Survivability Analysis of Network Specifications*

S. Jha† J. Wing† R. Linger‡ T. Longstaff‡

Abstract

Survivability is the ability of a system to maintain a set of essential services despite the presence of abnormal events such as faults 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 networks. A system architect injects fault and intrusion events into a given specification of a network and then visualizes the effects of the injected events in the form of scenario graphs. *In our method, we automatically generate scenario graphs using model checking. Our method enables further global analysis, such as reliability analysis, 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

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 failures and intrusions. 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 maintain essential services in the presence of undesired events. These events include malicious attacks and intrusions, and otherwise benign, but unanticipated failures [6].

In this paper we address the issue of survivability in the context of a highly distributed network of nodes. We are particularly interested in the global behavior of an asynchronous network of concurrently computing nodes and in the properties that hold of the entire network. In general, survivability is a property of the *entire network* and not just a property of a single node. Our goal is to find techniques for analyzing networks of nodes for survivability using the specifications of the individual nodes, interconnections between nodes, and of faults and intrusions to which the system is susceptible.

We believe that survivability analysis is fundamentally different from analysis techniques found in other areas (e.g., verification and analysis for fault tolerance, and reliability analysis). First, survivability analysis takes a *service view* of the system, i.e., the analysis focuses on certain key services provided by the system. Second, survivability analysis deals with *multiple dimensions* of the system with respect to a service, i.e., analysis simultaneously deals with fault tolerance, functional correctness, and reliability issues. In an abstract sense, survivability takes a *holistic view* of the system and hence is interested in a conglomerate of properties rather than an isolated one. To achieve this goal, the analytical approach described in this paper combines many different kind of analysis techniques. There are two important issues that any technique for analyzing survivability must address. Faults and intrusions should be allowed to be present simultaneously. There are many attacks that only materialize because of the interplay between faults and intrusions. For ease of analysis, the *independence assumption* (assuming that two abnormal events are independent) is prevalent in the fault-tolerant and reliability literature. 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. Therefore, the chance that an intruder will succeed in spoofing a

*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.

†Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. emails: {sjha,wing}@cs.cmu.edu

‡Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213. emails: {rlinger,tal}@sei.cmu.edu

server depends on the event that the server crashes. In our method we allow users to express such dependencies. Introducing dependence between events gives rise to phenomena such as *correlated attacks* and *cascading effects*, where local attacks might not succeed, but when these attacks occur in tandem or succession 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`). Our framework addresses both issues.

Our main goal is to provide valuable information to the system architect during the design phase, and enable him/her to make important decisions at an early phase in the software life-cycle. Using our method a designer can visualize the global effect of local faults and intrusions. We provide the user with this information through a data structure called *scenario graphs*, which we automatically generated using model checking. By assigning probabilities to faults and intrusions of local nodes an architect can compute the reliability of the entire network. Using our method, the architect can also easily identify critical nodes in the network, i.e., where their failure would have a severe effect on the reliability of the network.

We use model checking for a very specific purpose in our method. Model checking is a technique for proving properties (expressed in a logic called the *Computation Tree Logic* or *CTL*) about specifications of reactive systems. We do not provide details of model checking here. Interested readers can refer to [4] for background material on model checking. The lack of knowledge about model checking will not impair the reader's understanding about the entire method. In this paper we use the model checker `NuSMV` [1].

The next section gives a general overview of our method. We describe a small example based on the United States Payment System in Section 3. We use this system as a running example throughout the remainder of the paper. Section 4 provides additional details related to each step in our method. Section 5 briefly describes a prototype tool *Trishul* that we are implementing based on our method, and describes some case studies that we have performed. Sections 6 and 7 discuss related work and conclusions respectively.

2 The General Methodology

In this section, we provide a brief overview of the general method proposed in this paper. We provide a detailed description of each step in Section 4.

