *holistic view* of a system is at the core of achieving survivability for the system's critical services.

There are several directions for future work. First, we plan to finish the prototype tool that supports our method. We are working on several case studies, including protocols used in an electronic commerce system. Since for real systems, scenario graphs can be very large, we plan to improve the display and query capabilities of our tool so architects can more easily manipulate its output. Finally, to make the fault injection process systematic, we are investigating how best to integrate operational security analysis tools such as COPS [9] into our method.

References

[1] Nusmv: a new symbolic model checker. http://afrodite.itc.it:1024/nusmv/.

[2] E. Altman. *Constrained Markov Decision Processes*. Chapman and Hall, 1998.

[3] D. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, Aug. 1986.

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[6] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of ACM*, 42(4):857–907, 1995.

[7] R. Ellison, D. Fisher, R. Linger, H. Lipson, T. Longstaff, and N. Mead. Survivable network systems: An emerging discpline. Technical Report CMU/SEI-97-153, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, November 1997.

[8] R. Ellison, R. Linger, T. Longstaff, and N. Mead. Survivability network system analysis: A case study. *IEEE Software*, 16/4, July/August 1999.

[9] D. Farmer and E. Spafford. The cops security checker system. In *Proceedings Summer Usenix Conference*, 1990.

[10] In *Information Survivability Workshop, ISW*, October 1998. http://www.cert.org/research/isw98.html.

[11] J. Knight, M. Elder, J. Flinn, and P. Marx. Summaries of three critical infrastructure applications. Technical Report CS-97-27, Department of Computer Science, University of Virginia, Charlottesville, VA 22903, December 1997.

[12] N. Lynch, I. Saias, and R. Segala. Proving time bounds for randomized distributed algorithms. In *Proceedings PODC*, pages 314–323, 1994.

[13] R. Ortalo, Y. Deswarte, and M. Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*, 25/5:633–650, Sept/Oct 1999.

[14] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[15] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Comput. Sci.*, 13:45–60, 1981.

[16] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1998.