Scenario Graphs and Attack Graphs

Oleg Mikhail Sheyner

CMU-CS-04-122

April 14, 2004

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Jeannette Wing, Chair
Edmund Clarke
Michael Reiter
Somesh Jha (University of Wisconsin)

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy.

This research was sponsored by the US Army Research Office (ARO) under contract no. DAAD190110485, the US Air Force Research Laboratory (AFRL) under grant no. F306020020523, and the Maryland Procurement Office (MPO) under contract no. MDA90499C5020. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the ARO, AFRL, MPO, the U.S. government or any other entity.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Jeannette Wing for her guidance and support in working with me through the Ph.D. program.

My committee members, Edmund Clarke, Somesh Jha, and Mike Reiter, have given me valuable feedback. Special thanks to Somesh for opening up the research on scenario graphs.

Roman Lototski and Alexey Roschyna implemented a large part of the toolkit. The idea for the traceback method grew out of our discussions about the hashcompact method implementation.

Finally, I would like to thank my parents, Elena and Mikhail Sheyner, for their patience and encouragement.

Prologue

We develop formal techniques that give users flexibility in examining design errors discovered by automated analysis. We build our results using the model checking approach to verification. The two inputs to a *model checker* are a finite system model and a formal correctness property specifying acceptable behaviors. The correctness property induces a bipartition on the set of behaviors of the model: *correct* behaviors, which satisfy the property, and *faulty* behaviors, which violate the property. Traditional model checkers give users a single counterexample, chosen from the set of faulty behaviors. Giving the user access to the entire set, however, lets him have more control over the design refinement process. The focus of our work is on ways of generating, presenting, and analyzing faulty behavior sets.

We present our results in two parts. In Part I we introduce concepts that let us define faulty behavior sets as *failure scenario graphs*. We then describe algorithms for generating scenario graphs. The algorithms use model checking techniques to produce faulty behavior sets that are sound and complete.

In Part II we apply our formal concepts to the security domain. Building on the foundation established in Part I, we define and analyze *attack graphs*, an application of scenario graphs to represent ways in which intruders attack computer networks. This application of formal analysis contributes to techniques and tools for strengthening network security.

Part I

Scenario Graphs

Chapter 1

Introduction

For the past twenty years, *model checking* has been used successfully in many engineering projects. Model checkers assist the engineer in identifying automatically individual design flaws in a system. A typical model checker takes as input a model of the system and a correctness specification. It checks the model against the specification for erroneous behavior. If erroneous behavior exists, the model checker produces an example that helps the user understand and address the problem. Once the problem is fixed, the user can repeat the process until the model satisfies the specification perfectly.

In some situations the process of repeatedly checking for and fixing individual flaws does not work well. Sometimes it is not feasible to eliminate every undesirable behavior. For instance, network security cannot in practice be made perfect due to a combination of factors: software complexity, desire to keep up with the latest features, expense of fixing known system vulnerabilities, etc. Despite these difficulties, network system administrators would invest the time and resources necessary to find and prevent the most destructive intrusion scenarios. However, the "find problem-fix problem-repeat" engineering paradigm inherent in traditional uses of model checkers does not make it easy to prioritize the problems and focus the limited available resources on the most pressing tasks.

We adapt model checking to situations where the ideal of a perfect design is not feasible. A recently proposed formalism appropriate for this task is *failure scenario graphs* [53]. Informally, a failure scenario graph is a succinct representation of all execution paths through a system that violate some correctness condition. Scenario graphs show everything that can go wrong in a system, leaving the engineer free to prioritize the problems as appropriate. In Part I of this thesis we give formal definitions for scenario graphs and develop algorithms that generate scenario graphs automatically from finite models. Part II contains a detailed discussion of a specific kind of scenario graph: *attack graphs*. We present examples where the system administrator uses attack graphs to assess the network's exposure to attack and evaluates effectiveness of various security fixes.

In the appendices we briefly address further avenues of research that can take advantage of the bird's-eye view of the system offered by scenario graphs. In the area of embedded systems, it is frequently impossible to predict adverse environmental behavior in advance, or deal perfectly with those situations that can be foreseen. More generally, any design problem involving reactive systems that has to deal with an unpredictable environment could benefit from scenario graph analysis. Whenever we can anticipate the environment to be hostile or unpredictable, designing the system to cope with the environment becomes a priority.

In Appendix A we treat such situations as games between the system and the environment, where the environment sometimes makes hostile moves and the system must find an effective counter-move. Adopting this stance, we view a scenario graph as a part of the decision graph of the game, with moves by the environment and counter-moves by the system. Specifically, the scenario graph is equivalent to the decision subgraph that includes all scenarios where the game is won by the environment.

When we are modeling a system that operates in an uncertain environment, certain transitions in the model represent the system's reaction to changes in the environment. We can think of such transitions as being outside of the system's control—they occur when triggered by the environment. When no empirical information is available about the relative likelihood of such environment-driven transitions, we can model them only as nondeterministic "choices" made by the environment. However, sometimes we have empirical data that make it possible to assign probabilities to environment-driven transitions. In Appendix B we show how to incorporate probabilities into scenario graphs and demonstrate a correspondence between the resulting construction and Markov Decision Processes [3] (MDPs). The correspondence lets us take advantage of the existing body of algorithms for analyzing MDPs.

Traditional automata-theoretic model checkers accept correctness properties written in temporal logic and convert them into automata. Existing converstion algorithms work with automata accepting infinite executions only. Since our techniques are designed to work with both finite and infinite paths, in Appendix C we sketch a modification to the classic algorithm for converting LTL formulas [39] to Büchi automata. The modified algorithm outputs Büchi automata that accept both finite and infinite executions.

The rest of Part I is organized as follows. In Chapter 2 we define scenario graphs and the associated terminology used throughout the thesis. Chapter 3 presents algorithms for generating scenario graphs from finite models. In Chapter 4 we measure performance of our scenario graph generator and evaluate some algorithmic adjustments that trade off running time and completeness guarantees for memory requirements.

Chapter 2

Scenario Graphs: Definitions

In this chapter we present definitions and terminology. Our first task is to define data structures to represent all failures in a system. We call such structures *failure scenario graphs*, or *scenario graphs* for brevity.

2.1 Scenario Graphs

Broadly, system requirements (also called *correctness properties*) can be classified into two categories, *safety* and *liveness* properties. Safety properties state that nothing bad ever happens in the system. For example: an intruder never gets user or administrative privileges on the network; a single component failure does not bring the entire system down; the controller does not allow the boiler temperature to rise above a certain level. Safety properties can be expressed as an invariant stating that the system never enters an unsafe state. Informally, a scenario graph representing violations of a safety property incorporates all the executions of a system that lead to an unsafe state. We can view a scenario graph as a directed graph where edges represent state transitions and vertices represent system states. Certain states are labeled unsafe. Each path from an initial state to an unsafe state is an execution of the system violating the safety property.

Safety properties alone are not enough to specify correct behavior in interesting systems—a system with an empty behavior set is safe, but not very useful. We use liveness requirements to specify the active tasks the system is designed to do. For example, a banking system might have a liveness requirement stating that if a check is deposited and sufficient funds are available, the check eventually clears. A violation of a liveness requirement occurs when the system gets stuck, either by stopping completely or going around behavioral cycles that do not accomplish the tasks specified in the liveness requirement. To represent liveness violations, both the modeling language and the scenario graph formalism must support infinite executions.

To account for both types of correctness properties, we define scenario graphs with respect to models that admit both finite and infinite executions. A simple formalism capable of describing such models is the *Büchi automaton*.

2.1.1 Büchi Automata

Traditional Büchi automata are nondeterministic finite automata equipped with an acceptance condition that works for infinite words (ω-words): an ω-word is accepted if and only if the automaton can read the word from left to right while visiting a sequence of states in which an *acceptance state*

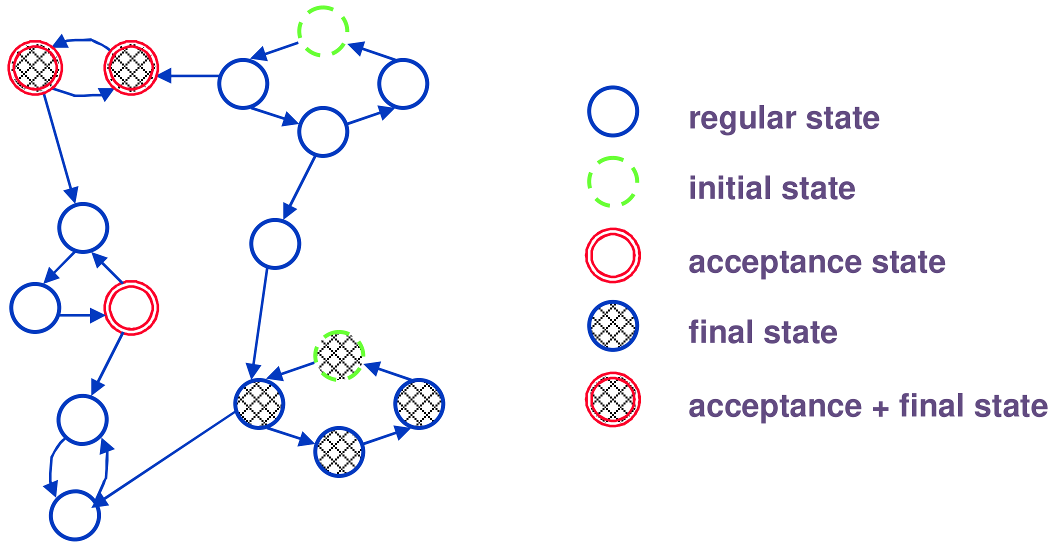


Figure 2.1: Büchi Automaton

occurs infinitely often. This is called the *Büchi acceptance* condition. Since we wish to include finite executions, we extend the set of executions accepted by Büchi automata to finite executions whose last state is a *final state* in the automaton.

Hereafter, we will work exclusively with Büchi automata that accept words over the alphabet $A = 2^{AP}$, where AP is a set of atomic propositions. A word in the alphabet—that is, a member of $A^\omega \cup A^*$, is called a *propositional sequence*.

Definition 2.1 *Given a set of atomic propositions AP, a* Büchi automaton *over the alphabet $A = 2^{AP}$ is a 6-tuple $\mathcal{B} = (S, \tau, S_0, S_a, S_f, D)$ with finite state set S, transition relation $\tau \subseteq S \times S$, initial state set $S_0 \subseteq S$, a set $S_a \subseteq S$ of* acceptance *states, a set $S_f \subseteq S$ of* final *states, and a labeling $D : S \to 2^A$ of states with sets of letters from A.*

Definition 2.2 *An* infinite execution *of a Büchi automaton $\mathcal{B} = (S, \tau, S_0, S_a, S_f, D)$ on an infinite propositional sequence $\xi = l_0 l_1 \ldots l_n \ldots$ is a sequence of states $\alpha = s_0 s_1 \ldots s_n \ldots$ such that $s_0 \in S_0$ and for all $i \geq 0$, $(s_i, s_{i+1}) \in \tau$ and $l_i \in D(s_i)$. We say that the execution α* accepts *the propositional sequence ξ.*

An infinite execution α is successful *if some acceptance state $s_a \in S_a$ occurs infinitely often in α. $\mathcal{B}$* accepts *propositional sequence ξ if there exists a successful execution of $\mathcal{B}$ that accepts ξ.*

Definition 2.3 *A* finite execution *of a Büchi automaton $\mathcal{B} = (S, \tau, S_0, S_a, S_f, D)$ on a finite propositional sequence $\xi = l_0 l_1 \ldots l_n$ is a sequence of states $\alpha = s_0 s_1 \ldots s_n$ such that $s_0 \in S_0$ and for all $0 \leq i < n$, $(s_i, s_{i+1}) \in \tau$ and $l_i \in D(s_i)$. We say that the execution α* accepts *the propositional sequence ξ.*

A finite execution α is successful *if $s_n \in S_f$. $\mathcal{B}$* accepts *propositional sequence ξ if there exists a successful execution of $\mathcal{B}$ that accepts ξ.*

It is often convenient to name a transition $(s, s') \in \tau$ with a label: $t = (s, s')$. Using this convention, we will sometimes include transition labels explicitly when writing out an execution, e.g. $\alpha = s_0 t_0 s_1 \ldots t_{i-1} s_i \ldots$, where for all $i \geq 0$ $t_i = (s_i, s_{i+1})$.

Definition 2.4 *The set*

$$\mathcal{L}(\mathcal{B}) = \{\alpha \in A^\omega \cup A^* \mid \mathcal{B} \text{ accepts } \alpha\}$$

is the language *recognized by Büchi automaton $\mathcal{B}$.*

Figure 2.1 is an example of a Büchi automaton. The set of acceptance states S_a is marked with double circles. The language recognized by the automaton includes any infinite sequence that goes through any of the three acceptance states infinitely often. An acceptance state may or may not also be an final state. In Figure 2.1 the final states are shaded. Any finite sequence of states that begins in an initial state and ends in one of the final (shaded) states is accepted by the automaton.

2.1.2 Generalized Büchi Automata

Sometimes it is convenient to work with a Büchi automaton with several accepting sets, even though this addition does not extend the expressive power of the Büchi formalism. In particular, algorithms presented in Chapter 3 convert Linear Temporal Logic formulas into generalized Büchi automata. Instead of a single acceptance set S_a, a *generalized Büchi automaton* has a set of acceptance sets $\mathcal{F} \subseteq 2^S$. The automaton accepts an execution α if and only if α visits each acceptance set $F_i \in \mathcal{F}$ infinitely often. There is a simple translation of a generalized Büchi automaton to an equivalent Büchi automaton [19], which we omit here.

2.1.3 Büchi Models

We say that a Büchi automaton M is a *Büchi model* of system J if the language $L(M)$ is a subset of the possible behaviors of J. Hereafter, we shall consider Büchi models over the alphabet $A = 2^{AP}$, where the set of atomic propositions AP describes the states of the system that we would like to model. For each state s in a Büchi model, the label $D(s)$ is a singleton set containing one letter of A. The letter identifies a particular state in the modeled system J.

A *scenario* is a synonym for a word in the alphabet—that is, a propositional sequence in $A^\omega \cup A^*$. A scenario describes a sequence of changes in the system state. The alphabet thus serves as the formal link between the system J and the Büchi automaton that models it. We distinguish between finite scenarios (members of A^*) and infinite scenarios (members of A^ω).

Since each state of a Büchi model M is labeled by exactly one letter of A, each execution $\alpha = s_0 s_1 \ldots$ of M always accepts exactly one scenario $\eta = D(s_0)D(s_1)\ldots$. When the context is clear, we sometimes use the term *scenario* to refer directly to an execution of M instead of the corresponding propositional sequence.

2.1.4 Scenario Graphs

Given a Büchi model M over the alphabet 2^{AP}, a *scenario graph* is a subset of $L(M)$, i.e. a collection of scenarios in the model. Most such collections are uninteresting; we care particularly about scenario graphs that describe erroneous system behavior with respect to some correctness condition. Scenarios can be finite or infinite, so we use Büchi automata to represent both the model and the scenario graph. We refer to Büchi automata representing scenario graphs as *scenario automata*.

Definition 2.5 *Given a Büchi model over a set of atomic propositions AP and alphabet $A = 2^{AP}$, a* correctness property *P is a subset of the set of scenarios $A^\omega \cup A^*$. A scenario $\alpha \in A^\omega \cup A^*$ is* correct *with respect to P iff $\alpha \in P$. A scenario α is* failing *with respect to P (*violates *P) iff $\alpha \notin P$.*

We say that a Büchi model M *satisfies* a correctness property P if it does not accept any failing scenarios (that is, if $L(M) \subset P$). If, however, M does accept some failing scenarios, we say that the set of such scenarios, $L(M)\backslash P$, is the *failure scenario graph* of M with respect to P.

Traditional model checking techniques focus on producing one member of a failure scenario graph as a counterexample. In Chapter 3 we present algorithms that generate a full scenario automaton for a Büchi model M and correctness property P. The algorithms guarantee that the generated scenario automaton accepts every failing scenario, yet does not contain any extraneous states and transitions.

Definition 2.6 *Given a Büchi model M and a correctness property P, a scenario automaton $M_P = (S, \tau, S_0, S_a, S_f, D)$ is* sound *if every scenario accepted by M_P is a failing scenario of M with respect to P: $L(M_P) \subseteq L(M)\backslash P$.*

Definition 2.7 *Given a Büchi model M and a correctness property P, a scenario automaton $M_P = (S, \tau, S_0, S_a, S_f, D)$ is* exhaustive *if it accepts every failing scenario of M with respect to P: $L(M_P) \supseteq L(M)\backslash P$.*

Definition 2.8 *Given a Büchi model M and a correctness property P, a scenario automaton $M_P = (S, \tau, S_0, S_a, S_f, D)$ is* succinct *if the following two conditions hold:*

(1) For every state $s \in S$ there exists a failing scenario $\alpha \in L(M)\backslash P$ containing s

(2) For every transition $t \in \tau$ there exists a failing scenario $\alpha \in L(M)\backslash P$ containing t

2.1.5 Examples

The structure of a scenario automaton depends on the structure of the model and the type of the correctness property. Figure 2.2 shows three different examples. In the first example (Figure 2.2(a)) the scenario automaton represents violations of a safety property ("b is never true") in a model without cycles. The scenario automaton is a directed acyclic graph, with the final states shaded. In the second example (Figure 2.2(b)) the underlying model contains cycles, and the resulting scenario automaton is a directed graph with cycles.

Finally, a model with cycles and a liveness property ("Eventually b is true") produce a general Büchi scenario automaton (Figure 2.2(c)). The shaded nodes in (c) represent cycles in which b never becomes true. Automaton executions that reach those cycles along a path where b is always false violate the liveness property. The set of such executions is the set of counterexamples to the liveness property.

2.1.6 Representative Scenarios

Scenario automata representing all failing scenarios can be quite large. Sometimes it is possible to represent the failing scenarios in a more compact way. We observe that a Büchi model may, after a finite number of executions steps, find itself in a state where every possible continuation leads to

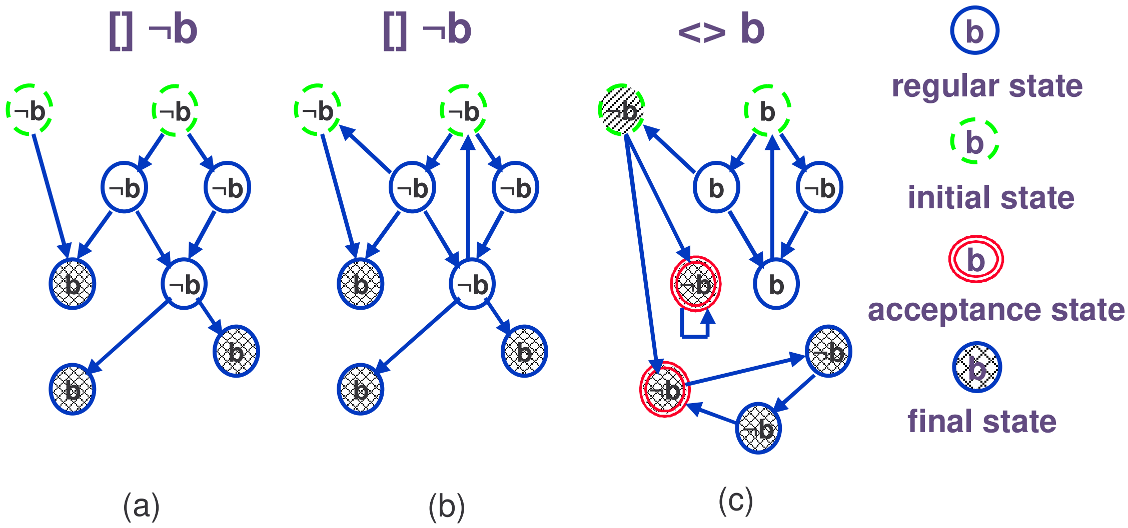


Figure 2.2: Scenario Automata Examples

a violation of the correctness property. The simplest example is a finite execution α ending in a state where some proposition q is true. The execution α is a valid scenario violating the invariant correctness property $\mathbf{G}\neg q$, and so is every finite or infinite execution with the prefix α. In practice, including all of these executions in a scenario graph is counter-productive. Instead, we can include only the execution α and designate it as a representative for all other executions that have the prefix α.

Figure 2.3(a) illustrates a part of a Büchi automaton where a set of finite executions can be replaced with a representative prefix. Consider a finite prefix ending in the state indicated by the arrow. However we choose to extend the prefix into a finite scenario, the scenario will be accepted by the automaton. Thus, it is possible to replace that part of the automaton with a single state and designate scenarios ending in this state as representative (Figure 2.3(b)).

Less obviously, this observation also applies to situations where all infinite executions with a certain finite prefix violate a liveness property. Consider the LTL correctness property

$$\mathbf{G}\ \mathbf{F}\ \mathit{DeviceEnabled}$$

stating that some device must be enabled infinitely often in every execution. Only infinite executions could properly be considered as scenarios violating this property. However, if a certain finite execution α reaches a state s from which it is not possible to reach another state where *DeviceEnabled* holds, then we know that any infinite scenario with the prefix α is a violation of the correctness property. α could therefore serve as a representative for an entire set of infinite scenarios in the scenario graph, reducing graph clutter and size without losing interesting information.