2.1 Modeling the Network

First, we derive a finite state model from the specification of a network's architecture. We assume that nodes are described using a *stimulus-response* or *state machine* model. We model a network as a set of concurrently executing finite state machines. Each node has a set of input channels and a set of output channels. We associate finite queues with the input and output channels. When an input arrives at a channel, it is appended to the associated queue. Similarly, when a system processes an output, it appends it to the relevant queue. A node can be in one of a finite set of states. In any given state, a node receives inputs from queues associated with a set of input channels, transitions to a state depending on the data it receives, and then outputs data on queues associated with a set of output channels. A network is a set of nodes and a set of *interconnections* or *couplings*. An interconnection is simply a pairing of an input channel to an output channel. The general techniques presented here can be applied to any specification language capable of modeling these basic primitives of distributed systems. In our work, we use the input language of the model checker `NuSMV` to specify our example network; using this model checker makes it convenient for us at later steps in our method when for additional global analysis, we need to derive information from `NuSMV`'s output.

2.2 Injecting Faults and Intrusions

By our model of the network, we need not make a distinction between nodes and links when considering failures. That is, a link is simply a node that passes data between two other nodes. Both links and nodes may be faulty or be under attack.

For each node, the architect needs to decide the behavior of the node when a failure occurs. The exact behavior of a faulty or compromised node depends on the specific example. In practice, the nature of faults and intrusions that are injected into a node depends on the security policies and technologies deployed at that node. The specific details depend on the system being modeled.

For each module in the `NuSMV` specification that models a node, we introduce a special variable called `fault` that indicates whether a node is in the normal mode of operation, faulty, or compromised by an intruder. This special variable can have as many symbolic values as the user desires. For instance, the following definition in `NuSMV` states that there are three modes of operation for a node.

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

The user specifies the actual behavior of the node in each mode of operation. Transitions between various modes can be specified by the user or be completely non-deterministic. For example, a node can transition from the normal mode of operation to one of the abnormal modes (`fault` or `intruded` in our example) at any time. Figure 1 illustrates transitions between various modes of operation.

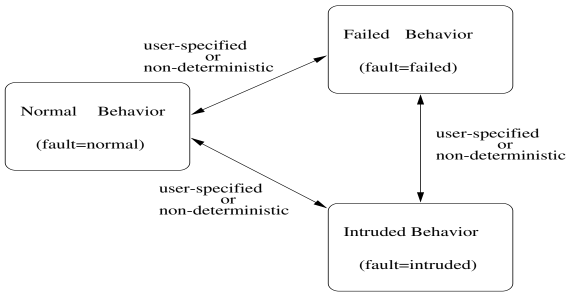

Figure 1. Transitions between different modes

2.3 Expressing Survivability Properties

Survivability properties in our methodology are expressed in the temporal logic *Computation Tree Logic* (*CTL*). We selected the Computation Tree Logic (*CTL*) because the tool we use accepts specifications in that logic. Our method would also work with other temporal logics (such as *Linear Time Logic* or *LTL* [13]). In this paper, we focus on two classes of survivability properties: *fault detection* and *transactional*. The first class of properties expresses whether the network under scrutiny can enter a faulty state. The second class of properties are related to the specific system services.

2.4 Generating Scenario Graphs

We first perform model checking to verify properties about the network. If a certain property turns out to be false, then we output a *scenario graph*. For example, if the property expresses that the network can enter a certain faulty state starting from the initial state, the model checker will output a scenario graph that encapsulates network behaviors that start in an initial state and lead to a faulty final state. A scenario graph is a compact representation of all the traces that are counterexamples of a given property. For example, suppose we want to check whether during the operation of a network a certain event (e.g., buffer overflow) never happens. If the property is not true (i.e., buffer flow can happen), the scenario graph encapsulates all sequences of states and transitions that lead the network to a state where a buffer flow occurs. We had to modify the model checker `NuSMV` to produce scenario graphs since this information is computed internally and not stored. We are still building tools to display the graphs to a system architect in a visually pleasing manner. In the operational security literature, scenario graphs are similar to *attack state graphs* [11].

2.5 Additional Analysis

Once we have a scenario graph, we can perform further analysis. In this paper we describe two kinds: *symbolic analysis* and *reliability analysis*.