Figure 2.4(a) illustrates a part of a Büchi automaton where a set of infinite executions can be replaced with a (finite) representative prefix. Consider a finite prefix ending in the state indicated by the arrow. However we choose to extend the prefix into an infinite scenario, the scenario will be accepted by the automaton. Thus, it is possible to replace that part of the automaton with a single state and designate scenarios ending in this state as representative (Figure 2.4(b)).

The key to constructing representatives for finite scenarios is picking out short executions that prefix a class of (longer) failing executions.

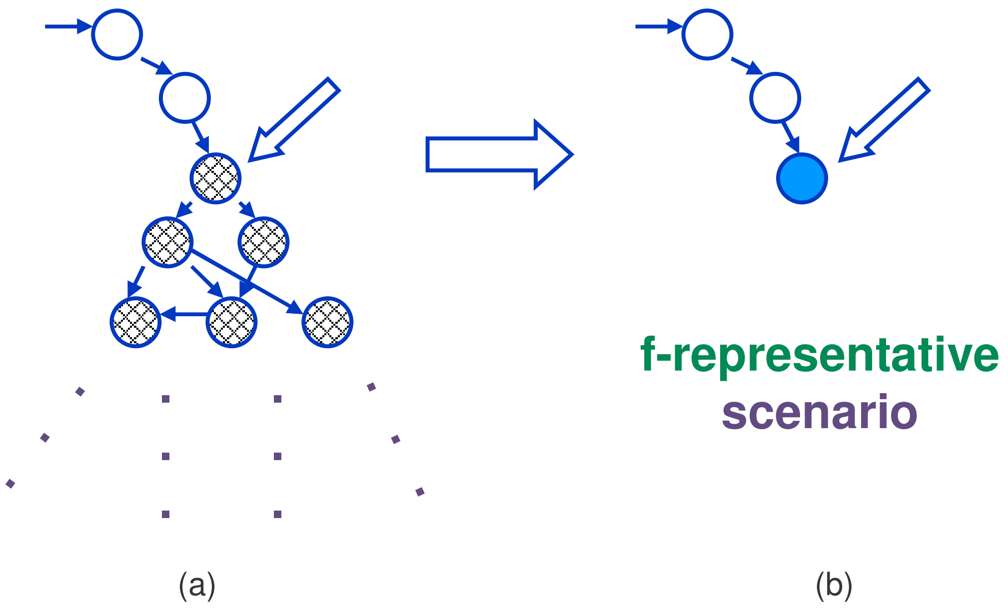


Figure 2.3: Representative Scenario for Finite Executions

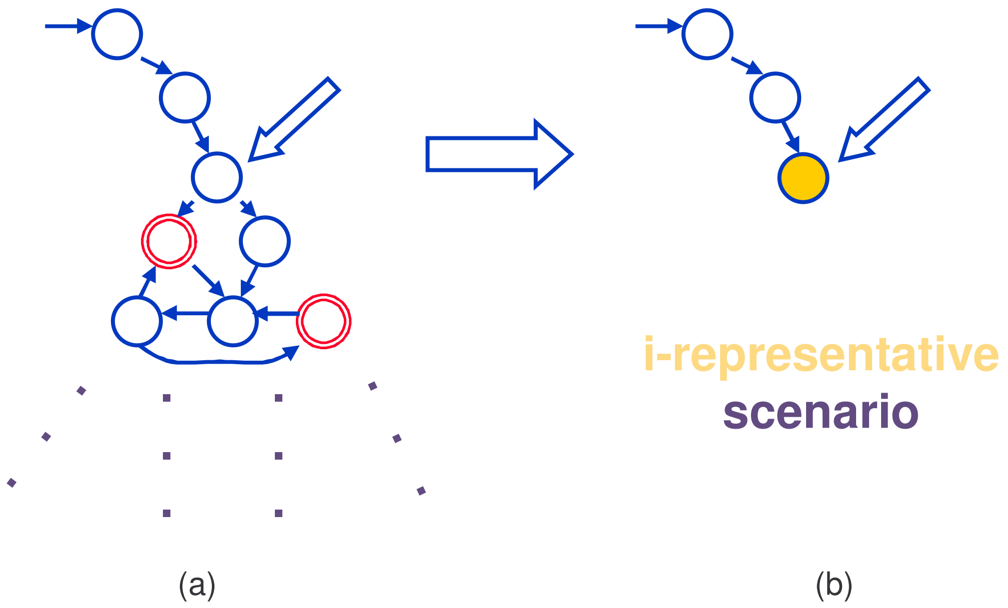


Figure 2.4: Representative Scenario for Infinite Executions

Definition 2.9 *Given a Büchi automaton $M = (S, \tau, S_0, S_a, S_f, D)$ and a correctness property P, a finite execution γ in M is* f-representative *with respect to P if every finite execution α that has prefix γ is a failing execution of M with respect to P.*

Definition 2.10 *Given a Büchi automaton $M = (S, \tau, S_0, S_a, S_f, D)$, a correctness property P, and a finite execution α in M, we say that a finite execution γ is an* f-prefix *of α with respect to P if (1) γ if a prefix of α and (2) γ is f-representative with respect to P. An f_n*-prefix *of α is an f-prefix of size n.*

From here on, whenever the correctness property P is clear from context, we shall refer to f-representative scenarios and f-prefixes without mentioning P explicitly.

Definition 2.11 *The shortest f-prefix γ of α is the* characteristic f-prefix *of α.*

Definition 2.12 *Two finite executions α_1 and α_2 are f_n*-equivalent *(designated $=_{f_n}$, $n \in$ **N**) if they share an identical f_n-prefix. The notation $[\alpha]_{=_{f_n}}$ designates the* f_n-equivalence class *of execution α with respect to the relation $=_{f_n}$. The f_n-prefix of α is also the* f_n-prefix *of the f_n-equivalence class $[\alpha]_{=_{f_n}}$.*

A similar set of special finite prefixes can be identified for infinite executions.

Definition 2.13 *Given a Büchi automaton $M = (S, \tau, S_0, S_a, S_f, D)$ and a correctness property P, a finite execution γ in M is* i-representative *with respect to P if every infinite execution β that has prefix γ is a failing execution of M with respect to P.*

Definition 2.14 *Given a Büchi automaton $M = (S, \tau, S_0, S_a, S_f, D)$, a correctness property P, and an infinite execution β in M, we say that a finite execution γ is an* i-prefix *of β with respect to P if (1) γ if a prefix of β and (2) γ is i-representative with respect to P. An i_n*-prefix *of β is an i-prefix of size n.*

From here on, whenever the correctness property P is clear from context, we shall refer to i-representative scenarios and i-prefixes without mentioning P explicitly.

Definition 2.15 *The shortest i-prefix γ of β is the* characteristic i-prefix *of β.*

Definition 2.16 *Two infinite executions β_1 and β_2 are* i_n-equivalent *(designated $=_{i_n}$) if they share an identical i_n-prefix γ. The notation $[\beta]_{=_{i_n}}$ designates the* i_n-equivalence class *of execution β with respect to the relation $=_{i_n}$. The i_n-prefix of β is also the* i_n-prefix *of the i_n-equivalence class $[\beta]_{=_{i_n}}$.*

2.1.7 Representative Scenario Automata

When appropriate, we replace full scenario automata with automata that accept an f-prefix or an i-prefix execution of every failing scenario. Each such execution α then serves as a representative for the entire equivalence class of failing scenarios that have the prefix α. Omitting these other scenarios from the scenario graph reduces clutter and size.

Definition 2.17 *Given a Büchi model M and a correctness property P, a* representative scenario automaton *(or* rs-automaton *for short) is a triple $M_P^r = (M_r, S_{rf}, S_{ri})$, where $M_r = (S, \tau, S_0, S_a, S_f, D)$ is a Büchi automaton, S_{rf} and S_{ri} are subsets of S_f, and executions in the language $L(M_r)$ fall into three classes:*

1. *Any finite execution $\alpha \in L(M_r)$ whose final state is in S_{rf} is an f-prefix of some failing execution in M. We call such executions* f-scenarios.
2. *Any finite execution $\gamma \in L(M_r)$ whose final state is in S_{ri} is an i-prefix of some infinite failing execution in M. We such executions* i-scenarios.
3. *Any execution in $L(M_r)$ that does not belong to the other two classes is a failing execution in M. We call such executions* r-scenarios.

In an rs-automaton $M_P^r = (M_r, S_{rf}, S_{ri})$ each f-scenario α serves as a representative for the entire equivalence class $[\alpha]_{=_{|\alpha|}}$ of finite failing scenarios of M. Similarly, each i-scenario γ serves as a representative for the entire equivalence class $[\gamma]_{=_{|\gamma|}}$ of infinite failing scenarios of M. Thus, it is understood that every finite execution of M with an f-scenario prefix in M_P^r and every infinite execution of M with an i-scenario prefix in M_P^r is a failing scenario of M with respect to property P.

Ideally, an rs-automaton would accept only the characteristic (that is, shortest) prefix of every full failing scenario. However, it may not be efficient to look for the shortest prefix in every case. Furthermore, it may not be feasible or desirable to replace every failing scenario of M with its representative prefix. So the definition of rs-automata leaves us room to include f- and i-prefixes of any size, as well as full failing scenarios.

Clearly, rs-automata have weaker soundness and exhaustiveness guarantees than regular scenario automata. Definitions 2.18 and 2.19 specify new, weaker soundness and exhaustiveness conditions that hold for representative scenario automata.

Definition 2.18 *Given a Büchi model M and a correctness property P, an rs-automaton $M_P^r = (M_r, S_{rf}, S_{ri})$ is* sound *if for every scenario γ accepted by M_P^r one of the following holds:*

1. *γ is failing scenario of M with respect to P, or*
2. *γ is an f-prefix of some finite failing scenario α of M, or*
3. *γ is an i-prefix of some infinite failing scenario β of M*

Definition 2.19 *Given a Büchi model M and a correctness property P, an rs-automaton $M_P^r = (M_r, S_{rf}, S_{ri})$ is* exhaustive *if for every failing scenario γ of M with respect to P one of the following holds:*

1. *γ is an r-scenario in M_P^r, or*
2. *an f_n-prefix of γ is an f-scenario in M_P^r for some value of n, or*
3. *an i_n-prefix of γ is an i-scenario in M_P^r for some value of n*

The succinctness condition for rs-automata is identical to the succinctness condition of regular scenario automata.

Occasionally it is useful to put finite and infinite scenarios into separate rs-automata. For such cases we define exhaustiveness conditions that treat finite and infinite scenarios separately.

Definition 2.20 *Given a Büchi model M and a correctness property P, an rs-automaton $M_P^r = (M_r, S_{rf}, S_{ri})$ is* f-exhaustive *if for every finite failing scenario γ of M with respect to P one of the following holds:*

1. *γ is an r-scenario in M_P^r, or*
2. *an f_n-prefix of γ is an f-scenario in M_P^r for some value of n*

Definition 2.21 *Given a Büchi model M and a correctness property P, an rs-automaton $M_P^r = (M_r, S_{rf}, S_{ri})$ is* i-exhaustive *if for every infinite failing scenario γ of M with respect to P one of the following holds:*

1. *γ is an r-scenario in M_P^r, or*
2. *an i_n-prefix of γ is an i-scenario in M_P^r for some value of n*

2.1.8 Safety Scenario Graphs

In general, we require the full power of Büchi automata to represent arbitrary scenario graphs. *Safety properties* are a particular kind of correctness property that admit simpler representations for corresponding scenario graphs. A safety property P is associated with a characteristic logical predicate $Q_P : 2^{AP} \rightarrow$ *bool* specifying a set of states (from 2^{AP}) that are not allowed to appear in any scenario in P. Any infinite scenario that fails with respect to P must have a finite prefix ending in a state s where $Q_P(s)$ is false. In practice, the set of all such prefixes is usually a sufficient substitute for the full scenario graph. When this is the case, we can represent the graph with a simple rs-automaton.

Definition 2.22 *Given a Büchi model M and a safety property P, a* safety scenario graph *is an rs-automaton $M_P^r = (M_r, S_{rf}, S_{ri})$ where*

$M_r = (S, \tau, S_0, \emptyset, S_f, D)$

$S_{rf} = S_f$

$S_{if} = S_f$

Safety scenario graphs accept only f_n- and i_n-prefixes of failing scenarios of M. They do not accept any infinite scenarios. Therefore, in a safety scenario graph the set of acceptance states S_a is always empty and sets S_{rf} and S_{ri} are always equal to S_f. Whenever context permits, we will omit those components and specify safety scenario graphs as a 5-tuple (S, τ, S_0, S_f, D).

Chapter 3

Algorithms

We turn our attention now to algorithms for generating scenario automata automatically. Starting with a description of a model M and correctness property P, the task is to construct a scenario automaton representing all executions of M that violate P. It is essential that the automata produced by the algorithms be sound, exhaustive and succinct.

3.1 Background: Temporal Logic

Temporal logic is a formalism for describing sequences of transitions between states in a reactive system. In the temporal logics that we will consider, time is not mentioned explicitly; instead, a formula might specify that *eventually* some designated state is reached, or that an error state is *never* entered. Properties like *eventually* or *never* are specified using special *temporal operators*. These operators can also be combined with boolean connectives or nested arbitrarily. Temporal logics differ in the operators that they provide and the semantics of those operators. We will cover the temporal logic CTL* and its sub-logic LTL.

3.1.1 The Computation Tree Logic CTL*

Conceptually, CTL* formulas describe properties of *computation trees*. The tree is formed by designating a state in a Büchi model as the *initial state* and then unwinding the model into an infinite tree with the designated state as the root. The computation tree shows all of the possible executions starting from the initial state.

In CTL* formulas are composed of *path quantifiers* and *temporal operators*. The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers: **A** (for "all computation paths") and **E** (for "some computation path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have some property. The temporal operators describe properties of a path through the tree. There are five basic operators:

- The **X** ("next time") operator requires that a property holds in the second state of the path.
- The **F** ("eventually") operator is used to assert that a property will hold at some state on the path.

- The **G** ("always" or "globally") operator specifies that a property holds at every state on the path.
- The **U** ("until") operator is used to combine two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path the first property holds.
- The **R** ("release") operator is the logical dual of the **U** operator. It requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

There are two types of formulas in CTL*: *state* formulas (which are true in a specific state) and *path* formulas (which are true along a specific path). Let AP be a set of atomic propositions. The syntax of state formulas is given inductively by the following rules:

- If $p \in AP$, then p is a state formula.
- If f and g are state formulas, then $\neg f$, $f \vee g$, and $f \wedge g$ are state formulas.
- If f is a path formula, then **E** f and **A** f are state formulas.

Two additional rules specify the syntax of path formulas:

- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, **X** f, **F** f, **G** f, f **U** g, and f **R** g are path formulas.

CTL* is the set of state formulas generated by the above rules.

We define the semantics of CTL* with respect to a Büchi model $M = (S, \tau, S_0, S_a, S_f, D)$. A *path* in M is a sequence of states and transitions $\alpha = s_0 t_0 \ldots t_{n-1} s_n \ldots$ such that for every $i \geq 0$, $(s_i, t_i, s_{i+1}) \in \tau$. A path can be finite or infinite. We allow paths of length zero, denoted $\alpha = [\,]$. Sometimes we will omit the transitions and specify paths as $\alpha = s_0 \ldots s_n \ldots$. We use α^i to denote the *suffix* of α starting at s_i. The *length* of a finite path α, denoted $|\alpha|$, is the number of states in the sequence α. Conventionally, we say that the length of an infinite path is ∞.

If f is a CTL* state formula, the notation $M, s \models f$ means that f holds at state s in the Büchi model M. Similarly, if f is a path formula, $M, \alpha \models f$ means that f holds along the path α in the Büchi model M. When the Büchi model M is clear from context, we will usually omit it. The relation $\models$ is defined inductively below. In the following definition, p is an atomic proposition, f, f_1, and f_2 are state formulas, μ and ψ are path formulas, and α is any path.

1.	$M, s \models p$	$\Leftrightarrow$	$p \in \bigcup_{l \in D(s)} l$.
2.	$M, s \models \neg f$	$\Leftrightarrow$	$M, s \not\models f$.
3.	$M, s \models f_1 \vee f_2$	$\Leftrightarrow$	$M, s \models f_1$ or $M, s \models f_2$.
4.	$M, s \models f_1 \wedge f_2$	$\Leftrightarrow$	$M, s \models f_1$ and $M, s \models f_2$.
5.	$M, s \models$ **E** g	$\Leftrightarrow$	there is a path α from s such that $M, \alpha \models g$.
6.	$M, s \models$ **A** g	$\Leftrightarrow$	for every path α starting from s, $M, \alpha \models g$.
7.	$M, \alpha \models f$	$\Leftrightarrow$	s is the first state of α and $M, s \models f$.
8.	$M, \alpha \models \neg\mu$	$\Leftrightarrow$	$M, \alpha \not\models \mu$.

9.	$M, \alpha \models \mu \vee \psi$	$\Leftrightarrow$	$M, \alpha \models \mu$ or $M, \alpha \models \psi$.
10.	$M, \alpha \models \mu \wedge \psi$	$\Leftrightarrow$	$M, \alpha \models \mu$ and $M, \alpha \models \psi$.
11.	$M, \alpha \models \mathbf{X}\, \mu$	$\Leftrightarrow$	$M, \alpha^1 \models \mu$.
12.	$M, \alpha \models \mathbf{F}\, \mu$	$\Leftrightarrow$	there exists a $0 \leq k < \lvert\alpha\rvert$ such that $M, \alpha^k \models \mu$.
13.	$M, \alpha \models \mathbf{G}\, \mu$	$\Leftrightarrow$	for all $0 \leq i < \lvert\alpha\rvert$, $M, \alpha^i \models \mu$.
14.	$M, \alpha \models \mu\, \mathbf{U}\, \psi$	$\Leftrightarrow$	$\exists\, 0 \leq k < \lvert\alpha\rvert$ s.t. $M, \alpha^k \models \psi$ and for all $0 \leq j < k$, $M, \alpha^j \models \mu$.
15.	$M, \alpha \models \mu\, \mathbf{R}\, \psi$	$\Leftrightarrow$	$\forall\, 0 \leq j < \lvert\alpha\rvert$, if for every $i < j$ $M, \alpha^i \not\models \mu$ then $M, \alpha^j \models \psi$.

We use `T` as an abbreviation for $f \vee \neg f$ and `F` as an abbreviation for $\neg$`T`.

3.1.2 Linear Temporal Logic

We define Linear Temporal Logic (LTL) formulas over individual propositional sequences (instead of computation trees). The set of well-formed LTL formulas is constructed from the set of atomic propositions AP, the boolean operators $\neg$, $\vee$, and $\wedge$, and the temporal operators **X**, **F**, **G**, **U**, and **R**. Precisely,

- every member of AP is a formula,
- if f and g are formulas, then $\neg f$, $f \vee g$, $f \wedge g$, **X** f, **F** f, **G** f, f **U** g, and f **R** g are formulas.

Given a propositional sequence $\xi = l_0 l_1 \ldots \in A^\omega \cup A^*$ and an LTL formula η, the notation $\xi \models \eta$ means that η holds along the sequence ξ. Formally,

1.	$\xi \models A$	$\Leftrightarrow$	$A \in AP$ and $A \in l_0$.
2.	$\xi \models \neg\mu$	$\Leftrightarrow$	$\xi \not\models \mu$.
3.	$\xi \models \mu \vee \psi$	$\Leftrightarrow$	$\xi \models \mu$ or $\xi \models \psi$.
4.	$\xi \models \mu \wedge \psi$	$\Leftrightarrow$	$\xi \models \mu$ and $\xi \models \psi$.
5.	$\xi \models \mathbf{X}\, \mu$	$\Leftrightarrow$	$\xi^1 \models \mu$.
6.	$\xi \models \mathbf{F}\, \mu$	$\Leftrightarrow$	there exists a $0 \leq k < \lvert\alpha\rvert$ such that $\xi^k \models \mu$.
7.	$\xi \models \mathbf{G}\, \mu$	$\Leftrightarrow$	for all $0 \leq i < \lvert\alpha\rvert$, $\xi^i \models \mu$.
8.	$\xi \models \mu\, \mathbf{U}\, \psi$	$\Leftrightarrow$	$\exists\, 0 \leq k < \lvert\alpha\rvert$ s.t. $\xi^k \models \psi$ and for all $0 \leq j < k$, $\xi^j \models \mu$.
9.	$\xi \models \mu\, \mathbf{R}\, \psi$	$\Leftrightarrow$	$\forall\, 0 \leq j < \lvert\alpha\rvert$, if for every $i < j$ $\xi^i \not\models \mu$ then $\xi^j \models \psi$.

3.2 Algorithm I: Symbolic Model Checking

Model checking technology [11, 19, 59] has been successfully employed in designing and debugging systems. Model checking is a technique for checking whether a formal model M of a system satisfies a given property P. If the property is false in the model, model checkers typically output a counter-example, or a sequence of transitions ending with a violation of the property.

We briefly describe a symbolic algorithm for constructing safety scenario graphs (defined in Section 2.1.8). The algorithm is due to Jha and Wing [53]. The algorithm is a modification of an iterative fixpoint model checking procedure that returns a single counterexample [55]. The input $M = (S, \tau, S_0, S, S, D)$ is a Büchi model that accepts finite and infinite executions. The input P is

Input:

$M = (S, \tau, S_0, S, S, D)$ – Büchi model
S – set of states
$\tau \subseteq S \times S$ – transition relation
$S_0 \subseteq S$ – set of initial states
$D : S \rightarrow 2^{AP}$ – labeling of states with propositional formulas
$P = \mathbf{AG}(\neg unsafe)$ – a safety property

Output:

Safety scenario graph $G_P = (S_P, \tau_P, S_0^P, S_f^P, D)$

Algorithm: `SymScenarioGraph`(S, τ, S_0, D, P)

– *Construct binary decision diagrams for the components of M*
(1) $(S^{bdd}, \tau^{bdd}, S_0^{bdd}) \leftarrow$ `ConstructBDD`(S, τ, S_0)
– *Compute the set of states reachable from the set of initial states*
(2) $S_{reach} \leftarrow$ `ComputeReachable`$(S^{bdd}, \tau^{bdd}, S_0^{bdd})$
– *Use iterative fixpoint symbolic procedure to find the set of states S_P where*
– *the safety property P fails*
(3) $S_P \leftarrow$ `SymModelCheck`$(S_{reach}, \tau^{bdd}, S_0^{bdd}, D, P)$
– *Restrict the transition relation τ to states in the set S_P*
(4) $\tau_P \leftarrow \tau \cap (S_P \times S_P)$
(5) $S_0^P \leftarrow S_0^{bdd} \cap S_P$
(6) $S_f^P \leftarrow \{s \in S_P \mid D(s) \rightarrow unsafe\}$
(7) $Return(S_P, \tau_P, S_0^P, S_f^P, D)$

Figure 3.1: Symbolic Algorithm for Generating Safety Scenario Graphs

a safety property written in CTL, a sub-logic of CTL*. CTL safety properties have the form $\mathbf{AG}f$ (i.e., $P = \mathbf{AG}f$, where f is a formula in propositional logic).

The algorithm is shown in Figure 3.1. The first step is to compute symbolically the set of states reachable from the set of initial states. The second step is to determine the set of states S_P that have a path from an initial state to an unsafe state. The set of states S_P is computed using an iterative algorithm derived from a fix-point characterization of the **AG** operator [55]. Let τ be the transition relation of the model, i.e., $(s, s') \in \tau$ if and only if there is a transition from state s to s'. By restricting the domain and range of τ to S_P we obtain a transition relation τ_P that encapsulates the edges of the scenario graph. Therefore, the safety scenario graph is $(S_P, \tau_P, S_0^P, S_f^P, D)$, where S_P and τ_P represent the set of nodes and set of edges of the graph, respectively, $S_0^P = S_0 \cap S_P$ is the set of initial states, and $S_f^P = \{s | s \in S_P \wedge D(s) \rightarrow unsafe\}$ is the set of final states.

In symbolic model checkers, such as NuSMV [68], the transition relation and sets of states are represented using BDDs, a compact representation for boolean functions. There are efficient BDD algorithms for all operations used in the algorithm shown in Figure 3.1. These algorithms are described elsewhere [55, 19], and we omit them here.

Using the rs-automata definition of soundness and exhaustiveness, we show that the safety scenario graph G_P generated by the algorithm in Figure 3.1 is sound, exhaustive, and succinct.

Lemma 3.1 ***(sym-sound)*** *Let the safety scenario graph $G_P = (S_P, \tau_P, S_0^P, S_f^P, D)$ be the output of the algorithm in Figure 3.1 for input model $M = (S, R, S_0, S, S, D)$ and safety property $P =$*

AG$(\neg unsafe)$. *Then, any scenario* γ *accepted by* G_P *is failing in* M *with respect to* P*:* $L(G_P) \subseteq L(M) \backslash P$.

Proof:

Let γ be a scenario accepted by G_P. Then γ is finite and terminates in a state contained in S_f^P. By construction, *unsafe* holds in that state (step 6). Furthermore, G_P contains only states and transitions from M, so γ is accepted by M. γ fails with respect to P since it has a state where *unsafe* holds.

□

Lemma 3.2 ***(sym-exhaustive-finite)*** *A finite scenario* α *of the input Büchi model* $M = (S, R, S_0, S, S, D)$ *is failing with respect to property* $P =$ **AG**$(\neg unsafe)$ *if and only if there is an f-prefix* γ *of* α *accepted by the safety scenario graph* $G_P = (S_P, \tau_P, S_0^P, S_f^P, D)$ *output by the algorithm in Figure 3.1.*

Proof:

$(\Rightarrow)$ Let $\alpha = s_0 t_0 \ldots t_{j-1} s_j$ be a finite execution of M that fails with respect to property $P =$ **AG**$(\neg unsafe)$. α must contain at least one state where *unsafe* is true. Let s_n be the first such state, and let $\gamma = s_0 t_0 \ldots t_{n-1} s_n$. We show that G_P accepts γ and that γ is an f-prefix of α.

To prove that γ is accepted by G_P, it is sufficient to show that (1) $s_0 \in S_0^P$, (2) $s_n \in S_f^P$, and (3) for all $0 \leq k \leq n$, $s_k \in S$ and $t_k \in \tau_P$. Since *unsafe* holds at s_n, by definition every state s_k along γ violates **AG**$(\neg unsafe)$. Therefore, by construction in step 3, every s_k in γ is in S_P and every t_k is in τ_P, guaranteeing (3). (1) and (2) follow from steps 5 and 6 of the algorithm.

It remains to show that γ is an f-prefix of α with respect to property $P =$ **AG**$(\neg unsafe)$. Let α' be any execution of M that has the prefix γ. α' contains the state s_n, where *unsafe* holds. Therefore, α' is failing with respect to P. By Definitions 2.9 and 2.10, γ is an f-prefix of α.

$(\Leftarrow)$ Suppose that $\gamma = s_0 t_0 \ldots t_{n-1} s_n$ is a prefix of a finite execution $\alpha = s_0 t_0 \ldots t_{j-1} s_j$ of M, and that γ is accepted by G_P. Then, the terminal state s_n is in S_f^P, so *unsafe* is true at s_n. It follows that α violates the property **AG**$(\neg unsafe)$.

□

Lemma 3.3 ***(sym-exhaustive-infinite)*** *An infinite scenario* β *of the input Büchi model* $M = (S, R, S_0, S, S, D)$ *is failing with respect to property* $P =$ **AG**$(\neg unsafe)$ *if and only if there is an i-prefix* γ *of* β *accepted by the safety scenario graph* $G_P = (S_P, \tau_P, S_0^P, S_f^P, D)$ *output by the algorithm in Figure 3.1.*

Proof:

The proof is analogous to the proof of Lemma 3.2.

□

Lemma 3.4 ***(sym-succinct-state)*** *Let the safety scenario graph* $G_P = (S_P, \tau_P, S_0^P, S_f^P, D)$ *be the output of the algorithm in Figure 3.1 for input model* $M = (S, R, S_0, S, S, D)$ *and safety property* $P =$ **AG**$(\neg unsafe)$*. Then, for every state* $s \in S_P$ *there exists a scenario* α *in* G_P *that contains* s.

Proof:

By construction in steps 2 and 3 of the algorithm, all states generated for the safety scenario graph G_P are reachable from an initial state, and all of them violate $\mathbf{AG}(\neg unsafe)$. Therefore, for any state $s \in S_P$, there is a path α_1 from an initial state to s, and there is a path α_2 from s to an *unsafe* state.

Let $\alpha = \alpha_1\alpha_2$. α violates $\mathbf{AG}(\neg unsafe)$, so it is accepted by G_P. In addition, α contains s, as desired.

□

Lemma 3.5 ***(sym-succinct-transition)*** *Let the safety scenario graph $G_P = (S_P, \tau_P, S_0^P, S_f^P, D)$ be the output of the algorithm in Figure 3.1 for input model $M = (S, R, S_0, S, S, D)$ and safety property $P = \mathbf{AG}(\neg unsafe)$. Then, for every transition $(s_1, t, s_2) \in \tau_P$ there exists a scenario α in G_P that contains t.*

Proof:

By Lemma 3.4, there is a scenario $\alpha_1 = q_0t_0 \dots s_1 \dots t_{m-1}q_m$ in G_P that contains state s_1 and another scenario $\alpha_2 = r_0t_0 \dots s_2 \dots t_{n-1}r_n$ in G_P that contains state s_2. The scenario $\alpha = q_0t_0 \dots s_1ts_2 \dots t_{n-1}r_n$ includes both states s_1 and s_2 and the transition t and is accepted by G_P.

□

Theorem 3.1 *Algorithm in Figure 3.1 generates sound, exhaustive, and succinct safety scenario graphs.*

Proof:

Follows directly from Lemmas 3.2, 3.3, 3.1, 3.4, and 3.5 and Definitions 2.8, 2.18, and 2.19.

□

3.3 Explicit-State Model Checking Algorithms

The symbolic scenario graph algorithm can sometimes take advantage of efficient state space representation inherent in the symbolic approach. However, it is not efficient for the category of applications that need many variables to represent the complexities of the problem, yet have relatively small reachable state spaces. The attack graph application (discussed in Part II) falls into this category. In our experience the symbolic model checker could spend hours constructing the binary decision diagrams for the transition relation in step 1 of the symbolic algorithm, only to discover in step 2 that the reachable state space is very small. For such problems explicit-state model checking is more appropriate.

Given a model represented by a Büchi automaton M and a correctness property P specified in Linear Temporal Logic, our objective is to build a scenario graph automaton M_s that recognizes precisely the set of executions of M that violate the property P. For example, given the model shown in Figure 3.2(a) and correctness property "$P =$ eventually c is *true*", the scenario automaton M_s is shown in Figure 3.2(b).

In outline, our algorithm is similar to the classic LTL model checking algorithm used in model checkers such as SPIN. Correctness property P induces a language $\mathcal{L}(P)$ of executions that are

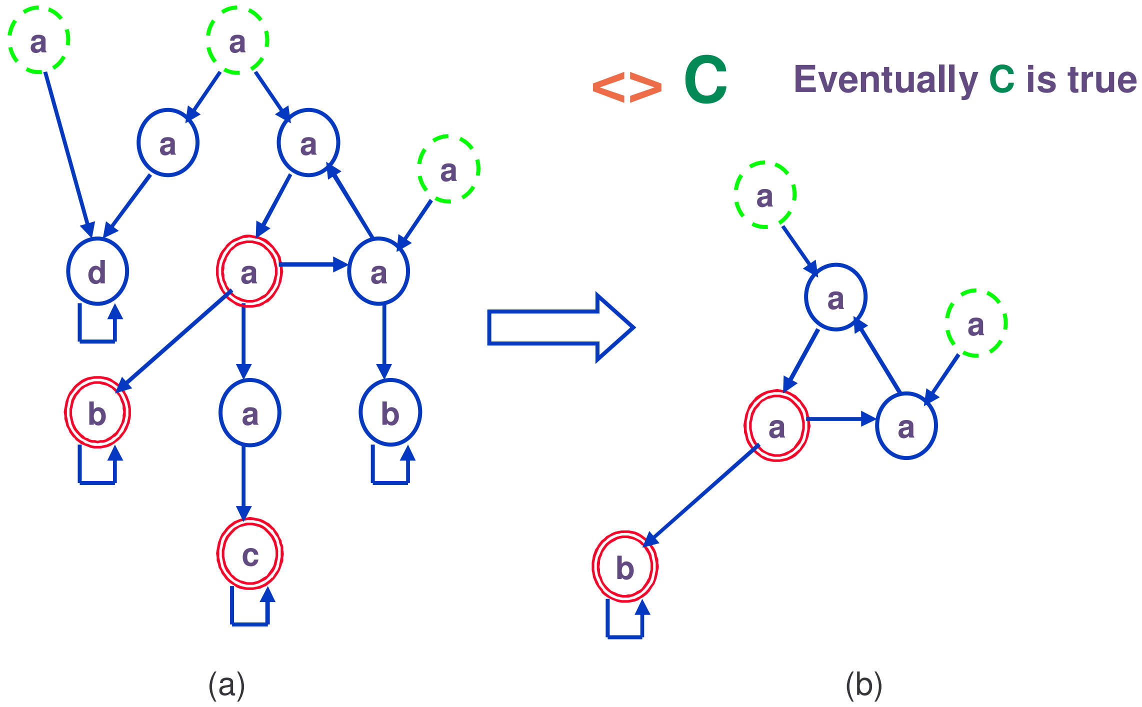


Figure 3.2: Example Scenario Automaton

permitted under the property. The executions of the model M that violate P thus constitute the language $\mathcal{L}(M)\backslash\mathcal{L}(P)$. The scenario graph automaton accepting this language is $M_s = M \cap \neg P$. In the following sections we describe each part of building M_s in detail and consider several variations.

3.3.1 Strongly Connected Component Graphs

The explicit state algorithms described later in this Chapter will refer to *strongly connected components* of a Büchi automaton. A strongly connected component of a Büchi model $M = (S, \tau, S_0, S_a, S_f, D)$ is a subset of states $C \subseteq S$ such that for any pair of states $s_1 \in C$, $s_2 \in C$ there is a path from s_1 to s_2 via the transition relation τ. Thus, we can partition the set of states S into strongly connected components and construct a strongly connected component graph out of them.

Definition 3.1 *Let $M = (S, \tau, S_0, S_a, S_f, D)$ be a Büchi model. The* strongly connected component graph (SCC graph) *of M is a triple $SCC_M = (\mathcal{C}, E_\tau, \mathcal{C}_0)$, where*

- *$\mathcal{C}$ is the set of strongly connected components of M*
- *$E_\tau \subseteq \mathcal{C} \times \mathcal{C}$ is a transition relation on $\mathcal{C}$ such that $(c_1, c_2) \in E_\tau$ iff $c_1 \neq c_2$ and there exists a transition $(s_1, s_2) \in \tau$ such that $s_1 \in c_1$ and $s_2 \in c_2$*
- *$\mathcal{C}_0$ is the set of* initial *components such that $c_0 \in \mathcal{C}_0$ iff $c_0 \cap S_0 \neq \emptyset$.*

An SCC graph cannot contain cycles, since trivial cycles are ruled out by the definition of E_τ, and any nontrivial cycle involving more than one component would be collapsed into a single component. Thus, an SCC graph is always a directed acyclic graph. We can construct SCC graphs easily

Input:
$SCC_M = (\mathcal{C}, E_\tau, \mathcal{C}_0)$, an SCC graph
$Q : \mathcal{C} \to bool$, a predicate on the set $\mathcal{C}$
Output:
$\mathcal{C}'$, a subset of $\mathcal{C}$

```
Algorithm RSET-UNIVERSAL(C, E_τ, C_0, Q)
    C' ← ∅;
    V' ← ∅;
    forall c_0 ∈ C_0 \ V' do
        (C', V') ← Visit_u(c_0, C, E_τ, Q, C', V');
    end;
    return C';
end RSET-UNIVERSAL

function Visit_u(c, C, E_τ, Q, C', V')
    V' ← V' ∪ {c};
    forall c' ∈ C \ V' s.t. (c, c') ∈ E_τ do
        (C', V') ← Visit_u(c', C, E_τ, Q, C', V');
    end;
    if Q(c) ∧ ∀c' ∈ C | (c, c') ∈ E_τ . c' ∈ C' then
        C' ← C' ∪ {c};
    end;
    return (C', V');
end Visit_u
```

(a) RSET-UNIVERSAL

Input:
$SCC_M = (\mathcal{C}, E_\tau, \mathcal{C}_0)$, an SCC graph
$Q : \mathcal{C} \to bool$, a predicate on the set $\mathcal{C}$
Output:
$\mathcal{C}'$, a subset of $\mathcal{C}$

```
Algorithm RSET-EXISTENTIAL(C, E_τ, C_0, Q)
    C' ← ∅;
    V' ← ∅;
    forall c_0 ∈ C_0 \ V' do
        (C', V') ← Visit_e(c_0, C, E_τ, Q, C', V');
    end;
    return C';
end RSET-EXISTENTIAL

function Visit_e(c, C, E_τ, Q, C', V')
    V' ← V' ∪ {c};
    forall c' ∈ C \ V' s.t. (c, c') ∈ E_τ do
        (C', V') ← Visit_e(c', C, E_τ, Q, C', V');
    end;
    if Q(c) ∨ ∃c' ∈ C | (c, c') ∈ E_τ . c' ∈ C' then
        C' ← C' ∪ {c};
    end;
    return (C', V');
end Visit_e
```

(b) RSET-EXISTENTIAL

Figure 3.3: *RSet* Predicate Algorithms

using Tarjan's classic procedure for finding strongly connected components [90]. Tarjan's SCC procedure is linear in the size of τ.

For each SCC graph $SCC_M = (\mathcal{C}, E_\tau, \mathcal{C}_0)$ we define a function *RSet*: $\mathcal{C} \to \mathcal{P}(\mathcal{C})$. For each component $c \in \mathcal{C}$, another component c' is in *RSet*(c) iff either $c' = c$ or c' is reachable from c via the transition relation E_τ. That is, *RSet*(c) is the set of components reachable from c via the transition relation E_τ.

In the algorithms described later, the following two sub-problems come up several times. Given a Büchi model M and its SCC graph $SCC_M = (\mathcal{C}, E_\tau, \mathcal{C}_0)$, for each $c \in \mathcal{C}$ determine whether a given boolean predicate $Q : \mathcal{C} \to bool$ is true

1. for every component in *RSet*(c), or
2. for some component in *RSet*(c).

To make it more convenient to identify the components described by sub-problems (1) and (2), we define two more predicates on components.

Definition 3.2 *Given an SCC graph $SCC_M = (\mathcal{C}, E_\tau, \mathcal{C}_0)$ and a predicate $Q : \mathcal{C} \to$ bool, define predicates $Q^\forall : \mathcal{C} \to$ bool and $Q^\exists : \mathcal{C} \to$ bool as follows:*

$Q^\forall(c)$ is true iff Q is true at every component in RSet(c)

$Q^\exists(c)$ is true iff Q is true at some component in RSet(c).

So sub-problem (1) asks to find all components in an SCC graph where $Q^\forall$ holds. Similarly, sub-problem (2) asks to find all components in an SCC graph where $Q^\exists$ holds. Figure 3.3 shows two algorithms that compute answers to subproblems (1) and (2).

The procedure `RSET-UNIVERSAL` in Figure 3.3(a) takes as input an SCC graph $SCC_M = (\mathcal{C}, E_\tau, \mathcal{C}_0)$ and a predicate Q and finds every component $c \in \mathcal{C}$ such $Q^\forall$ is true at c. `RSET-UNIVERSAL` traverses the SCC graph in depth-first order, invoking the recursive function $\texttt{Visit}_u$ on every previously unexplored component that contains an initial state. Each invocation of $\texttt{Visit}_u$ receives two arguments: the set V' of components that have been visited previously, and a subset $\mathcal{C}' \subseteq V'$ of previously visited components. For each component c' in this subset $\mathcal{C}'$, Q is guaranteed to be true everywhere in *RSet*(c'). That is, $Q^\forall$ is true for every member of $\mathcal{C}'$.

The task of $\texttt{Visit}_u$ is to determine whether $Q^\forall$ is true for its argument component c. $\texttt{Visit}_u$ marks c as visited and then invokes itself recursively on each successor of c. Each recursive call on a successor of c determines whether or not $Q^\forall$ is true at that successor; each successor where $Q^\forall$ is true will be added to $\mathcal{C}'$. Once all successors have been visited, $\texttt{Visit}_u$ checks that Q is true at c and that every successor of c belongs to $\mathcal{C}'$. These conditions together guarantee that Q is true at every member of *RSet*(c), so $\texttt{Visit}_u$ adds c to $\mathcal{C}'$. When every component in the SCC graph SCC_M has been visited by $\texttt{Visit}_u$, the set $\mathcal{C}'$ contains precisely those components that satisfy the condition in subproblem (1).

Lemma 3.6 ***(visit-u-invariant)*** *Suppose function $\texttt{Visit}_u$ in Figure 3.3(a) is invoked with the arguments $(c, \mathcal{C}, E_\tau, Q, \mathcal{C}', V')$, where*

$\mathcal{C}$ is a set of components

$E_\tau : \mathcal{C} \times \mathcal{C}$ is a transition relation

$Q : \mathcal{C} \to$ bool is a predicate

$V' \subseteq \mathcal{C}$ is the set of previously visited components

$\mathcal{C}' \subseteq V'$ is the subset of previously visited components where $Q^\forall$ holds,

c is a member of $\mathcal{C} \setminus V'$

Then the first set of components output by function $\texttt{Visit}_u$ contains the entire set $\mathcal{C}'$ plus every component $c' \in$ RSet$(c) \setminus V'$ such that $Q^\forall(c')$ holds. The second set of components output by $\texttt{Visit}_u$ is equal to $V' \cup$ RSet(c).

Proof:

The proof proceeds by induction on the diameter of the subgraph rooted at c. When the diameter is 1, c has no successors and *RSet*$(c) = \{c\}$, so $Q^\forall(c)$ holds if and only if $Q(c)$ holds. In this case $\texttt{Visit}_u$ will not invoke itself recursively. $\texttt{Visit}_u$ adds c to V' and also adds c to $\mathcal{C}'$ whenever $Q(c)$ holds, as desired.

For the inductive case (diameter $= n$), $\texttt{Visit}_u$ invokes itself on each previously unexplored successor of c. The subgraph rooted at each successor of c has a diameter smaller than n, so we can apply the inductive hypothesis to each recursive invocation of $\texttt{Visit}_u$ and conclude that at the end of the **forall** loop every component reachable from c has been added to V' and every component reachable from c for which $Q^\forall$ holds has been added to $\mathcal{C}'$.