Symbolic analysis
In this step the designer assigns symbolic probabilities, such as `high` and `low`, to events of interest (generally faults and intrusions of nodes and links). Since we do not assume independence of events, we use a formalism based on *Bayesian networks* [12] to specify the probabilities of the events. Each event has a set of events on which it depends. The probability of an event occurring depends on the past history of the set of events on which it depends. This point will become clear when we provide an example later in the paper. We combine this table of symbolic probabilities with the scenario graph, which the designer can then query. For example, the designer can ask for all scenarios that have at least one event with `high` likelihood of occurrence. Our tool then produces only those scenarios that satisfy the query.

Reliability analysis
Here, the designer provides numeric probabilities instead of symbolic ones. Again, we incorporate these probabilities into the scenario graph to obtain a state machine structure that has both non-deterministic and probabilistic transitions. We give an algorithm for computing system reliability on such a structure in Section 4. Later sections will provide more detail on each of these analysis.

3 Example

We consider a simplified model of the United States Payment System, depicted in Figure 2. To illustrate the architecture, we describe what happens when a bank customer deposits a check. For a detailed description of the system and this scenario see [9]. Assume that customer *A* gives a check worth 50 dollars to customer *B*. Let *Bank(A)* and *Bank(B)* denote *A*'s and *B*'s banks respectively. The following steps occur for the check to clear:

1. *B* deposits the check in his bank. If *A* and *B* have the same bank, the check is cleared in-house.

2. *Bank(B)* processes the check and bundles other checks received on the same day and sends it to the branch of Federal Reserve Bank nearest to it. To be concrete, let us assume that the Federal Reserve Bank nearest to *Bank(B)* is the Los Angeles (LA) branch.

3. The LA branch of Federal Reserve Bank sends the check written by *A* to the Federal Reserve Bank nearest *A*'s bank, say the New York (NY) Federal Reserve

Bank. The LA Federal Reserve Bank sends a bundle of checks, including *A*'s check, to the NY Federal Reserve Bank.

4. The NY Federal Reserve Bank processes the checks and sends the checks to the relevant banks. *Bank(A)* receives *A*'s check. NY Federal Reserve Bank debits *Bank(A)*'s account and credits the LA Federal Reserve Bank. The LA Federal Reserve Bank credits *Bank(B)*'s account.

5. *Bank(A)* processes the check and debits customer *A*'s account.

As illustrated in Figure 2, there is one more level between small banks and the Federal Reserve Banks. Institutions at this middle level are called *money centers*. 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 example, suppose a check with source address Bank-A and destination address Bank-C is issued. Bank-A and Bank-C are not connected through a money center, so the check is then sent to the money center MC-1. Assuming that the federal reserve bank FRB-2 is nearest to the money center MC-1, the check is transferred to the federal reserve bank FRB-2. Assuming that Bank-C is in the jurisdiction of the federal reserve bank FRB-3, the check is sent to the federal reserve bank FRB-3, and then makes it way to Bank C through the money center MC-3. We show the path of the check using dot-dashed lines.

4 Detailed Description

This section provides details about each step in our method. We give high-level descriptions of each technique and algorithm.

4.1 Modeling the Network

We model each node and link in the network as a finite state machine. The distributed network is a composition of state machines. We assume that suitable abstraction techniques have been applied to the real network to make it finite state.

In our banking example, nodes corresponding to the banks, the money centers, the federal reserve banks, and the links. Each node in the banking infrastructure corresponds to a `MODULE` description in `NuSMV` and message passing is simulated by parameter passing. We also assume the existence of a user who issues checks. The source and destination address of the checks are decided non-deterministically, i.e., the source address can be banks A, B, or C, and similarly for the destination. For simplicity, we assume that only one check is active at any time, and the exact amount of the check is irrelevant.

4.2 Injecting Faults and Intrusions

Next we inject faults and intrusions in our model. Each node has a special state variable (called fault) associated with it. This state variable indicates the mode of operation of the node. For example, fault=normal and fault=intruded means that the node is in the normal and intruded mode, i.e., compromised by an intruder. We also specify the behavior of the node under each mode of operation.