It remains to check that c itself is added to $\mathcal{C}'$ if and only if $Q^\forall(c)$ holds. The **if** statement following the **forall** loop adds c to $\mathcal{C}'$ when Q holds at c and every successor of c is in $\mathcal{C}'$. The fact that every successor of c is in $\mathcal{C}'$ means that $Q^\forall$ holds at every successor of c, which in turn means that Q holds at every state reachable from c. So $Q^\forall$ holds at c, as desired.

□

Theorem 3.2 ***(rset-universal-correct)*** *Let $\mathcal{C}'$ be the output of algorithm `RSET-UNIVERSAL` on the input SCC graph $SCC_M = (\mathcal{C}, E_\tau, \mathcal{C}_0)$ and predicate Q. Then $\mathcal{C}'$ is precisely the set of components that are reachable from some component in $\mathcal{C}_0$ and where $Q^\forall$ holds.*

Proof:

`RSET-UNIVERSAL` invokes $\mathtt{Visit}_u$ on every component of SCC_M that contains an initial state. The Theorem follows directly by applying invariant Lemma 3.6 to each of these invocations of $\mathtt{Visit}_u$.

□

The procedure `RSET-EXISTENTIAL` in Figure 3.3(b) is almost identical to `RSET-UNIVERSAL`. It takes as input an SCC graph $SCC_M = (\mathcal{C}, E_\tau, \mathcal{C}_0)$ and a predicate Q and finds every component $c \in \mathcal{C}$ such $Q^\exists$ is true at c. The recursive function $\mathtt{Visit}_e$ plays the same role for `RSET-EXISTENTIAL` as the function $\mathtt{Visit}_u$ plays for `RSET-UNIVERSAL`. The only difference between `RSET-EXISTENTIAL` and `RSET-UNIVERSAL` is in the condition of the **if** statement inside the function $\mathtt{Visit}_e$. To ensure that component c is added to $\mathcal{C}'$ if and only if $Q^\exists$ is true at c, the **if** statement checks that Q is true at c or at one of c's successors.

Lemma 3.7 and Theorem 3.3 guarantee correctness of the algorithm `RSET-EXISTENTIAL`. We omit the proofs, which are analogous to the proofs of Lemma 3.6 and Theorem 3.2.

Lemma 3.7 ***(visit-e-invariant)*** *Suppose function $\mathtt{Visit}_e$ in Figure 3.3(b) is invoked with the arguments $(c, \mathcal{C}, E_\tau, Q, \mathcal{C}', V')$, where*

$\mathcal{C}$ is a set of components

$E_\tau : \mathcal{C} \times \mathcal{C}$ is a transition relation

$Q : \mathcal{C} \to$ bool is a predicate

$V' \subseteq \mathcal{C}$ is the set of previously visited components

$\mathcal{C}' \subseteq V'$ is the subset of previously visited components where $Q^\exists$ holds,

c is a member of $\mathcal{C} \setminus V'$

Then the first set of components output by function $\mathtt{Visit}_u$ contains the entire set $\mathcal{C}'$ plus every component $c' \in RSet(c) \setminus V'$ such that $Q^\exists(c')$ holds. The second set of components output by $\mathtt{Visit}_u$ is equal to $V' \cup RSet(c)$.

Theorem 3.3 ***(rset-existential-correct)*** *Let $\mathcal{C}'$ be the output of algorithm `RSET-EXISTENTIAL` on the input SCC graph $SCC_M = (\mathcal{C}, E_\tau, \mathcal{C}_0)$ and predicate Q. Then $\mathcal{C}'$ is precisely the set of components that are reachable from some component in $\mathcal{C}_0$ and where $Q^\exists$ holds.*

3.3.2 Büchi Automata Intersection

The explicit-state model checking algorithm is based on constructing the intersection of the model automaton M and the automaton N_P representing the negation of the correctness property. Let

$$M = (S, \tau_m, S_m^0, S_a, S_f, D)$$

and

Input:

$M = (S, \tau, S_0, S, S, D)$ – the model (Büchi automaton)

P – an LTL correctness property

Output:

Scenario automaton M_s, where $\mathcal{L}(M_s) = \mathcal{L}(M) \backslash \mathcal{L}(p)$

Algorithm: `GenerateScenarioAutomaton`(M, P)

– Convert $\neg p$ to a Büchi automaton N_p

(1) $N_P \leftarrow$ `ConvertToBüchi`$(\neg P)$;

– Construct an intersection of M and N_P

(2) $I = (S^i, \tau^i, S^i_0, S^i_a, S^i_f, D^i) \leftarrow M \cap N_P$;

– Compute the SCC graph of I

(3) $SCC_I = (\mathcal{C}, E_\tau, \mathcal{C}_0) \leftarrow$ `TarjanSCC`(I);

– Define a predicate on $\mathcal{C}$ identifying SCCs with a final state

– or an acceptance cycle

(4) $Q_s \leftarrow \lambda\, c : \mathcal{C}.(\exists s \in c.(s \in S^i_f \ \vee \ (c \textit{ has an acceptance cycle})))$;

– Compute the set of components where $Q^\exists_s$ is true

(5) $\mathcal{C}_s \leftarrow$ `RSET-EXISTENTIAL`$(\mathcal{C}, E_\tau, \mathcal{C}_0, Q_s)$;

– $\mathcal{C}_s$ comprises the set of states of the result scenario automaton

(6) $S_s \leftarrow \bigcup_{c \in \mathcal{C}_s} c$.

– Restrict the transition relation to the states in S_s

(7) $\tau_s \leftarrow \tau^i \cap (S_s \times S_s)$.

– M_s consists of SCCs of I that can reach a component with an acceptance state

– or a final state

(8) $M_s \leftarrow (S_s, \tau_s, S^i_0 \cap S_s, S^i_a \cap S_s, S^i_f \cap S_s, D^i)$.

(9) Return M_s.

Figure 3.4: Explicit-State Algorithm for Generating Scenario Automata

$$N_P = (S_p, \tau_p, S^0_p, S^a_p, S^f_p, \oslash)$$

be the Büchi automata representing the model and the negation of the correctness property, respectively. When checking correctness, we want to include every possible finite and infinite execution of M, so every state of M is both accepting and final:

$$S_a = S$$

$$S_f = S$$

We use a simplified definition of Büchi automata intersection that is valid whenever all states of one automaton are all accepting and final. The intersection of M and N_P is defined as follows:

$$M_s = M \cap N_P = (S \times S_p, \tau, S^0_m \times S^0_p, S \times S^a_p, S \times S^f_p, D)$$

Here the transition relation τ is such that $((r_1, q_1), t, (r_2, q_2)) \in \tau$ if and only if $(r_1, t, r_2) \in \tau_m$ and $(q_1, t, q_2) \in \tau_p$. A state (r, q) is accepting in $M \cap N_P$ if and only if state q is accepting in N_P. Likewise, a state (r, q) is final in $M \cap N_P$ if and only if state q is final in N_P.

3.3.3 Computing Full Scenario Automata

An algorithm for generating scenario automaton M_s from a Büchi model M and an LTL correctness property P is shown in Figure 3.4 and illustrated in Figure 3.5. The algorithm uses Gerth *et.al.*'s procedure [39] to convert the LTL correctness formula P to a Büchi automaton N_P (line 1) and computes the intersection $I = M \cap N_P$ (line 2).

The intersection automaton I accepts the set of failing scenarios of the model M. However, it also represents the entire state space of the model. The next several steps of the algorithm separate the states of I that do no participate in any failing scenarios. Step 3 takes the intersection automaton I (Figure 3.5(a)) and computes its SCC graph (Figure 3.5(b)), using a version of the classic SCC algorithm due to R. Tarjan [90].

Let any cycle containing at least one acceptance state be called an *acceptance cycle*. Steps 4 and 5 use the RSET−EXISTENTIAL algorithm from Section 3.3.1 to compute the set of strongly connected components $\mathcal{C}_s$ such that from each state $s \in \bigcup_{c \in \mathcal{C}_s} c$ it is possible to reach an SCC with either a final state or an acceptance cycle. To find these strongly connected components, RSET−EXISTENTIAL is invoked with the following predicate on the components of SCC_I:

$$Q_s = \lambda\, c : \mathcal{C}\ .\ (\exists s \in c\ .\ (s \textit{ is final in } I))\ \vee\ (c \textit{ has an acceptance cycle})$$

In Figure 3.5(c) the set $\mathcal{C}_s$ of SCCs found in step 5 is shaded. Components 2 and 3 contain an acceptance cycle. Component 5 has final states. Components 1 and 4 are included in the set because from any state in those components it is possible to reach component 5.

The SCCs found by RSET−EXISTENTIAL comprise the set of states of the result scenario automaton (Figure 3.5(d)). To ensure succinctness, we discard the components of I that do not belong to the set $\mathcal{C}_s$ (components 6, 7, and 8 in Figure 3.5(c)).

The following Lemmas show that the algorithm in Figure 3.4 produces sound, exhaustive, and succinct scenario automata.

Lemma 3.8 ***(explicit-sound)*** *If α is an execution in the the output automaton $M_s = (S_s, \tau_s, S_0^s, S_f^s, L_s)$, then α violates the correctness property P in the input automaton $M = (S, \tau, S_0, S, S, D)$: $L(M_s) \subseteq L(M) \backslash P$.*

Proof:

Let $\alpha = s_0 t_0 \ldots t_{n-1} s_n \ldots$ be an execution of the output automaton M_s. Then α is also an execution of $I = M \cap N_P$. Thus, α is an execution of M and violates the property P.

□

Lemma 3.9 ***(explicit-exhaustive)*** *If an execution α of the input automaton $M = (S, \tau, S_0, S, S, D)$ violates the correctness property P, then α is accepted by the output automaton $M_s = (S_s, \tau_s, S_0^s, S_f^s, L_s)$: $L(M_s) \supseteq L(M) \backslash P$.*

Proof:

Let $\alpha = s_0 t_0 \ldots t_{n-1} s_n \ldots$ be an execution of the input model M such that α violates the correctness property P. Since α violates P, every state and transition in α will be present in the intersection automaton $I = M \cap N_P$ ([19], pp. 123-125). To prove that α is an execution in M_s, we first show that every state and every transition in α is present in M_s. There are two cases.

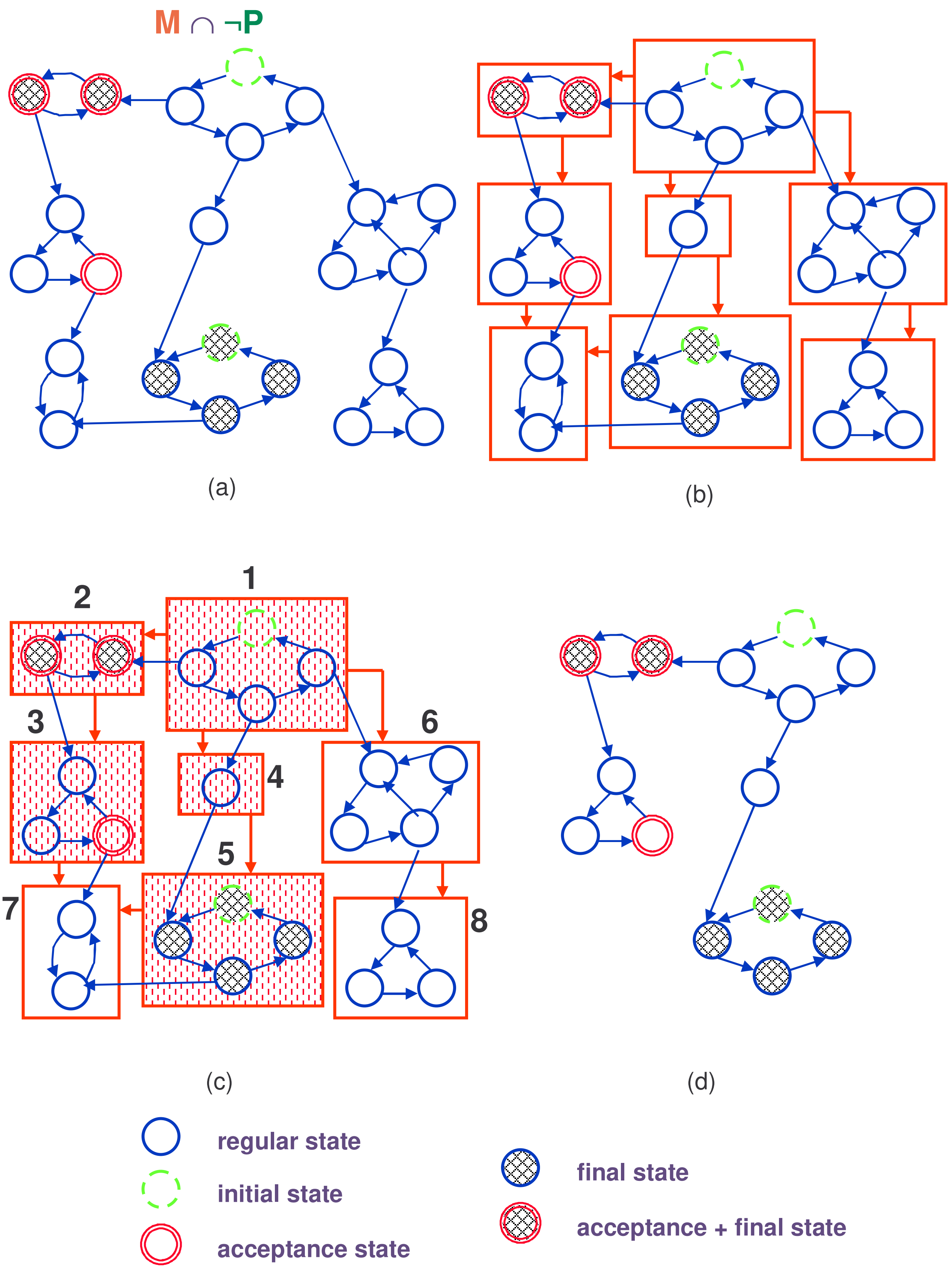


Figure 3.5: Explicit Algorithm Example

Case 1 Suppose that α is infinite. Then there must be an acceptance cycle of I that α traverses infinitely often. Let s_a be the an acceptance state in this cycle. All states and transitions following s_a in α must be in the strongly connected component c_a of I that contains s_a. Since the component c_a contains an acceptance cycle, step 5 adds c_a to $\mathcal{C}_s$, and subsequently steps 6 and 7 add the states and transitions of c_a to M_s.

All states and transitions preceding s_a in α belong to strongly connected components from which it is possible to reach s_a. By Theorem 3.3 (rset-existential-correct), in step 5 all of those components will be added to the set $\mathcal{C}_s$. So steps 6 and 7 also add the states and transitions belonging to those components to M_s.

Case 2 Suppose that α is finite. Let s_f be the first final state of I reached by α. All states and transitions following s_f in α must be in the strongly connected component c_f of I that contains s_f. Step 5 adds c_f to $\mathcal{C}_s$, and subsequently steps 6 and 7 add the states and transitions of c_f to M_s.

All states and transitions preceding s_f in α belong to strongly connected components from which it is possible to reach s_f. By Theorem 3.3 (rset-existential-correct), in step 5 all of those components will be added to the set $\mathcal{C}_s$. So steps 6 and 7 also add the states and transitions belonging to those components to M_s.

It remains to show that α is accepted by M_s. α violates the correctness property P, so it is accepted by the negated property automaton N_P, and therefore also by the intersection $I = M \cap N_P$. All accepting states of I that also exist in M_s are accepting in M_s. Similarly, all final states of I that also exist in M_s are final in M_s. Since α is accepted by I, it must also be accepted by M_s.

□

Lemma 3.10 ***(explicit-succinct state)*** *A state s of the input automaton $M = (S, \tau, S_0, S, S, D)$ is a state in the output automaton $M_s = (S_s, \tau_s, S_0^s, S_a^s, S_f^s, D_s)$ if and only if there is an execution α of the input automaton $M = (S, \tau, S_0, S, S, D)$ that violates the correctness property P and contains s.*

Proof:

($\Rightarrow$) Let s be a state in the output automaton $M_s = (S_s, \tau_s, S_0^s, S_f^s, L_s)$. By construction in steps 4-7, s is either a member of a reachable SCC that contains a final state or an acceptance cycle, or it is a member of a reachable SCC from which it is possible to reach another SCC that contains an acceptance cycle or a final state. In either case, there exists a path α in M_s that starts in an initial state, goes through s, and then either ends in a final state or loops around an SCC visiting an acceptance state infinitely often. By definition, the path α is an execution of M_s. By lemma 3.8, α is an execution of the input automaton M and it violates correctness property P.

($\Leftarrow$) If there is an execution in M_s that contains s, then trivially s is in M_s. □

Lemma 3.11 ***(explicit-succinct transition)*** *A transition $t = (s_1, s_2) \in \tau$ of the input automaton $M = (S, \tau, S_0, S, S, D)$ is a transition in the output automaton $M_s = (S_s, \tau_s, S_0^s, S_a^s, S_f^s, D_s)$ if and only if there is an execution α of the input automaton $M = (S, \tau, S_0, S, S, D)$ that violates the correctness property P and contains t.*

Proof:

($\Rightarrow$) Let t be a transition in the output automaton $M_s = (S_s, \tau_s, S_0^s, S_a^s, S_f^s, L_s)$. The proof that t must lie on an execution of M_s that is also an execution of M and violates P is analogous to the proof of Lemma 3.10, ($\Rightarrow$).

($\Leftarrow$) If there is an execution in M_s that contains t, then trivially t is in M_s. □

Theorem 3.4 *The algorithm in Figure 3.4 generates sound, exhaustive, and succinct scenario automata.*

Proof:

Follows directly from Lemmas 3.9, 3.8, 3.10, and 3.11 and Definitions 2.6, 2.7, and 2.8.

□

3.3.4 Computing RS-Automata for Finite Scenarios

Now we turn our attention to the task of producing rs-automata that are smaller in size than the full scenario automata computed by the algorithm `GenerateScenarioAutomaton`. For simplicity, we will treat finite and infinite executions separately.

Given a Büchi model M and an LTL correctness property P, our basic approach still relies on extracting a suitable sub-graph of the intersection automaton $I = M \cap \neg P$. To begin, we prove some facts about f-representative scenarios with respect to I.

Lemma 3.12 *Given a Büchi automaton M and a correctness property P, a finite execution $\gamma = s_0 t_0 \ldots t_{n-1} s_n$ in M is f-representative with respect to P if and only if every state reachable from s_n in the automaton $I = M \cap \neg P$ is final.*

Proof:

($\Rightarrow$) Let $\gamma = s_0 t_0 \ldots t_{n-1} s_n$ be an f-representative execution in M with respect to P. Let s be any state reachable from s_n in I via some path β. The finite execution $\alpha = \gamma\beta$ has an f-representative prefix γ, so by Definition 2.9 the execution α is failing with respect to property P. Automaton I accepts all failing executions of M, so it must accept the execution α. Therefore, the terminal state s of α is final in I.

($\Leftarrow$) Let $\gamma = s_0 t_0 \ldots t_{n-1} s_n$ be an finite execution in M with respect to P. Assume that every state reachable from γ's terminal state s_n in the automaton I is final. To prove that γ is f-representative with respect to P, we have to show that any finite scenario of M with prefix γ is failing with respect to P. Let $\alpha = \gamma\beta$ be a finite scenario of M. By assumption the terminal state of α is final in I, so α is accepted by I. I accepts only failing scenarios of M with respect to P, therefore α fails with respect to P.

□

Given a Büchi automaton M and a correctness property P, let $I = M \cap \neg P$ and let $SCC_I = (\mathcal{C}, E_\tau, \mathcal{C}_0)$ be the SCC graph of automaton I. Define the following predicate on the components of SCC_I:

$$Q_f = \lambda\, c : \mathcal{C}\ .\ (\forall s \in c\ .\ s \textit{ is final in I})$$

Then we have the following Corollary to Lemma 3.12:

Corollary 3.1 *A finite execution $\gamma = s_0 t_0 \ldots t_{n-1} s_n$ in M is f-representative with respect to P if and only if in the SCC graph SCC_I, $Q_f^\forall$ is true for the component containing state s_n.*

Input:

$M = (S, \tau, S_0, S, S, D)$ – the model (Büchi automaton)

P – an LTL correctness property

Output:

Rs-automaton $M_P^r = (M_r, S_{rf}, \oslash)$

Algorithm: `F-RSAutomaton`(M, P)

– *Convert $\neg p$ to a Büchi automaton N_p*

(1) $N_P \leftarrow$ `ConvertToBüchi`$(\neg P)$;

– *Construct an intersection of M and N_P*

(2) $I = (S^i, \tau^i, S_0^i S_0^i, S_a^i, S_f^i, D^i) \leftarrow M \cap N_P$;

– *Compute the SCC graph of I*

(3) $SCC_I = (\mathcal{C}, E_\tau, \mathcal{C}_0) \leftarrow$ `TarjanSCC`(I);

– *Define a predicate on $\mathcal{C}$ identifying SCCs with a final state*

(4) $Q_s \leftarrow \lambda\, c : \mathcal{C}.(\exists s \in c.(s \in S_f^i))$;

– *Define a predicate on $\mathcal{C}$ identifying SCCs where every state is final*

(5) $Q_f \leftarrow \lambda\, c : \mathcal{C}.(\forall s \in c.(s \in S_f^i))$;

– *Compute the set of components where $Q_s^\exists$ is true*

(6) $\mathcal{C}_s \leftarrow$ `RSET-EXISTENTIAL`$(\mathcal{C}, E_\tau, \mathcal{C}_0, Q_s)$;

– *Compute the set of components where $Q_f^\forall$ is true*

(7) $\mathcal{C}_f \leftarrow$ `RSET-UNIVERSAL`$(\mathcal{C}, E_\tau, \mathcal{C}_0, Q_f)$;

– *Compute the set of components where $Q_s^\exists$ is true and $Q_f^\forall$ is false*

(8) $\mathcal{C}_1 \leftarrow \mathcal{C}_s \setminus \mathcal{C}_f$;

– *Compute the set of components where both $Q_s^\exists$ and $Q_f^\forall$ are true*

(9) $\mathcal{C}_2 \leftarrow \mathcal{C}_s \cap \mathcal{C}_f$;

– *Gather all the states in $\mathcal{C}_1$*

(10) $S_1 \leftarrow \bigcup_{c \in \mathcal{C}_1} c$;

– *Isolate the initial states of $\mathcal{C}_2$*

(11) $S_2 \leftarrow S_0^i \cap (\bigcup_{c \in \mathcal{C}_2} c)$;

– *Isolate the border states of $\mathcal{C}_2$*

(12) $S_3 \leftarrow \{s \in (\bigcup_{c \in \mathcal{C}_2} c) \mid \exists s' \in S_1 \,.\, (s', s) \in \tau^i\}$;

– *The rs-automaton includes all states in $\mathcal{C}_1$, initial states of $\mathcal{C}_2$, and border states of $\mathcal{C}_2$*

(13) $S_s \leftarrow S_1 \cup S_2 \cup S_3$.