In our example, we allow only the links between the banks and the money centers to fail and only the banks to be intruded. When a link fails, it blocks all messages and consequently no message ever reaches the recipient. We assume that a link can fail at any time; thus in our specification of a link, we allow a non-deterministic transition to the state where `fault` is equal to `failed`. We also assume that banks can sense a failed link and route the checks accordingly. Under the normal mode of operation, a bank receives a check (non-deterministically 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 (see Figure 2). If a bank is intruded, then checks are routed arbitrarily by the intruder (without paying attention to the destination address of the check). A bank can at any time non-deterministically transition from the normal mode (fault = normal) to the intruded state (fault=intruded). Once the bank is intruded it stays in that state forever.

The precise behavior of a faulty or an intruded node depends on the example, 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 a completely non-deterministic behavior, i.e., accepts any inputs and produces arbitrary or non-deterministic outputs. Byzantine fault can also be used to model an intruded node.

4.3 Expressing Survivability Properties

In this section, we model survivability properties in *CTL*. Although *CTL* is a rich logic and allows us to express a variety of properties, we focus on two classes of survivability properties. The first class is *fault detection properties* and the second is *transaction properties*.

Fault Detection Properties

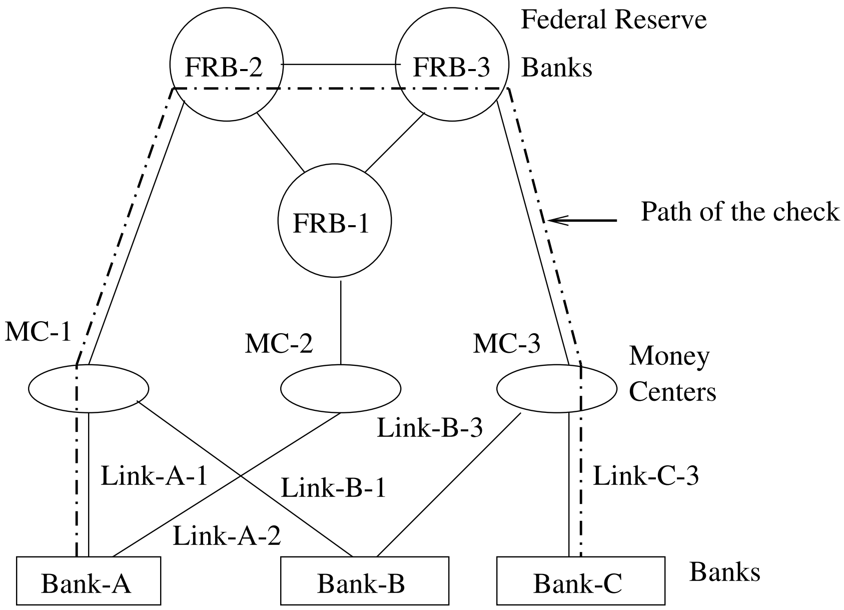

Figure 2. United States Payment System

Suppose we want to express the property that *it is not possible for a node N to reach a certain faulty state if the network starts from one of the initial states*. Let $fault$ represent the atomic property that node N is in a faulty state. We can then express the desired property in *CTL* as follows:

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

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

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

which is true if there exists a state reachable from the initial state where the atomic proposition $fault$ is true; in other words if the network starts in one of the initial states it is possible to reach a state where we have a fault. Suppose the desired property given above is not true in the specified model. Most model checkers will produce a counter-example, i.e., a trace or a scenario through the network that leads the node N to a faulty state. The atomic property $fault$ can be 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). In certain situations it could also mean that a certain unauthorized operation occurred on a critical node. For example, if a node represents a computer with 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.

Transaction Properties

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

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

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

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

We can also analyze the effect of a certain node (say N) being compromised by an intruder on the network. Assume that we have modeled the effect of an intrusion on node N (see discussion on injecting faults and intrusions). Now we can check whether the desired properties are true in the modified network. If the transaction property turns out to be true, the network is resistant to an intruder compromising the node N. This type of analysis will be very 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 transaction to complete, then one can deploy sophisticated intrusion detection algorithms for that node or bolster the security infrastructure around it. Thus our analysis can identify the critical nodes in a networked architecture and provide guidelines on how to make a network more survivable and robust with respect to the mission or service of the system.

4.4 Generating Scenario Graphs

This section describes how we automatically construct scenario graphs. These graphs depict ways in which a network can enter a faulty state or ways in which a transaction can fail to finish. Scenario graphs encapsulate the effect of failures on the global behavior of the network.

Fault scenario graph
Recall that we can express the property of the absence of a faulty reachable state as:

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

Suppose the formula given above is not true. This means that there are states that are reachable from the initial state that are faulty. A *fault scenario graph* encapsulates all the scenarios or traces that drive the initial state of the network to a faulty state. If the architect models intrusions, 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.

We briefly describe the construction of a fault scenario graph. Assume that we are trying to verify using model checking whether the specification of the network satisfies $\mathbf{AG}(\neg \mathit{fault})$. 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, one determines the set of reachable states S_{fault} that have a path to a faulty state. The set of states S_{fault} are computed using fix-point equations [4]. 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_{fault} one obtains a transition relation R_f which encapsulates the edges of the fault scenario graph. Therefore, the fault scenario graph is $G = (S_{fault}, R_f)$, where S_{fault} 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) [3]. 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.

Transaction success/fail scenario graph
In the case of transactions we are interested in verifying that every transaction started always eventually finishes. Recall that we can express this property in *CTL* as:

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

Since we allow several nodes to fail or be intruded in the network, in our experience we find that most of the time the property fails to hold. Thus more interestingly, during the model checking procedure, we derive two graphs: a *transaction success scenario graph* and a *transaction fail scenario graph*. The success scenario graph encapsulates all the traces in which the transaction finishes. The fail scenario graph captures all the traces or scenarios in which the transaction fails to finish. These scenario graphs are constructed using a procedure similar to the one presented for the fault scenario graphs. In our banking example, issuing a check corresponds to a transaction. The scenario graph shown in Figure 3 shows the effect of link failures on a check issued with source address Bank-A and destination address Bank-C (this is labeled as issueCheck(Bank-A,Bank-C) in the figure). The action of sending a check from location L1 to L2 is denoted as sendCheck(L1,L2). 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 non-deterministically. Therefore, an action 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 performing the action sendCheck(Bank-A,MC-2). If a pre-condition is not shown, it is assumed to be true. Note that the failure of the link can also be construed as an intruder taking over the link and shutting it down using a denial-of-service attack. From the graph it is easy to see that a check clears if links Link-A-2 and Link-C-3 are up, or Link-A-2 is down and links Link-A-1 and Link-C-3 are up. We modified the model checker NuSMV to produce such scenario graphs automatically.

4.5 Additional Analysis

This subsection describes two types of further analysis that can be performed on scenario graphs, once they have been generated.

Symbolic Analysis
We first explain this analysis using the banking example and then provide a formal explanation. Assume that $A1$ and $\overline{A1}$ correspond to `Link-A-1` being up and down, respectively. In general $\overline{E}$ will denote the complement of event E. Analogously, $A2$ and $C3$ denote the events corresponding to links Link-A-2 and Link-C-3 being up. Assume that event $A2$ is dependent on $A1$ and there are no other dependencies. We assume probabilities are symbolic, e.g., `high`, `medium`, and `low`. In order to perform computation with symbolic probabilities we need *abstract functions* corresponding to $1 - x$ and $x \star y$. Basically, we are using symbolic probabilities as an abstract domain for the real numbers. A multiplication table for symbolic probabilities is shown in Figure 4. Notice that the result of the abstract multiplication operation can be non-deterministic