– *Restrict the transition relation to the states in S_s*

(14) $\tau_s \leftarrow \{(s_1, s_2) \in \tau^i \mid (s_1 \in S_s \ \wedge \ s_2 \in S_s)\}$.

(15) $M_r \leftarrow (S_s, \tau_s, S_0^i \cap S_s, \oslash, S_f^i \cap S_s, D^i)$.

– *The set S_{rf} of f-scenario end-points consists of initial states of $\mathcal{C}_2$ and border states of $\mathcal{C}_2$*

(16) Return $M_P^r = (M_r, S_2 \cup S_3, \oslash)$.

Figure 3.6: Algorithm for Generating F-Exhaustive RS-Automata

Corollary 3.1 suggests a way to modify the algorithm `GenerateScenarioAutomaton` to compute an f-exhaustive rs-automaton $M_P^r = (M_r, S_{rf}, S_{ri})$ instead of the full scenario automaton M_s. The modified algorithm `F-RSAutomaton` is shown in Figure 3.6 and illustrated in Figure 3.7. The result rs-automaton accepts only f-scenarios and finite r-scenarios; we treat infinite scenarios separately in Section 3.3.5.

The algorithm begins by computing the intersection automaton $I = M \cap \neg P$ (Figure 3.7(a)) and its strongly connected component graph SCC_I (Figure 3.7(b)). Lines 4 and 5 define two predicates on the components in SCC_I, Q_s and Q_f. The predicate Q_s identifies SCCs with at least one final state. The predicate Q_f identifies SCCs where every state is final.

Lines 6 and 7 compute two sets of components using the *RSET* algorithms. Set $\mathcal{C}_s$ consists of components where $Q_s^\exists$ holds, and set $\mathcal{C}_f$ consists of components where $Q_f^\forall$ holds. Next, the algorithm splits the SCCs of I into three classes:

1. Components where $Q_s^\exists$ is true and $Q_f^\forall$ is false (set $\mathcal{C}_1$).
2. Components where both $Q_s^\exists$ and $Q_f^\forall$ are true (set $\mathcal{C}_2$).
3. All other components.

The three component classes are shown in Figure 3.7(c) with distinct shades. Components in class 1 consist of states from which it is possible to reach a final state. Furthermore, since $Q_f^\forall$ is false for class 1 components, by Corollary 3.1 f-representative scenarios cannot terminate inside class 1 components. It is therefore necessary to include class 1 components in the result rs-automaton in their entirety (state set S_1, lines 10 and 13). The automaton in Figure 3.7(c) has four class 1 components, numbered 1 through 4. Every state and transition in those components is included in the result automaton in Figure 3.7(d).

Components in class 2 can contain initial states. One such state is shown in Figure 3.7(c) in component 5. In all component 2 initial states $Q_f^\forall$ is true. By Corollary 3.1 finite scenarios of length 1 starting and ending in such initial states are f-representative. The algorithm places initial states from class 2 components in the result rs-automaton (state set S_2, lines 11 and 13). It should be noted that each initial state in a class 2 component signifies that every finite scenario starting in the state fails with respect P, meaning that the model M violates P in its initial state. Therefore, in practice class 2 components are unlikely to contain initial states.

Since $Q_f^\forall$ is true in all of the states in class 2 components, Corollary 3.1 guarantees that any finite scenario ending in a class 2 component is f-representative. Therefore, it is sufficient to include in the result rs-automaton only those border states of class 2 components that have an incoming transition from a class 1 component (state set S_3, lines 12 and 13). Components 5 and 7 in Figure 3.7(c) have one border state each. Both border states are included in the result automaton in Figure 3.7(d).

Finally, components in class 3 have no final states, so they do not participate in any finite scenarios accepted by automaton I and can therefore be safely discarded.

Three points about the three component classes are important in the proofs of correctness that follow. First, to justify the case analysis in the proofs we observe that the classes are mutually exclusive; an SCC can belong to one and only one of the three classes. Second, once a path enters a class 2 component, it can never re-enter a class 1 component. Third, once a path enters a class 3 component, it can never re-enter a component from any of the other three classes. These observations follow trivially from the definitions of the component classes.

Lemmas 3.13, 3.14, 3.15, and 3.16 refer to inputs and outputs of the algorithm `F-RSAutomaton`. The proofs refer to the three component classes defined above.

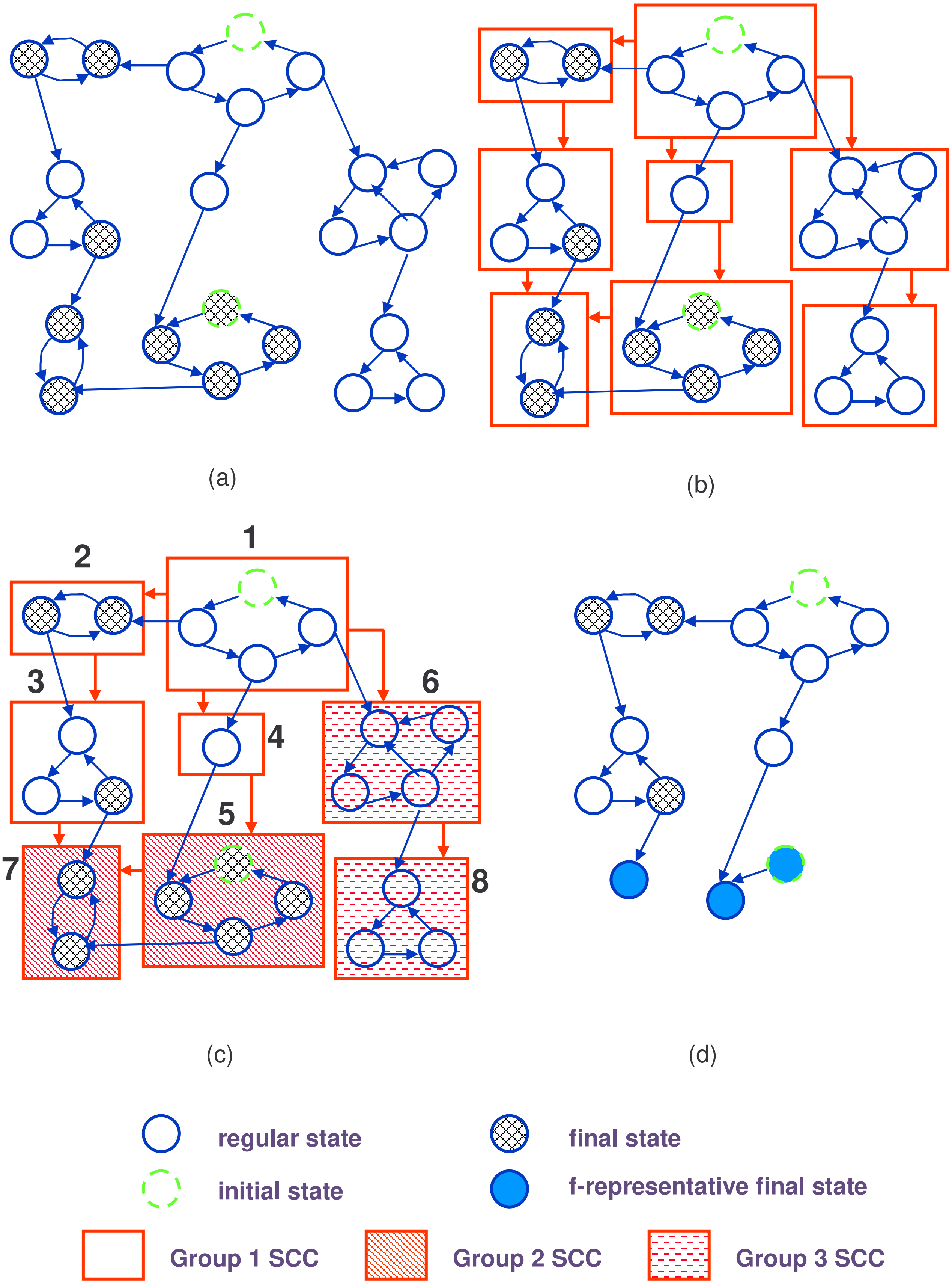


Figure 3.7: F-Exhaustive RS-Automaton Algorithm Example

Lemma 3.13 ***(f-rs-sound)*** *If γ is an execution in the the output rs-automaton $M_P^r = (M_r, S_{rf}, \oslash)$, then γ is failing scenario of the input automaton M with respect to P.*

Proof:

Let $\gamma = s_0 t_0 \ldots t_{n-1} s_n$. Since γ is accepted by the output rs-automaton M_P^r, the state s_n must be final in M_P^r. Therefore, s_n is also final in I (step 15). So γ is accepted by I, and must be a failing scenario of M with respect to P.

□

Lemma 3.14 ***(f-rs-exhaustive)*** *If a finite execution α of the input automaton $M = (S, \tau, S_0, S, S, D)$ violates the input correctness property P, then for the output rs-automaton $M_P^r = (M_r, S_{rf}, \oslash)$ one of the following holds:*

1. α is an r-scenario in M_P^r, or

2. an f_n-prefix of α is an f-scenario in M_P^r for some value of n

Proof:

Let $\alpha = s_0 t_0 \ldots t_{n-1} s_n$ be a finite execution of the input model such that α violates the correctness property P. Since α violates the correctness property P, every state and transition in α will be present in the intersection automaton $I = M \cap N_P$ ([19], pp. 123-125).

Consider the initial and the terminal states of α with respect to the SCC graph of intersection automaton I. The following case analysis covers the possible locations for these two states.

Case 1 Both s_0 and s_n are in class 1 components. In this case, the entire execution α must go through class 1 components, so the algorithm includes all states of α in the output rs-automaton M_P^r (lines 10 and 13). By construction, all the final states of I that also exist in M_P^r are final in M_P^r (line 15), so state s_n is final in M_P^r. Therefore, α is an r-scenario in M_P^r.

Case 2 s_0 is in a class 2 component. In this case the algorithm adds state s_0 to M_P^r in steps 11 and 13. Furthermore, s_0 is placed in the set S_{rf} of f-scenario termination points in step 16. Finally, all states belonging to class 2 components are final in I and therefore final in M_P^r (step 15). So s_0 is final in M_P^r. We conclude that the trivial execution consisting of state s_0, which is the f_1-prefix of α, is an f-scenario in M_P^r.

Case 3 s_0 is in a class 1 component and s_n is in a class 2 component. In this case at some point α must transition from a state s_k belonging to a class 1 component to a state s_{k+1} belonging to a class 2 component. All the states prior to s_k belong to class 1 components, so they are added to the output rs-automaton in steps 10 and 13. Steps 12 and 13 add state s_{k+1} to the automaton. In addition, step 16 adds state s_{k+1} to the set S_{rf}. Finally, state s_{k+1} is final in I, so it is also final in M_P^r. Therefore, the finite execution $\gamma = s_0 t_0 \ldots t_{k-1} s_k t_k s_{k+1}$ is an f-scenario in M_P^r. We know that $Q_f^\forall$ is true in the component containing state s_{k+1}, so by Corollary 3.1 execution γ is f-representative in M. We conclude that γ is an f_{k+1}-prefix of α and an f-scenario in M_P^r, as required.

Case 4 The remaining cases are impossible. The terminal state s_n of α is final, so α cannot go through any states in a class 3 component.

□

Lemma 3.15 ***(f-rs-succinct state)*** *A state s of the input automaton $M = (S, \tau, S_0, S, S, D)$ is a state in the output rs-automaton $M_P^r = (M_r, S_{rf}, \oslash)$ if and only if there is a finite execution α of* *the input automaton $M = (S, \tau, S_0, S, S, D)$ that violates the correctness property P and contains s.*

Proof:

($\Rightarrow$) Let s be a state in the output rs-automaton $M_P^r = (M_r, S_{rf}, \oslash)$. By construction, s is is a member of a reachable SCC from which it is possible to reach another SCC that contains a final state. So there exists a path α in M_P^r that starts in an initial state, goes through s, and ends in a final state. By definition, α is an execution of M_P^r. By lemma 3.13, α is an execution of the input automaton M and it violates correctness property P.

($\Leftarrow$) If there is an execution in M_P^r that contains s, then trivially s is in M_P^r. ∎

Lemma 3.16 ***(f-rs-succinct transition)*** *A transition $t = (s_1, s_2) \in \tau$ of the input automaton $M = (S, \tau, S_0, S, S, D)$ is a transition in the output rs-automaton $M_P^r = (M_r, S_{rf}, \oslash)$ if and only if there is a finite execution α of the input automaton $M = (S, \tau, S_0, S, S, D)$ that violates the correctness property P and contains t.*

Proof:

($\Rightarrow$) Let t be a transition in the output rs-automaton $M_P^r = (M_r, S_{rf}, \oslash)$. The proof that t must lie on an execution of M_s that is also an execution of M and violates P is analogous to the proof of Lemma 3.15, ($\Rightarrow$).

($\Leftarrow$) If there is an execution in M_s that contains t, then trivially t is in M_s. ∎

Theorem 3.5 *The algorithm in Figure 3.6 generates sound, f-exhaustive, and succinct rs-automata.*

Proof:

Follows directly from Lemmas 3.14, 3.13, 3.15, and 3.16 and Definitions 2.18, 2.20, and 2.8.

∎

3.3.5 Computing RS-Automata for Infinite Scenarios

Lemma 3.12 and Corollary 3.1 have counterparts for i-representative scenarios.

Lemma 3.17 *Given a Büchi automaton M and a correctness property P, a finite execution $\gamma = s_0t_0 \ldots t_{n-1}s_n$ in M is i-representative with respect to P if and only if every cycle reachable from s_n in the automaton $I = M \cap \neg P$ contains an acceptance state.*

Proof:

($\Rightarrow$) Let $\gamma = s_0t_0 \ldots t_{n-1}s_n$ be an i-representative execution in M with respect to P. Let $\theta = s_0^c t_0^c \ldots t_{n-1}^c s_0^c$ be any cycle reachable from s_n in I via the path β. The infinite execution $\alpha = \gamma\beta\theta\theta\theta\ldots$ has an i-representative prefix γ, so by Definition 2.13 the execution α is failing with respect to property P. Automaton I accepts all failing executions of M, so it must accept the execution α. α must visit an acceptance state infinitely often, so the cycle θ contains an acceptance state.

($\Leftarrow$) Let $\gamma = s_0t_0 \ldots t_{n-1}s_n$ be an finite execution in M with respect to P. Assume that every cycle reachable from γ's final state s_n in the automaton I contains an acceptance state. To prove that γ is i-representative with respect to P, we have to show that any infinite scenario of M with prefix γ is failing with respect to P. Let $\alpha = \gamma\beta$ be an infinite scenario of M. β is an infinite suffix of α, and it must loop around at least one cycle infinitely often. By assumption that cycle contains an acceptance state in I, so α is accepted by I. I accepts only failing scenarios of M with respect to P, therefore α fails with respect to P.

□

Given a Büchi automaton M and a correctness property P, let $I = M \cap \neg P$ and let $SCC_I = (\mathcal{C}, E_\tau, \mathcal{C}_0)$ be the SCC graph of automaton I. Define the following predicate on the components of SCC_I:

$$Q_i = \lambda\, c : \mathcal{C}\ .\ (c \textit{ contains an acceptance cycle})$$

Then we have the following Corollary to Lemma 3.17:

Corollary 3.2 *A finite execution $\gamma = s_0 t_0 \ldots t_{n-1} s_n$ in M is f-representative with respect to P if and only if in the SCC graph SCC_I, $Q_i^\forall$ is true for the component containing state s_n.*

The algorithm `I-RSAutomaton` for generating i-exhaustive rs-automata is shown in Figure 3.6. It is almost identical to the algorithm for generating f-exhaustive automata. The output of the algorithm accepts only i-scenarios and infinite r-scenarios.

The algorithm begins by computing the intersection automaton $I = M \cap \neg P$ and its strongly connected component graph SCC_I. Lines 4 and 5 define two predicates on the components in SCC_I, Q_s and Q_i. The predicate Q_s identifies nontrivial SCCs with at least one acceptance state. The predicate Q_i identifies SCCs where every cycle has an acceptance state. The predicate Q_i can be evaluated on a component c algorithmically as follows: remove all acceptance states from component c and look for a cycle involving the remaining states (e.g. using depth first search).

Lines 6 and 7 compute two sets of components using the *RSET* algorithms. Set $\mathcal{C}_s$ consists of components where $Q_s^\exists$ holds, and set $\mathcal{C}_i$ consists of components where $Q_i^\forall$ holds. Next, the algorithm splits the SCCs of I into four classes:

1. Components where $Q_s^\exists$ is true and $Q_i^\forall$ is false (set $\mathcal{C}_1$).
2. Components where both $Q_s^\exists$ and $Q_i^\forall$ are true (set $\mathcal{C}_2$).
3. All other components.

Components in class 1 consist of states from which it is possible to reach an acceptance cycle. Furthermore, since $Q_i^\forall$ is false for class 1 components, by Corollary 3.2 i-representative scenarios cannot terminate inside class 1 components. It is therefore necessary to include class 1 components in the result rs-automaton in their entirety (state set S_1, lines 10 and 13).

Components in class 2 can contain initial states. In all of those initial states $Q_i^\forall$ is true. By Corollary 3.2 finite scenarios of length 1 starting and ending in such initial states are i-representative. The algorithm places initial states from class 2 components in the result rs-automaton (state set S_2, lines 11 and 13). As has been noted in Section 3.3.4, in practice class 2 components are unlikely to contain initial states.

Input:

$M = (S, \tau, S_0, S, S, D)$ – the model (Büchi automaton)

P – an LTL correctness property

Output:

Rs-automaton $M_P^r = (M_r, \oslash, S_{ri})$

Algorithm: `I-RSAutomaton`(M, P)

– *Convert* $\neg p$ *to a Büchi automaton* N_p

(1) $N_P \leftarrow$ `ConvertToBüchi`$(\neg P)$;

– *Construct an intersection of* M *and* N_P

(2) $I = (S^i, \tau^i, S_0^i S_0^i, S_a^i, S_f^i, D^i) \leftarrow M \cap N_P$;

– *Compute the SCC graph of* I

(3) $SCC_I = (\mathcal{C}, E_\tau, \mathcal{C}_0) \leftarrow$ `TarjanSCC`(I);

– *Define a predicate on* $\mathcal{C}$ *identifying SCCs with an acceptance cycle*

(4) $Q_s \leftarrow \lambda\, c : \mathcal{C}.(c \textit{ has an acceptance cycle})$;

– *Define a predicate on* $\mathcal{C}$ *identifying SCCs where every cycle has an acceptance state*

(5) $Q_i \leftarrow \lambda\, c : \mathcal{C}.(\forall \beta \in c \mid \beta \textit{ is a cycle } . (\exists s \in \beta \,.\, s \in S_a^i))$;

– *Compute the set of components where* $Q_s^\exists$ *is true*

(6) $\mathcal{C}_s \leftarrow$ `RSET-EXISTENTIAL`$(\mathcal{C}, E_\tau, \mathcal{C}_0, Q_s)$;

– *Compute the set of components where* $Q_i^\forall$ *is true*

(7) $\mathcal{C}_i \leftarrow$ `RSET-UNIVERSAL`$(\mathcal{C}, E_\tau, \mathcal{C}_0, Q_i)$;

– *Compute the set of components where* $Q_s^\exists$ *is true and* $Q_i^\forall$ *is false*

(8) $\mathcal{C}_1 \leftarrow \mathcal{C}_s \setminus \mathcal{C}_i$;

– *Compute the set of components where both* $Q_s^\exists$ *and* $Q_i^\forall$ *are true*

(9) $\mathcal{C}_2 \leftarrow \mathcal{C}_s \cap \mathcal{C}_i \cap \mathcal{C}_0$;

– *Gather all the states in* $\mathcal{C}_1$

(10) $S_1 \leftarrow \bigcup_{c \in \mathcal{C}_1} c$;

– *Isolate the initial states of* $\mathcal{C}_2$

(11) $S_2 \leftarrow S_0^i \cap (\bigcup_{c \in \mathcal{C}_2} c)$;

– *Isolate the border states of* $\mathcal{C}_2$

(12) $S_3 \leftarrow \{s \in (\bigcup_{c \in \mathcal{C}_2} c) \mid \exists s' \in S_1 \,.\, (s', s) \in \tau^i\}$;

– *The rs-automaton includes all states in* $\mathcal{C}_1$*, initial states of* $\mathcal{C}_2$*, and border states of* $\mathcal{C}_2$

(13) $S_s \leftarrow S_1 \cup S_2 \cup S_3$.