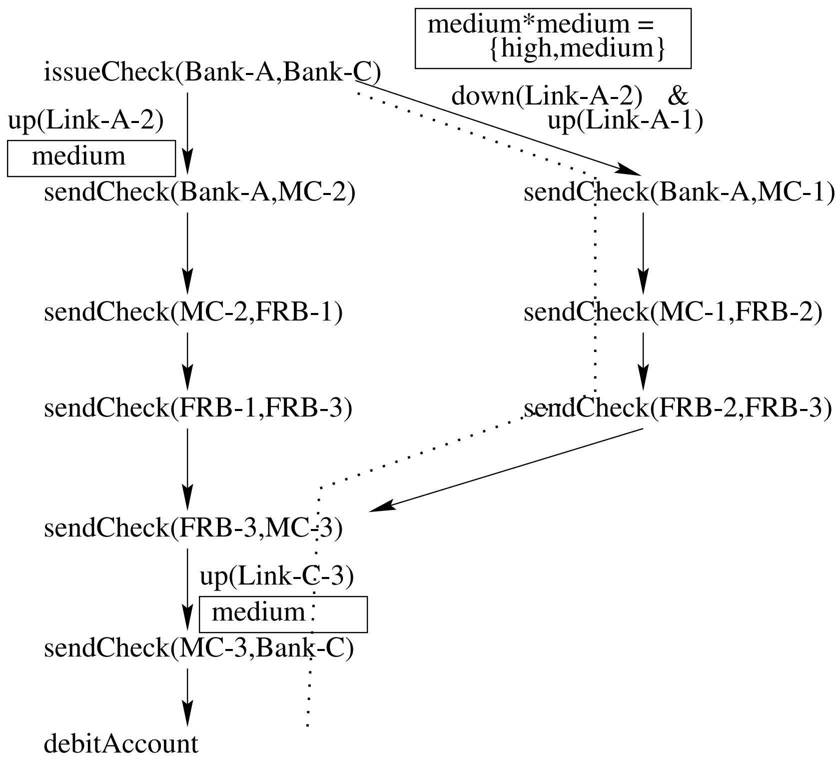

Figure 3. A simple scenario graph

$\star$	high	medium	low
high	high	high	high,medium
medium	high	high, medium	medium, low
low	high, medium	medium, low	low

Figure 4. Abstract multiplication table

(see `high` multiplied by `low`), which is common in abstract interpretation. Abstract complementation operation (denoted by $1-x$) is shown in the equations given below:

$$\begin{aligned} 1-\texttt{high} &= \texttt{low} \\ 1-\texttt{medium} &= \texttt{medium} \\ 1-\texttt{low} &= \texttt{high} \end{aligned}$$

Assume that probabilities $P(A1)$ and $P(C3)$ are both `medium`, where $P(A1)$ and $P(C3)$ are probabilities of links Link-A-1 and Link-C-3 being up. Probability of event $A2$ depends on the event $A1$ and is given in the following table.

Condition	$P(A2)$
$A1$	`medium`
$\bar{A1}$	`low`

The first row represents the probability of event $A2$ (or Link-A-2 being up) given that $A1$ is true (or Link-A-1 is up). Similarly, the second row represents the probability of event $A2$ when Link-A-1 is down. By incorporating these probabilities in the scenario graph we obtain a *annotated scenario graph*, i.e., graphs where some edges are annotated with symbolic expressions composed of `high`, `low`, and `medium`. The symbolic probability of the link Link-A-2 being down and Link-A-1 being up is:

$$\begin{aligned} P(\bar{A2} \wedge A1) &= P(\bar{A2}|A1)P(A1) \\ &= (1-\texttt{medium}) \star \texttt{medium} \\ &= \texttt{medium} \star \texttt{medium} \\ &= \{\texttt{high, medium}\} \end{aligned}$$

These annotations are shown in Figure 3. Now suppose the designer is only interested in traces in the scenario graph that have *at least one event* with probability `high`. We can express this property as the following regular expression R over the alphabet of symbolic probabilities (in this case `high`, `medium`, and `low`):

$$(\Sigma - \{\texttt{high}\})^* \cdot \texttt{high} \cdot \Sigma^*$$

We then convert the regular expression R into a *deterministic finite automata* or *DFA* $D(R)$, and then we compose the $D(R)$ with the annotated scenario graph. Finally, we perform a reachability analysis on the composed scenario graph to eliminate states that do not have a path to the final state of $D(R)$. In our example, we would eliminate states that do not have a path to the final state which has at least one event that has probability `high`. Such a path is shown using dotted lines in Figure 3. Using the technique we just outlined, users obtain a specific *view* of the scenario graph in which they interested. Or, a designer can enumerate threat scenarios that have a high likelihood of occurrence.

We now provide a formal description of the algorithm that we just outlined. Let L be the set of symbolic probabilities. We assume that there are two functions *minus* : $L \to 2^L$ and *mult* : $(L \times L) \to 2^L$. Recall that 2^L denotes the power set of L. We assume that the function *mult* is commutative. We write *minus*(l) and *mult*(l, l') as $1-l$ and $l \star l'$ respectively. Intuitively speaking *minus* and *mult* are abstract counterparts of the operations $1-x$ and multiplication for real numbers. First, we expand the scenario graph G by keeping the history of events with each state. We need to keep the history of events because in general the probability of an event is dependent on the occurrence of previous events. We write each state in the expanded structure, where we keep track of the history, as (s, h), where s is the state of the scenario graph and h is the history. Consider a transition $(s, h) \to (s', h')$ in the scenario graph G and let $E((s, h) \to (s', h'))$ be the set of events that occur on the transition $(s, h) \to (s', h')$. Let E be all the events of interest. In our banking example, E corresponds to link failures and bank intrusions and $E((s, h) \to (s', h'))$ are the failures and intrusions that occur on the transition. The transition $(s, h) \to (s', h')$ is labeled by the following subset of 2^L

$$\left(\prod_{e \in E((s,h)\to(s',h'))} P(e|h) \right) \left(\prod_{e \in E - E(s \to s')} 1 - P(e|h) \right)$$

The probability of an event e given history h (denoted by $P(e|h)$) can be found from the table the user provides. In the product given above we use the abstract multiplication operation for the set of symbolic values L. Therefore, in the annotated scenario graph each edge is labeled with an element of the power set 2^L. Hence the annotated scenario graph can be regarded as a non-deterministic automata over the alphabet L. Let $\mathcal{L}_G$ be the regular language corresponding to the annotated scenario graph. Next, the user provides a regular expression R over the alphabet L. Let $\mathcal{L}_R$ be the regular language corresponding to the regular expression $\mathcal{L}(R)$. We are interested in the intersection of $\mathcal{L}_G$ and $\mathcal{L}_R$, which provides the scenarios of interest. Notice that the intersection of the two languages can be computed using the composition of the annotated scenario graph and the automata corresponding the regular language R.

Reliability Analysis

In this step, numeric probabilities are assigned to the various events. In our example, symbolic probabilities `high`, `medium`, and `low` can be set to 0.75, 0.5, and 0.25. The probabilities of various events are provided by the user in a tabular form as shown earlier. Again, we incorporate these probabilities into the scenario graph. Since we might assign probabilities only to some events (typically faults and intrusions) and not others, we obtain a structure that has a combination of purely non-deterministic and probabilistic transitions. In our banking example, assume a designer assigns probabilities only to the events corresponding to intrusions of banks and link failures. The user, of course, still non-deterministically issues checks. Intuitively, non-deterministic transitions are moves of the environment or the user, and probabilistic transitions correspond to the moves of the adversary. These structures are called *concurrent probabilistic systems* in the distributed algorithms literature [10].