– *Restrict the transition relation to the states in* S_s

(14) $\tau_s \leftarrow \tau^i \cap (S_s \times S_s)$.

(15) $M_r \leftarrow (S_s, \tau_s, S_0^i \cap S_s, S_a^i \cap S_s, S_2 \cup S_3, D^i)$.

– *The set* S_{ri} *of i-scenario end-points consists of initial states in* $\mathcal{C}_2$ *and border states of* $\mathcal{C}_2$ *and* $\mathcal{C}_3$

(16) Return $M_P^r = (M_r, \oslash, S_2 \cup S_3)$.

Figure 3.8: Algorithm for Generating I-Exhaustive RS-Automata

Since $Q_i^{\forall}$ is true in all of the states in class 2 components, Corollary 3.2 guarantees that any finite scenario ending in a class 2 component is i-representative. Therefore, it is sufficient to include in the result rs-automaton only those border states of class 2 components that have an incoming transition from a class 1 component (state set S_3, lines 12 and 13). Finally, components in class 3 have no acceptance states, so they do not participate in any infinite scenarios accepted by automaton I and can therefore be safely discarded.

Lemmas 3.18, 3.19, 3.20, and 3.21 and Theorem 3.6 refer to inputs and outputs of the algorithm `I-RSAutomaton`. We omit the proofs, which are analogous to the proofs of Lemmas 3.13, 3.14, 3.15, and 3.16 and Theorem 3.5.

Lemma 3.18 ***(i-rs-sound)*** *If γ is an execution in the the output rs-automaton $M_P^r = (M_r, \oslash, S_{ri})$, then γ is either a failing scenario of the input automaton M with respect to P or an i-prefix of a failing scenario of M.*

Lemma 3.19 ***(i-rs-exhaustive)*** *If an infinite execution α of the input automaton $M = (S, \tau, S_0, S, S, D)$ violates the input correctness property P, then for the output rs-automaton $M_P^r = (M_r, \oslash, S_{ri})$ one of the following holds:*

1. *α is an r-scenario in M_P^r, or*
2. *an i_n-prefix of α is an i-scenario in M_P^r for some value of n*

Lemma 3.20 ***(i-rs-succinct state)*** *A state s of the input automaton $M = (S, \tau, S_0, S, S, D)$ is a state in the output rs-automaton $M_P^r = (M_r, \oslash, S_{ri})$ if and only if there is an infinite execution α of the input automaton $M = (S, \tau, S_0, S, S, D)$ that violates the correctness property P and contains s.*

Lemma 3.21 ***(i-rs-succinct transition)*** *A transition $t = (s_1, s_2) \in \tau$ of the input automaton $M = (S, \tau, S_0, S, S, D)$ is a transition in the output rs-automaton $M_P^r = (M_r, \oslash, S_{ri})$ if and only if there is an infinite execution α of the input automaton $M = (S, \tau, S_0, S, S, D)$ that violates the correctness property P and contains t.*

Theorem 3.6 *The algorithm in Figure 3.8 generates sound, i-exhaustive, and succinct rs-automata.*

Chapter 4

Performance

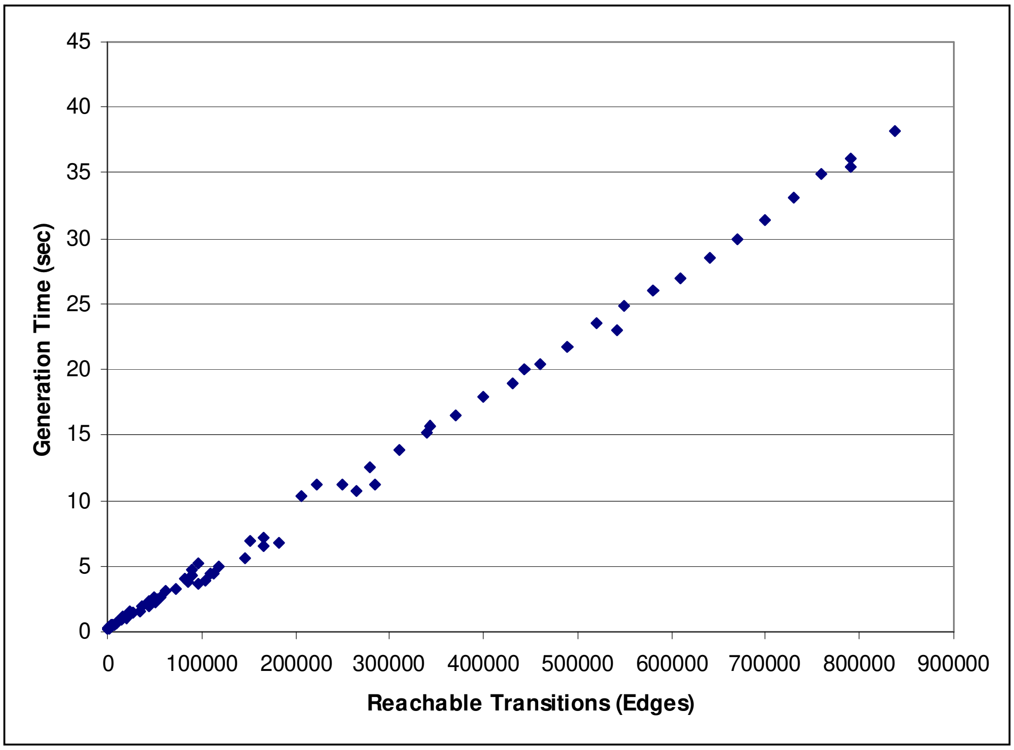
In this chapter we evaluate the performance of the scenario graph algorithms and consider implementation variations that trade off running time against state space coverage and memory consumption. The *traceback mode* variation described in Section 4.3.2 is a new scientific contribution to explicit state model checking.

4.1 Symbolic Algorithm Performance

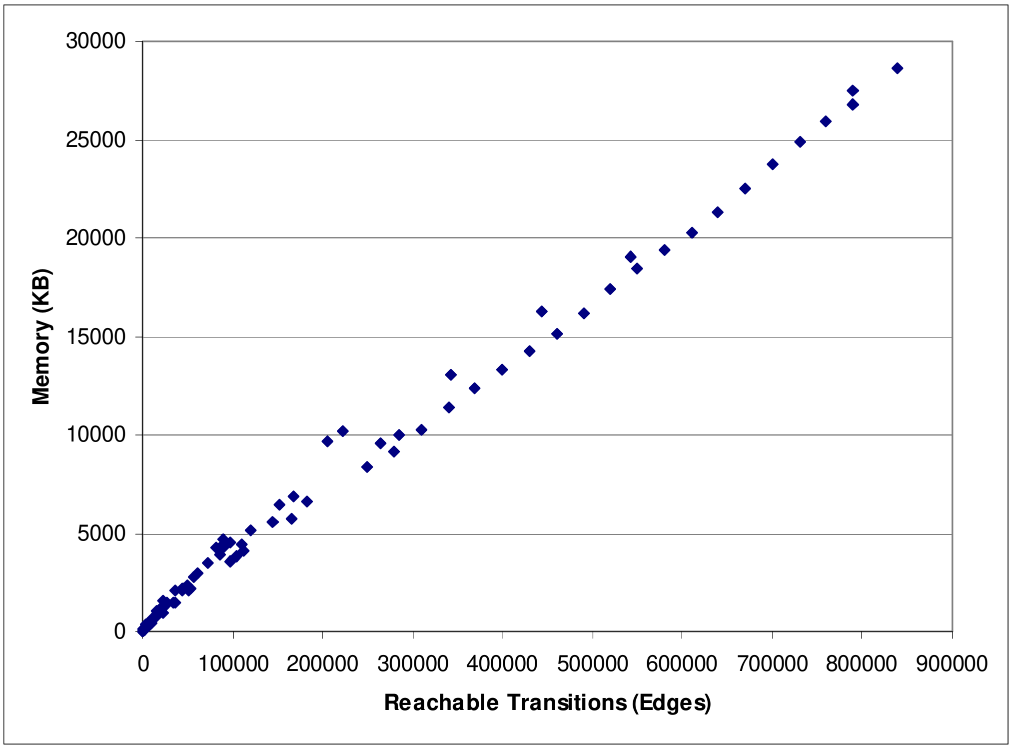
Preliminary tests on an implementation of the symbolic scenario graph algorithm indicated that the symbolic approach does not work well on the attack graph application that we used as our primary testing environment. For example, we could not initially process a small model with six hundred reachable states. The binary decision diagrams constructed for the model reached unmanageable size due to complex interactions between state variables. The model went through the model checker after we found a suitable variable ordering; however, it still took the symbolic algorithm over two hours to process the model. Based on the preliminary results, we did not conduct a thorough study of the symbolic algorithm's performance, choosing instead to focus on the explicit-state implementation.

4.2 Explicit-State Algorithm Performance

For clarity of presentation, the explicit-state algorithm in Figure 3.4 is split into several separate phases. Step 2 computes an intersection of two automata, step 3 computes the SCC graph of the result, step 5 traverses the SCC graph and marks components that will be included in the output scenario automaton, and finally steps 6 and 7 bring together the pieces of the output automaton. In practice, the tasks in steps 2, 5, 6, and 7 can be performed in parallel with step 3, i.e. during the search for strongly connected components. All of the steps can be accomplished using a single depth-first traversal of the reachable state space. Thus, the explicit-state algorithm has the same asymptotic complexity as a depth-first traversal of the state space graph. Asymptotic running time is $T(E) = S(E)O(E)$ [90], where E is the number of edges in the graph and $S(E)$ is the running time of the search algorithm used to detect when the current state in the search had been found previously. If a hash table is used to store states, searching for previously explored nodes can be done in amortized constant time, so the overall model checking algorithm runs in amortized $O(E)$ time.



(a)



(b)

Figure 4.1: Explicit State Model Checker Performance

We tested the performance of the explicit-state model checking algorithm implementation on 108 test cases. The largest test case produced a scenario graph with 46 thousand states and 838 thousand edges and took 38 seconds to generate.

Our empirical tests bear out the asymptotic bound. The test machine was a 1Ghz Pentium III with 1GB of RAM, running Red Hat Linux 7.3. Figure 4.1(a) plots running time of the implementation against the number of edges in the reachable state graph. The roughly linear relationship between the number of edges and running time is evident in the plot, and confirmed by statistical analysis. Least-squares linear regression estimates the linear coefficient at 1.12×10^{-4}, or 0.112 seconds of running time per one thousand explored edges of the reachable state graph. The coefficient of determination (R^2) is 0.9967, which means linear regression explains almost all of the variability in the data.

4.3 Explicit-State Algorithm Memory Usage

LTL model checking is PSPACE-complete [19], and memory is often the factor limiting the size of the model that we are able to verify. To ensure completeness of the search, all previously-explored states are kept until the end, so that we can detect when the search reaches a state that had been explored previously. Figure 4.1(b) plots memory consumption against the number of edges in the reachable state graph.

All of the work in the explicit-state algorithm for scenario graph generation (Section 3.3, Figure 3.4) is done in a single depth-first search pass through the state space of the model. The basic DFS procedure is shown in Figure 4.2. To traverse the state space starting in the initial state, we invoke the `DFS-Visit` procedure with the initial state as an argument. As the procedure discovers new nodes in the state space graph, it stores them in a table T. Before further exploring a new node, the algorithm consults the table to verify that the node has not been encountered previously.

Both performance and memory requirements of the algorithm depend critically on the implementation of table insertion (procedure `Insert`) and lookup (procedure `Explored`). If saving memory is critical, the table implementation can keep partial information about each state, achieving significant savings.

The simplest implementation choice is a classic hash table. The table stores the entire specification for each state. With a suitable hash function, both insertion and lookup operations require amortized $O(1)$ time. The DFS algorithm therefore takes amortized $O(E)$ time, where E is the number of edges in the state graph. This implementation guarantees full coverage of the state space, at the cost of high memory requirements. Each state kept in the table consumes the full number of bits required to represent it. The results in Figure 3.4 use a hash table.

```
DFS-Visit(v)

    for each child u of v
        if not Explored(T, u)
            Insert(T, v, u)
            DFS-Visit(u)
```

Figure 4.2: Basic Depth-First Search

4.3.1 Hashcompact Search Mode

The *hashcompact mode*, first proposed by Wolper and Leroy [98] and subsequently implemented in the SPIN model checker, is an alternative way of keeping state information. Instead of representing the entire state, only a hash value for each visited state is kept in the table. The `Explored` procedure looks for the hash of the current state in the table, returning *true* whenever the hash is found. This method retains the $O(E)$ asymptotic performance bound, but does not offer guarantees of full state coverage, as hash function collisions may cause the algorithm to skip unexplored states during the search. The advantage is that it is only necessary to keep a small, fixed number of bits per state in the table, drastically reducing memory requirements. There is some wiggle room: increasing the number of bits produced by the hash function produces a corresponding reduction in the probability of hash collisions.

4.3.2 Traceback Search Mode

The *traceback mode* is a new technique for explicit state space search. The mode retains collision detection functionality in the table implementation, reducing memory requirements at the cost of performance. In this mode the table keeps the minimum amount of information necessary to reconstruct a node s whenever a potential collision must be resolved. Specifically, it is sufficient to retain a pointer to the *parent* of s and the identity of the state machine transition that generated s. The parent pointer identifies the node from which s was first found during depth-first search.

When the algorithm detects a potential collision at node s, it reconstructs the DFS stack as it was when s was first found. The reconstruction procedure first pushes node s on the stack, then the parent of s, then the parent of the parent of s, etc., until an initial state is found. The procedure then traverses the stack, using the stored identity of the state machine transition at each node to reconstruct the full state at the node.

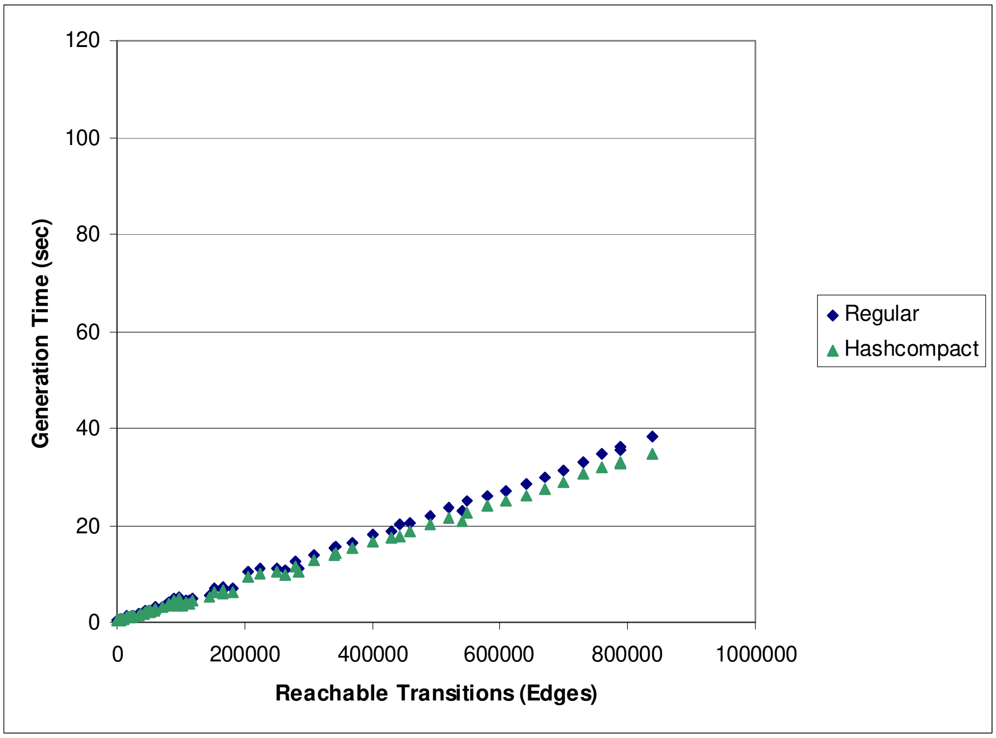
Assuming constant-size pointers, this method requires a small, fixed amount of memory per node and guarantees complete state space coverage. In the traceback mode collision detection and resolution (and by extension, the `Explored` operation) reconstructs the entire DFS stack each time it is invoked. Let the *diameter* of the state space graph be the longest possible cycle-free path that starts in an initial state. Then, the `Explored` operation is linear in the diameter d of the graph in the worst case, resulting in an $O(E)O(d)$ asymptotic performance bound for the main algorithm.

In summary, hashcompact and traceback modes offer trade-offs between memory consumption, performance, and state space coverage. The trade-offs are shown in the table below. S represents the number of bits necessary to store a single state of the model without loss of information.

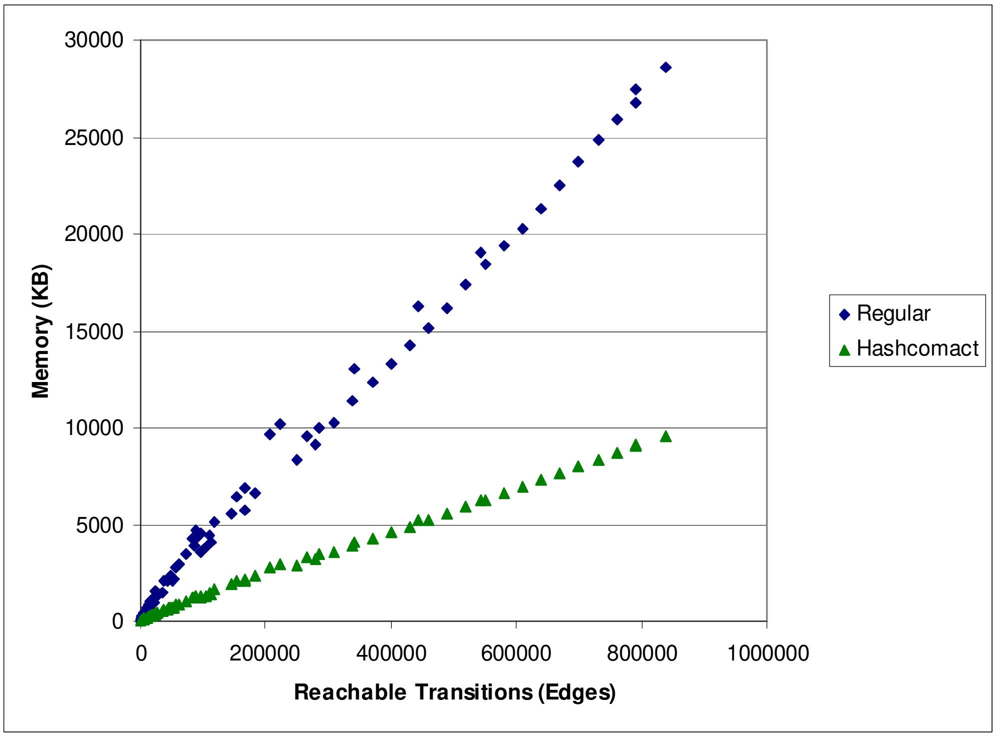
Mode	**Full state**	**Hashcompact**	**Traceback**
Amortized Running Time	$O(E)$	$O(E)$	$O(E)O(d)$
Memory per State	S	8 bytes	14 bytes
Coverage	Full	Partial	Full

4.4 Hashcompact Mode Trade-offs

We implemented both hashcompact and traceback modes in our scenario graph generator. In terms of running time and memory requirements, the hashcompact mode's performance is identical to the the regular, full state mode, as shown in Figure 4.3(a).



(a)



(b)

Figure 4.3: Hashcompact Mode Trade-offs

Hashcompact's memory footprint is significantly smaller. Figure 4.3(b) shows the memory savings of the hashcompact mode vs. regular search.

In exchange for memory efficiency, the hashcompact mode sacrifices correctness guarantees offered by the other modes. In general, the hashcompact search algorithm is neither sound nor exhaustive. We demonstrate this claim with a concrete example.

Let s_1, s_2, and s_3 be distinct states of the automaton $I = M \cap N_P$ and h the hash function used by the hash table. Assume further that $h(s_1) = h(s_2)$, there is a transition $t = (s_3, s_2)$ in the model, and the search discovers state s_1 prior to arriving at s_3. Upon discovering transition t, the search procedure checks to see if state s_2 had been visited previously. Due to the hash collision between s_1 and s_2, the algorithm erroneously assumes that s_2 had already been discovered, and adds an edge from s_3 to s_1 to the state graph.