We now explain the algorithm to compute reliability by first considering a property about transactions. Assume that we are interested in the following property:

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

Let G be the transaction success scenario graph corresponding to the property. For every state s in the scenario graph G, we assign a *value* $V(s)$. We refer to V as the *value function*. 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 *non-deterministic* if it is not probabilistic. For all states s that satisfy *finished* the value $V(s)$ is always 1, for all other states the value function is updated as follows: If s is non-deterministic then we update the value function $V(s)$ using the following equation:

$$V(s) = \min_{s' \in \mathit{succ}(s)} V(s')$$

If s is probabilistic we update the value function using the following equation:

$$V(s) = \sum_{s' \in \mathit{succ}(s)} p(s, s')V(s')$$

In the equations given above, $\mathit{succ}(s)$ 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 non-deterministic move is made to minimize the reliability, i.e., we are computing the worst case reliability. 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. If $V^\star$ is the value function obtained after convergence and s_0 is the initial state of the scenario graph, then $V^\star(s_0)$ is the *worst case reliability metric* corresponding to the given property. If non-deterministic moves are equated with the system's environment making a decision, then the algorithm just described is similar to *policy iteration* used for optimal control of *Markov Decision Processes (MDPs)* [2]. The proof of convergence is also similar to the one given in the context of MDPs.

Consider the scenario graph shown in Figure 3. If the symbolic probabilities `high`, `medium`, and `low` are set to 0.75, 0.5, and 0.25 respectively, then the reliability using the algorithm given above is $\frac{1}{2}$. Note that transition from the initial state to the state sendCheck(Bank-A,MC-1) is labeled with the set $\{$`high, medium`$\}$, i.e., a non-deterministic choice. We assume that this non-deterministic choice is broken to pick the lowest number in the set, i.e., `medium`.

5 Status

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

We have finished two major case studies: a model of a 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[1]. We have also modeled and analyzed the architecture of a bond trading floor of a major investment company in New York. The model of the bond trading floor is about 10,000 lines of `NuSMV` code and has about 100 state variables. Unfortunately, due to the propriety nature of the case study we cannot reveal additional details. We are in the process of "sanitizing" the model so that the case study can be published at a later date. Not surprisingly, we gained valuable experience during the case study. The most cumbersome part of the modeling process was the fault/intrusion injection phase because the nature of the faults/intrusions that were injected were heavily dependent on the security policies and technologies deployed at that node. We plan to automate the fault/intrusion injection process in the near future.

6 Related Work

Survivability is a fairly new discipline, and viewed by many as distinct from the traditional areas of security and fault-tolerance [6]. 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 [7]. The SNA methodology is informal and meant to provide general recommendations of "best practices" to an organization on how

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

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*.

Research on *operational security* [11] is closest to the work we presented. The attack state graph used in [11] is similar to scenario graphs we use. However, since we use symbolic model checking to generate scenario graphs, represented using *Binary Decision Diagrams* (*BDDs*), we can handle extremely large graphs. Moreover, in our method a scenario graph corresponds to a particular service in contrast to the global view taken in [11]. We are currently investigating how to incorporate their concepts and analysis techniques into our method.

Allowing individual nodes to fail is similar to *injecting faults* into the specification of the network architecture. Fault injection is a well-known technique in the fault tolerance community. We allow the designer to specify any kind of fault, and thus we can consider a wider class of faults. Moreover, we allow different classes of failure events, such as faults and intrusions, to be correlated. The idea of computing reliability is not new. There is a vast literature on verifying probabilistic systems and our algorithm for computing reliability draws on this previous work [5]. The novelty of our work is that it combines a number of techniques in a systematic way and thus provides a *holistic view* of the specification of the system. This view on systems is at the core of analyzing and achieving survivability of distributed systems.

7 Conclusion

Survivability has become increasingly important with society's increased dependence of critical infrastructures on computers. In this paper we presented a systematic methodology for analyzing the survivability of networked systems. We use scenario graphs to help a system architect visualize the effect of faults and intrusions on the entire network, and we use both symbolic and numeric probabilities to reason about its reliability. The novelty of our work is in the systematic combination of a variety of mathematical techniques: model checking, Bayesian analysis, and probabilistic systems. In combination, we provide a multi-faceted view of the network with respect to a desired service.

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, we are investigating how best to integrate operational security analysis tools such as COPS [8] into our method.

References

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

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

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

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

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

[6] 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.

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

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

[9] 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.

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

[11] 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.

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

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