The effects of this step are twofold. The search fails to consider state s_2 (and, potentially, the entire subgraph rooted at s_2), resulting in an incomplete search. Further, the added edge from s_3 to s_1 may not be a legitimate transition of I. In the resulting scenario graph, all executions containing this transition will represent non-existent failure scenarios. The hashcompact method can thus be seen as a risky heuristic search that makes no correctness claims but enables efficient exploration of larger state spaces than would be possible with full state space search.

4.5 Traceback Mode Trade-offs

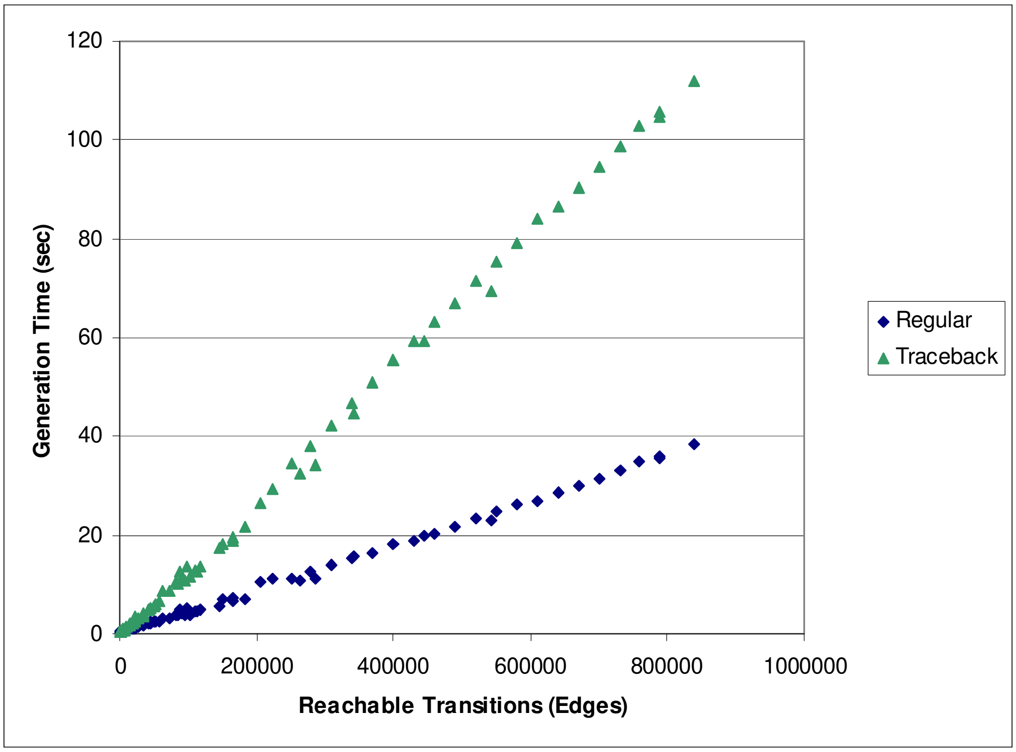
Figures 4.4(a,b) plot performance and memory requirements of the regular (full state) and traceback modes on the same set of models. The figures demonstrate the traceback trade-off between performance and memory consumption. The mode requires more time than regular search due to stack reconstruction overhead, but has a smaller memory footprint compared with the full state mode.

We have noted that the asymptotic performance bound for the traceback algorithm is $O(E)O(d)$, so the linear trend apparent in Figure 4.4(a) requires an explanation. The $O(d)$ factor comes from the state reconstruction step inside the `Explored` operation. The average per-edge overhead of this step depends on two factors:

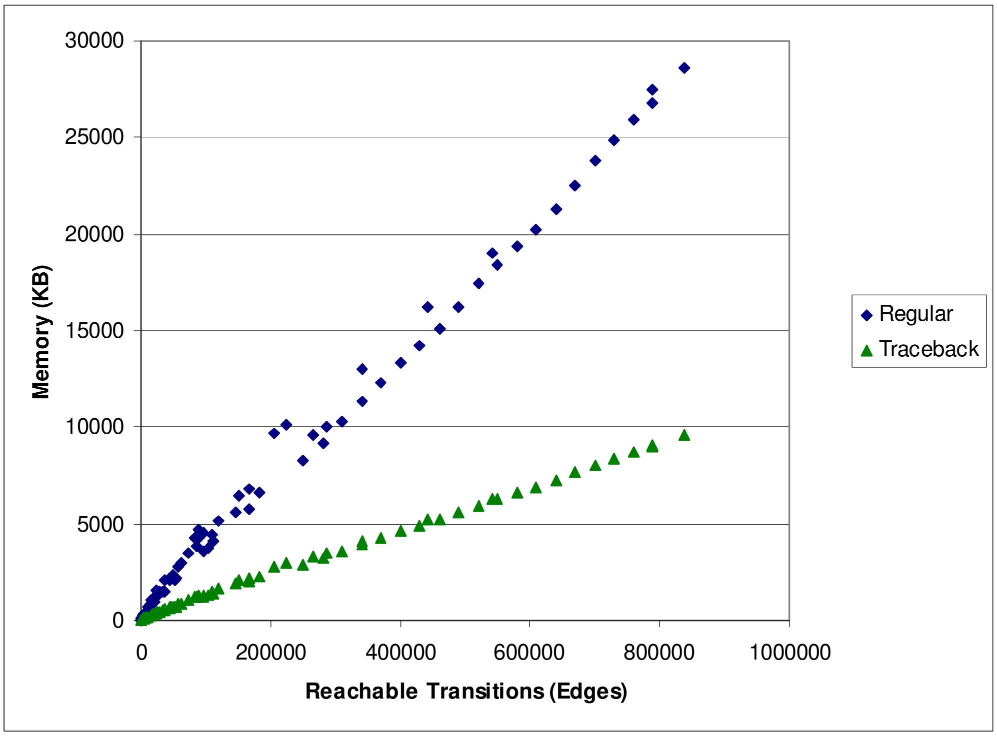
1. the average depth of the reconstructed node, and
2. the percentage of edges that require a reconstruction.

Figure 4.5(a) plots the total graph diameter and the average depth of a node that requires reconstruction against the number of reachable transitions. After some initial growth, both the graph diameter and the average depth of a reconstructed node stay approximately constant over the majority of the sample models, so the size of each reconstructed DFS stack does not grow appreciably as the size of the state space increases.

Figure 4.5(b) plots the proportion of edges that require a reconstruction, which is the second factor affecting the asymptotic bound. The proportion rapidly reaches about 95% and stays there over the majority of the sample models. This means that the second factor affecting the asymptotic bound also does not grow with the size of the model. Together, for our data set these two factors amount to roughly constant per-edge overhead for node reconstruction, which accounts for the linear trend in Figure 4.4(a).

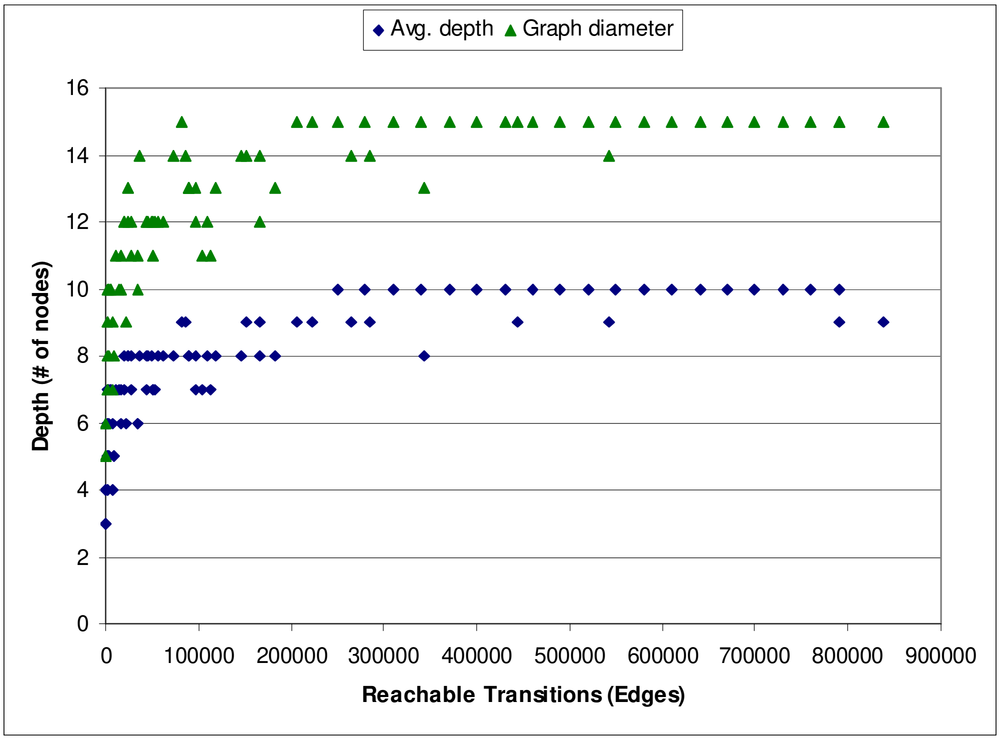


(a)

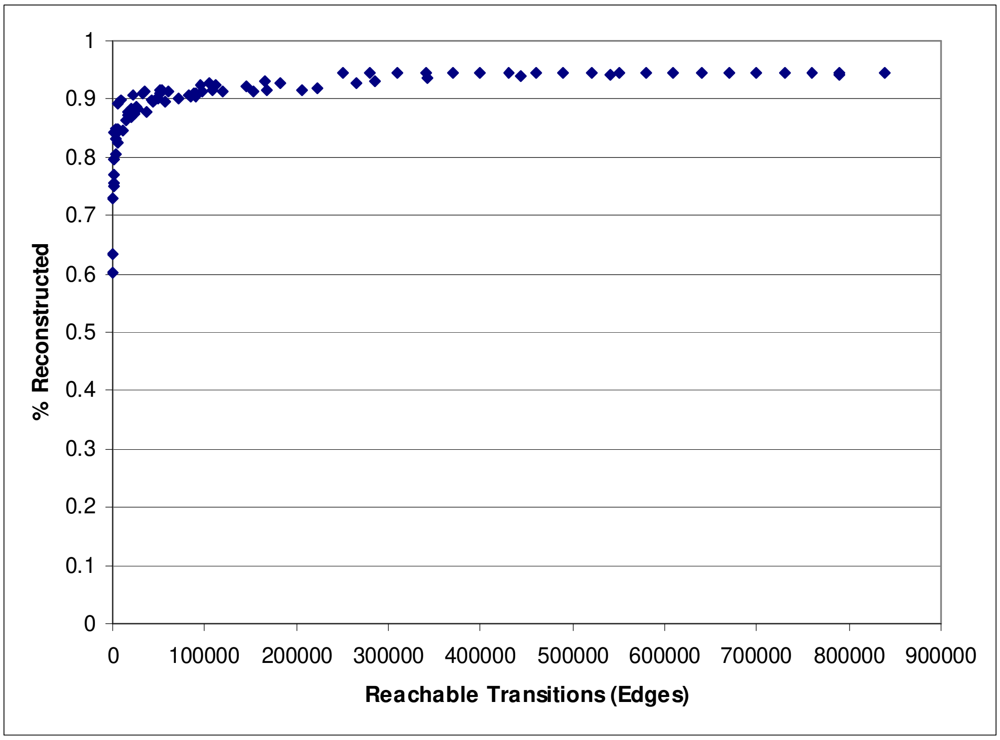


(b)

Figure 4.4: Traceback Mode Trade-offs



(a)



(b)

Figure 4.5: Traceback Mode Reconstruction Costs

Chapter 5

Scenario Graphs: Related Work

For over twenty years, model checking techniques have been studied extensively by many researchers. Because model-checking can be performed automatically, it is preferable to deductive verification, whenever it can be applied. One of the most attractive features of model checking as a computer aided verification technique is the ability to generate counterexamples that highlight for the user the often subtle errors in concurrent systems. To our knowledge, we are the first to study algorithms that aim specifically to generate the full set of counterexamples. Nevertheless, in this chapter we mention some recent related work. For completeness, we include a brief survey of traditional model checking techniques; material for the survey is drawn, with revisions, from [19].

5.1 Model Checking

The principal validation methods for complex systems are simulation, testing, deductive verification, and model checking. Simulation and testing both involve making experiments before deploying the systems in the field. While simulation is performed on an abstraction or a model of the system, testing is performed on the actual product. These methods can be a cost-efficient way to find many errors. However, checking of all possible interactions and potential pitfalls using simulation and testing techniques is rarely possible.

Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in a propositional temporal logic, and the reactive system is modeled as a state-transition graph. An efficient search procedure is used to determine if the specifications are satisfied by the state-transition graph. The technique was originally developed in 1981 by Clarke and Emerson [32, 18]. Quielle and Sifakis [76] independently discovered a similar verification technique shortly thereafter.

The model checking technique has several important advantages over mechanical theorem provers or proof checkers for verification of circuits and protocols. The benefit of restricting the problem domain to finite state models is that verification can be performed automatically. Typically, the user provides a high level representation of the model and the correctness specification to be checked. The procedure uses an exhaustive search of the state space of the system to determine if the specification is true or not. Given sufficient resources, the procedure will either terminate with the answer *true*, indicating that the model satisfies the specification, or give a counterexample execution that shows why the correctness formula is not satisfied. Moreover, it can be implemented with reasonable efficiency and run on typical desktops.

Temporal logics have proved to be useful for specifying concurrent systems, because they can describe the ordering of events in time without introducing time explicitly. Pnueli [74] was the first to use temporal logic for reasoning about concurrency. His approach involved proving properties of the program under consideration from a set of axioms that described the behavior of the individual statements of the program. The method was extended to sequential circuits by Bochmann [95] and Malachi and Owicki [64]. Since proofs were constructed by hand, the technique was often difficult to use in practice. The introduction of temporal-logic model checking algorithms by Clarke and Emerson [32, 33] in the early 1980s allowed this type of reasoning to be automated. Because checking that a single model satisfies a formula is much easier than proving the validity of a formula for all models, it was possible to implement this technique very efficiently. The algorithm developed by Clarke and Emerson for the branching-time logic CTL was polynomial in both the size of the model determined by the program under consideration and in the length of its specification in temporal logic. They also showed how fairness [38] could be handled without changing the complexity of the algorithm. This was an important step in that the correctness of many concurrent programs depends on some type of fairness assumption; for example, absence of starvation in a mutual exclusion algorithm may depend on the assumption that each process makes progress infinitely often.

The first step in program verification is to come up with a formal specification of the program. In a finite state system, each state is characterized by a finite amount of information, and this information can be described by certain atomic proposition. This means that a finite-state program can be viewed as a finite propositional Kripke structure and that it can be specified using propositional temporal logic. Thus, to verify the correctness of the program, one has only to check that the program, viewed as a finite Kripke structure, satisfies (is a model of) the propositional temporal logic specification. This was the approach of the early model checking algorithms [32, 33], further studied in [60, 34, 35].

At roughly the same time Quielle and Sifakis [76] gave a model checking algorithm for a subset of the branching-time temporal logic CTL, but they did not analyze its complexity. Later Clarke, Emerson, and Sistla [18] devised an improved algorithm that was linear in the product of the length of the formula and the size of the state transition graph. Early model checking systems were able to check state transition graphs with between 10^4 and 10^5 states. In spite of these limitations, model checking systems were used successfully to find previously unknown errors in several published circuit designs.

Sistla and Clarke [82, 83] analyzed the model checking problem for a variety of temporal logics and showed, in particular, that for linear temporal logic (LTL) the problem was PSPACE-complete. Pnueli and Lichtenstein [60] reanalyzed the complexity of checking linear-time formulas and discovered that although the complexity appears exponential in the length of the formula, it is linear in the size of the global state graph. Based on this observation, they argued that the high complexity of linear-time model checking might still be acceptable for short formulas. The same year, Fujita [36] implemented a tableau based verification system for LTL formulas and showed how it could be used for hardware verification.

CTL* is a very expressive logic that combines both branching-time and linear-time operators. The model checking problem for this logic was first considered in a paper by Clarke, Emerson, and Sistla [17], where it was shown to be PSPACE-complete, establishing that it is in the same general complexity class as the model checking problem for LTL. This result can be sharpened to show that CTL* and LTL model checking are of the same algorithmic complexity (up to a constant factor) in both the size of the state graph and the size of the formula. Thus, for purposes of model checking, there is no practical complexity advantage to restricting oneself to a linear temporal logic [35].

In the original implementation of the model checking algorithm, transition relations were represented explicitly by adjacency lists. For concurrent systems with small numbers of processes, the number of states was usually fairly small, and the approach was often quite practical. In systems with many concurrent parts, however, the number of states in the global state transition graph was too large to handle. The possibility of verifying systems with realistic complexity changed dramatically in the late 1980s with the discovery of how to represent transition relations using ordered binary decision diagrams (OBDDs) [8]. This discovery was made independently by three research teams [11, 24, 73]. The discovery made it possible to avoid explicitly constructing the state graph of the concurrent system. OBDDs provide a canonical form for boolean formulas that is often substantially more compact than conjunctive or disjunctive normal form, and very efficient algorithms have been developed for manipulating them. A *symbolic* model checker extracts a transition system represented as an OBDD from a program and uses an OBDD-based search algorithm to determine whether the system satisfies its specification. Because the symbolic representation captures some of the regularity in the state space determined by circuits and protocols, for those applications it is possible to verify systems with an extremely large number of states many orders of magnitude larger than could be handled by explicit-state algorithms. Various refinements of the OBDD-based techniques have pushed the state count up to more than 10^{120} [13, 12].

Alternative techniques for verifying concurrent systems have been proposed by a number of other researchers. Many of these approaches, including our treatment of explicit-state algorithms in Chapter 3, use automata for specifications as well as for implementations. Vardi and Wolper [93] first proposed the use of ω-automata (automata over infinite words) for automated verification. They showed how the linear temporal logic model checking problem could be formulated in terms of language containment between ω-automata. The basic idea underlying this approach is that for any temporal formula we can construct an automaton that accepts precisely the computations that satisfy the formula. The connection between propositional temporal logic and formal language theory has been quite extensively studied [38, 70, 82, 81, 97]. This connection is based on the fact that a computation is essentially an infinite sequence of states. Since every state is completely described by a finite set of atomic propositions, a computation can be viewed as an infinite word over the alphabet of truth assignments to the atomic propositions. Thus, temporal logic formulas can be viewed as finite-state acceptors. More precisely, given any propositional temporal formula, one can construct a finite automaton on infinite words [10, 66] that accepts precisely the sequences satisfied by the formula [97].

To use the above connection, we view a finite-state program as a finite-state generator of infinite words. Thus, if M is the program and P is the specification, then M meets P if every infinite word generated by M, viewed as a finite-state generator, is accepted by P viewed as a finite-state acceptor. This reduces the model-checking problem to a purely automata-theoretic problem: the problem of determining if the automaton $M \cap \neg P$ is empty, i.e., if it accepts no word.

Because the same type of model is used for both implementation and specification, an implementation at one level can also be used as a specification for the next level of refinement. The use of language containment is implicit in the work of Kurshan [2], which ultimately resulted in the development of the verifier COSPAN [41, 57, 42, 58]. Language containment is also the foundation for the popular SPIN model checker [50, 44, 49]. Other notions of conformance between the automata have also been considered, including observational equivalence and various refinement relations [21].

5.2 Improvements to Explicit-State Search

Model checkers based on language containment typically construct an explicit representation for each state as they search the state space of the model. The bitstate hashing (or supertrace) technique was introduced in 1987 [51] and elaborated in [50, 44] as a method to increase the quality of verification by explicit state search for large models that exhaust machine memory. The technique performs relatively high-coverage verifications within a memory area that may be orders of magnitude smaller than required for exhaustive verifications. The method has made it possible to apply formal verification techniques to problems that would normally have remained beyond the scope of explicit-state tools, e.g. [16, 15, 61, 47, 46].

The original bitstate hashing technique works by minimizing the amount of memory necessary for keeping the table of previously-visited states. The table is a table of bits all initially set to 0. To add a state to the table, one hashes the state description into an address in the table and sets the bit at this address to 1. To determine if a state is in the table, one applies the hash function to the state and checks whether the bit appearing at the computed address is 1.

The drawback of the method is that the hash function can compute the same address for distinct states and hence one can wrongly conclude that a state has been visited, whereas one has actually encountered a hash collision. Wolper and Leroy determined that the reduction in memory use comes at the price of a high probability of ignoring part of the state space and hence of missing existing errors [98]. They also proposed an improvement to the basic scheme, which they called the *hashcompact* method. The idea is to increase the number of bits stored in the table and to use a collision resolution scheme for the values stored. Wolper and Leroy claimed collision probabilities of 10^{-3} to 10^{-6} with memory use on the order of 40-100 bits per stored state.

Holzmann [48] provided an analysis of the different variants of bitstate hashing. He compared the original technique with two alternative variants, Wolper and Leroy's hashcompact and the *multihash* technique introduced by Stern and Dill [87, 88]. According to his results, the alternatives improve over the original bitstate hashing method in a limited domain of applications. The original bitstate hashing scheme tends to make fewer assumptions about its domain of application, and can outperform its competitors for large problem sizes and/or small memory bounds. Holzmann further showed that a variation of the multihash technique, sequential bitstate hashing, that can outperform the other algorithms in terms of state space coverage when applied to very large problem sizes, at the cost of increased running time. At the limit, sequential bitstate hashing can incur exponentially increasing runtime cost to approximate the results of an exhaustive search in shrinking memory.

We have implemented the hashcompact technique for reference and performance comparisons only, and do not claim to improve upon the existing bitstate hashing work. Our contribution, the traceback method, is an alternative means of trading performance for memory savings. The method guarantees full coverage of the state space, uses a small, fixed number of bits per stored state, and incurs only a polynomial runtime cost (the runtime is quadratic in the worst case).

5.3 Counterexamples

Modern model checkers produce counterexamples of a simple non-branching structure, i.e., finite or infinite paths (or *linear* counterexamples [20]). Despite the importance of counterexamples in the popularity of model checking, they have not been studied intensively. Algorithms for generating linear counterexamples and their complexity were first studied by Clarke et. al. [31] and Hojati et. al. [43]. Unfortunately, only properties in the weak temporal logic fragment ACTL $\cap$ LTL have linear counterexamples [63]. Results by Buccafurri et. al. [9] show that recognizing ACTL formulas with linear counterexamples is PSPACE-hard. Thus, linear counterexamples are not complete for ACTL (i.e., not every violation of an ACTL specification is witnessed by a linear counterexample), and recognizing the cases where linear counterexamples are applicable at all is hard.

In the most comprehensive effort to classify non-linear counterexamples, Clarke et. al. introduced a general framework for counterexamples [20]. They determined that the general form of counterexamples for ACTL has a tree-like structure, and showed that a large class of temporal logics admits counterexamples with a tree-like transition relation. This class includes the Buy One Get One Free Logic by Grumberg and Kupferman [56] and other temporal logics with ω-regular operators. Our scenario graphs consist of linear counterexamples only; incorporating non-linear counterexamples remains as future work.

Part II

Attack Graphs

Chapter 6

Introduction: Attack Graphs

As networks of hosts continue to grow, it becomes increasingly more important to automate the process of evaluating their vulnerability to attack. When evaluating the security of a network, it is rarely enough to consider the presence or absence of isolated vulnerabilities. Large networks typically contain multiple platforms and software packages and employ several modes of connectivity. Inevitably, such networks have security holes that escape notice of even the most diligent system administrator.

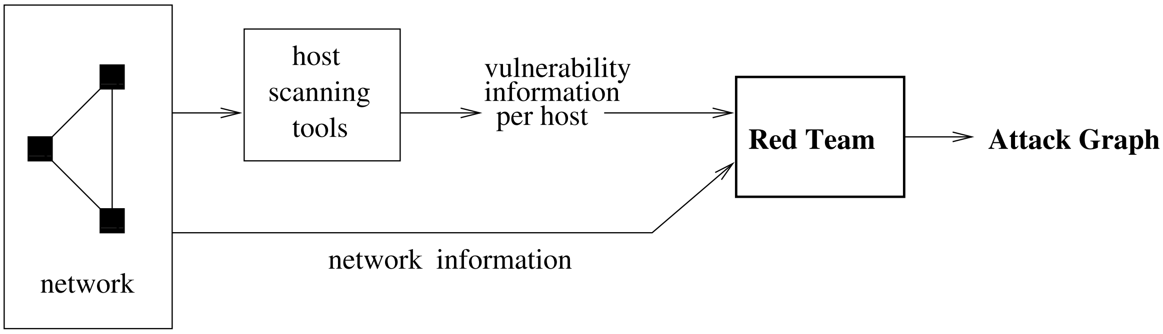


Figure 6.1: Vulnerability Analysis of a Network

To evaluate the security of a network of hosts, a security analyst must take into account the effects of interactions of local vulnerabilities and find global security holes introduced by interconnection. A typical process for vulnerability analysis of a network is shown in Figure 6.1. First, scanning tools determine vulnerabilities of individual hosts. Using this local vulnerability information along with other information about the network, such as connectivity between hosts, the analyst produces an *attack graph*. Each path in an attack graph is a series of exploits, which we call *actions*, that leads to an undesirable state. An example of an undesirable state is a state where the intruder has obtained administrative access to a critical host.

A typical result of such efforts is a hand-drawn "white board" attack graph, such as the one produced by a Red Team at Sandia National Labs for DARPA's CC20008 Information battle space preparation experiment and shown in Figure 6.2. Each box in the graph designates a single intruder action. A path from one of the boxes at the top of the graph to one of the boxes at the bottom is a sequence of actions corresponding to an attack scenario. At the end of any such scenario, the intruder has broken the network security in some way. The graph is included here for illustrative purposes only, so we omit the description of specific details.

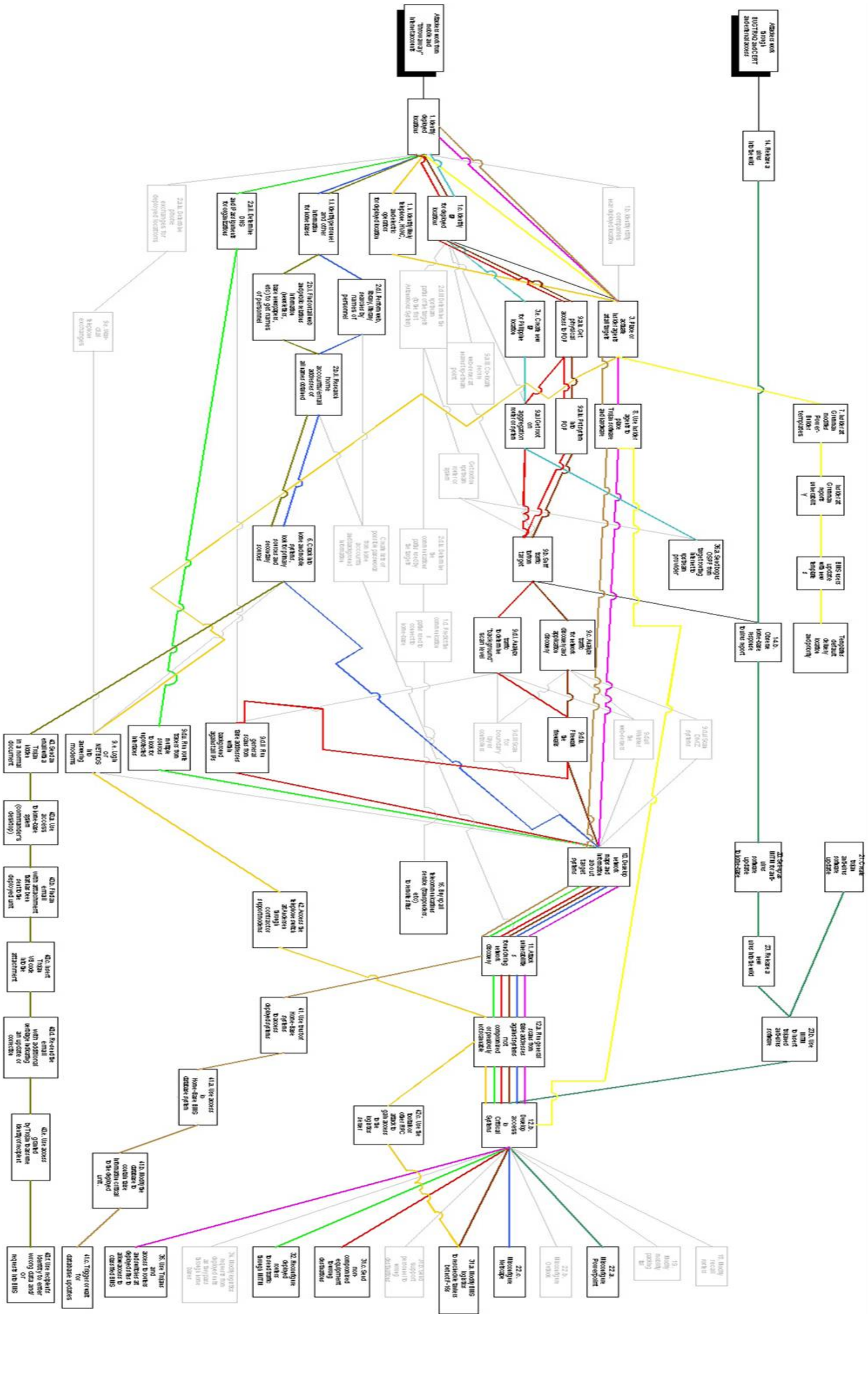


Figure 6.2: Sandia Red Team Attack Graph

Attack graphs can serve as a useful tool in several areas of network security, including intrusion detection, defense, and forensic analysis. To motivate our study of attack graphs as an application of scenario graph generation algorithms, we briefly review the potential applications to these areas of security.

Alert Correlation

System administrators are increasingly deploying intrusion detections systems (IDSs) to detect and combat attacks on their network. Such systems depend on software sensor modules that detect suspicious events and issue alerts. Configuring the sensors involves a trade-off between sensitivity to intrusions and the rate of false alarms in the alert stream. When the sensors are set to report all suspicious events, they frequently issue alerts for benign background events. On the other hand, decreasing sensor sensitivity reduces their ability to detect real attacks.

To deal with this problem, intrusion detection systems employ heuristic algorithms to correlate alerts from a large pool of heterogeneous sensors. Researchers have suggested that looking at entire intrusion scenarios can help in linking disparate low-level alerts into aggregate super-alerts [25, 91]. We think that alert aggregation can be even more effective with full attack graphs.

Defense

A further benefit of attack graphs is that they can help analyze potential effectiveness of many different security measures offline. In Chapters 8 and 10 we show how both the security policy and the intrusion detection system can be incorporated explicitly in the attack model. The system administrator can then perform several kinds of analysis on such configuration-specific attack graphs to assess the security needs of the network.

Forensics

After a break-in, forensic analysis attempts to identify the actions taken by the attacker and assess the damage. If legal action is desired, analysts seek evidence that a sequence of sensor alerts comprises a coherent attack plan, and is not merely a series of isolated, benign events. This task becomes harder when the intruders obfuscate attack steps by slowing down the pace of the attack and varying specific steps. Researcher have suggested that matching data extracted from IDS logs to a formal reference model based on attack graphs can strengthen the argument that the intruder's actions were malicious [86].

Security Policy Considerations

Experience has shown that organizations are reluctant to change their security policy or patch their machines even in the face of evidence that their network is vulnerable to attack. Sometimes it is impractical to keep all of the machines up to date—some networks contain thousands of hosts, and even with the help of automation applying frequent security patches is a challenge. Organizations sometimes keep a relatively lax security policy because tightening it prevents legitimate users from using the network to do their work. For example, an organization may choose to keep an FTP server open, or give its employees remote access to the network.

Compiling a list of vulnerabilities with a security scanning tool may not be enough to persuade the organization that action is needed. We show that existing vulnerability scanners, such as *Nessus* [30] can be connected with automated attack graph generation software. Such a combination automatically scans a network and generates customized attack graphs. An attack graph showing multiple concrete break-in scenarios that could be executed on the organization's network could be used as persuasive evidence that the organization's security policies are too lax.

Roadmap

Although our study of attack graphs and related models is motivated primarily by applications to network security, in the interest of keeping other research avenues open we make our definitions as broad as possible and make them applicable to any context where active agents attack and defend a complex system. In Chapter 7 we give a broad definition for *attack models* and attack graphs. Chapter 8 narrows the definitions specifically to the domain of network security. In Chapter 9 we consider some post-generation analyses that can be done on attack graphs to help solve real-world questions about network security. Chapter 10 illustrates the definitions and analysis techniques with a small example network. Chapter 11 focuses on the practical aspects of building a usable attack graph tool. We discuss several approaches to collecting the data necessary to build the network model. Finally, we review related work in Chapter 12 and state our conclusions in Chapter 13.

Chapter 7

Attack Graphs

Abstractly, an attack graph is a collection of scenarios showing how a malicious agent can compromise the integrity of a target system. Such graphs are a natural application of the scenario graph formalism defined in Chapter 2. With a suitable model of the system, we can use the techniques developed in Part I to generate attack graphs automatically. In this context, correctness properties specify the negation of the attacker's goal: an execution is correct with respect to the property if the attacker does not achieve his goal for that execution. We call such properties *security properties*.

7.1 Attack Models

Although our primary interest is in multi-stage cyber-attacks against computer networks, we define the attack graph formalism abstractly as a scenario graph for a model where agents attack and defend a complex system. An *attack model* $\mathcal{W} = (S, \tau, \{s_0\}, S, S, D)$ is a Büchi model representing a set of three agents $\mathcal{I} = \{E, D, S\}$. Agent E is an attacker, agent D is a defender, and agent S is the system under attack. The specifics of how each agent is represented in an attack model depend on the type of the system that is under attack; in Chapter 8 we specify the agents more precisely for network attack models.

With each agent $i \in \mathcal{I}$ we associate a set of actions A_i, so that the total set of actions in the model is $A = \bigcup_{i \in \mathcal{I}} A_i$. In general, the attacker's actions move the system "toward" some undesirable (from the system's point of view) state, and the defender's actions attempt to counteract that effect. There is a single root state s_0 representing the initial state of each agent before any action has taken place.

An attack model is a general formalism suitable for modeling a variety of situations. The system under attack can be virtually anything: a computer network under attack by hackers, a city under siege during war, an electrical grid targeted by terrorists, etc. The attacker is an abstract representation of a group of agents that seek to move the system to a state that is inimical to the system's interests. The defender is an agent whose explicit purpose, in opposition to the attacker, is to prevent this occurrence. The system itself is oblivious to the fact that it is under attack. It goes through its normal routine according to its purpose and goals regardless of the actions of the active players.

7.2 Attack Graphs

Formally, an attack graph is a safety scenario graph with an appropriately defined safety property (Section 2.1.8):

Definition 7.1 *Given a security property P, the* attack graph *for an attack model $\mathcal{W} = (S, \tau, \{s_0\}, S, S, D)$ is a safety scenario graph for $\mathcal{W}$ with respect to P.*

Chapter 8 applies this notion of attack graphs to the network security domain and shows how to generate intruder attack graphs automatically using scenario graph generation techniques from Part I.

Chapter 8

Network Attack Graphs

Network attack graphs represent a collection of possible penetration scenarios in a computer network. Each penetration scenario is a sequence of steps taken by the intruder, typically culminating in a particular goal—administrative access on a particular host, access to a database, service disruption, etc. For appropriately constructed network models, attack graphs give a bird's-eye view of every scenario that can lead to a serious security breach. Sections 8.1 and 8.2 show how attack models can be constructed for the network security domain.

8.1 Network Attack Model

A *network attack model* is an attack model where the system S is a computer network, the attacker E is a malicious agent trying to circumvent the network's security, and the defender D represents both the system administrator(s) and security software installed on the network. A state transition in a network attack model corresponds to a single action by the intruder, a defensive action by the system administrator, or a routine network action.

Real networks consist of a large variety of hardware and software pieces, most of which are not involved in cyber attacks. We have chosen six network components relevant to constructing network attack models. The components were chosen to include enough information to represent a wide variety of networks and attack scenarios, yet keep the model reasonably simple and small. The following is a list of the components:

1. *H*, a set of hosts connected to the network
2. *C*, a connectivity relation expressing the network topology and inter-host reachability
3. *T*, a relation expressing trust between hosts
4. *I*, a model of the intruder
5. *A*, a set of individual actions (exploits) that the intruder can use to construct attack scenarios
6. *Ids*, a model of the intrusion detection system

We construct an attack model $\mathcal{W}$ based on these components. The following table defines each agent i's state S_i and action set A_i in terms of the network components:

Agent $i \in \mathcal{I}$	S_i	A_i
E	I	A
D	*Ids*	$\{alarm\}$
S	$H \times C \times T \times S$	$\emptyset$

This construction gives the security administrator an entirely passive "detection" role, embodied in the *alarm* action of the intrusion detection system. For simplicity, regular network activity is omitted entirely.

It remains to make explicit the transition relation of the attack model $\mathcal{W}$. Each transition $(s_1, s_2) \in \tau$ is either an action by the intruder, or an *alarm* action by the system administrator. An *alarm* action happens whenever the intrusion detection system is able to flag an intruder action. An action action $a \in A$ requires that the preconditions of a hold in state s_1 and the postconditions of a hold in s_2.

8.2 Network Components

The rest of this chapter is devoted to explaining the network components in more detail.

Hosts

Hosts are the main hubs of activity on a network. They run services, process network requests, and maintain data. With rare exceptions, every action in an attack scenario will target a host in some way. Typically, an action takes advantage of vulnerable or mis-configured software to gain information or access privileges for the attacker. The main goal in modeling hosts is to capture as much information as possible about components that may contribute to creating an exploitable vulnerability.

A host $h \in H$ is a tuple *(id, svcs, sw, vuls)*, where

- *id* is a unique host identifier (typically, name and network address)
- *svcs* is a list of service name/port number pairs describing each service that is active on the host and the port on which the service is listening
- *sw* is a list of other software operating on the host, including the operating system type and version
- *vuls* is a list of host-specific vulnerable components. This list may include installed software with exploitable security flaws (example: a *setuid* program with a buffer overflow problem), or mis-configured environment settings (example: existing user shell for system-only users, such as *ftp*)

Network Connectivity

Following Ritchey and Ammann [79], connectivity is expressed as a ternary relation $C \subseteq H \times H \times P$, where P is a set of integer port numbers. $C(h_1, h_2, p)$ means that host h_2 is reachable from host h_1 on port p. Note that the connectivity relation incorporates firewalls and other elements that restrict the ability of one host to connect to another. Slightly abusing notation, we say $R(h_1, h_2)$ when there is a network route from h_1 to h_2.

Trust

We model trust as a binary relation $T \subseteq H \times H$, where $T(h_1, h_2)$ indicates that a user may log in from host h_2 to host h_1 without authentication (i.e., host h_1 "trusts" host h_2).

Services

The set of services S is a list of unique service names, one for each service that is present on any host on the network. We distinguish services from other software because network services so often serve as a conduit for exploits. Furthermore, services are tied to the connectivity relation via port numbers, and this information must be included in the model of each host. Every service name in each host's list of services comes from the set S.

Intrusion Detection System

We associate a boolean variable with each action, abstractly representing whether or not the IDS can detect that particular action. Actions are classified as being either *detectable* or *stealthy* with respect to the IDS. If an action is detectable, it will trigger an alarm when executed on a host or network segment monitored by the IDS; if an action is *stealthy*, the IDS does not see it.

We specify the IDS as a function *ids:* $H \times H \times A \to \{d, s, b\}$, where *ids(*$h_1, h_2, a$*)* = *d* if action a is *d*etectable when executed with source host h_1 and target host h_2; *ids(*h_1, h_2, a*)* = *s* if action a is *s*tealthy when executed with source host h_1 and target host h_2; and *ids(*h_1, h_2, a*)* = *b* if action a has *b*oth detectable and stealthy strains, and success in detecting the action depends on which strain is used. When h_1 and h_2 refer to the same host, *ids(*h_1, h_2, a*)* specifies the intrusion detection system component (if any) located on that host. When h_1 and h_2 refer to different hosts, *ids(*h_1, h_2, a*)* specifies the intrusion detection system component (if any) monitoring the network path between h_1 and h_2.

Actions

Each action is a triple (r, h_s, h_t), where $h_s \in H$ is the host from which the action is launched, $h_t \in H$ is the host targeted by the action, and r is the rule that describes how the intruder can change the network or add to his knowledge about it. A specification of an action rule has four components: *intruder preconditions*, *network preconditions*, *intruder effects*, and *network effects*. The *intruder preconditions* component places conditions on the intruder's store of knowledge and the privilege level required to launch the action. The *network preconditions* specifies conditions on target host state, network connectivity, trust, services, and vulnerabilities that must hold before launching the action. Finally, the *intruder* and *network effects* components list the action's effects on the intruder and on the network, respectively.

Intruder

The intruder has a *store of knowledge* about the target network and its users. The intruder's store of knowledge includes host addresses, known vulnerabilities, user passwords, information gathered with port scans, etc. Also associated with the intruder is the function *plvl: Hosts* $\to$ $\{$*none, user, root*$\}$, which gives the level of privilege that the intruder has on each host. For simplicity, we model only three privilege levels. There is a strict total order on the privilege levels: *none* $\leq$ *user* $\leq$ *root*.

For bookkeeping, three state variables specify the most recent action triple used by the intruder:

variable	meaning
action	action rule
source	source host
target	target host

8.2.1 Omitted Complications

Although we do not model actions taken by user services for the sake of simplicity, doing so in the future would let us ask questions about effects of intrusions on service quality. A more complex model could include services provided by the network to its regular users and other routine network traffic. These details would reflect more realistically the interaction between intruder actions and regular network activity at the expense of additional complexity.

Another activity worth modeling explicitly is administrative steps taken either to hinder an attack in progress or to repair the damage after an attack has occurred. The former corresponds to transitioning to states of the model that offer less opportunity for further penetration; the latter means "undoing" some of the damage caused by successful attacks.

Chapter 9

Analysis of Attack Graphs

Attack graphs provide a bird's-eye view of what can go wrong. We can use them to define a rough measure of how exposed the network is to malicious penetration. The candidate metrics for *network exposure* include total graph size and the number of successful attack scenarios. Whichever metric is used, there are many follow-up questions worth exploring. The following is a partial list:

- Find successful attack scenarios undetected by the intrusion detection system or unaffected by contemplated security upgrades
- Determine where to position new IDS components for best coverage, or where to target security upgrades
- Predict effectiveness of various security policies and different software and hardware configurations
- Predict the changes in overall network vulnerability that would occur if new exploits of a certain type became available
- Identify worst case scenarios and prioritize defenses accordingly
- Explore trade-offs between various approaches to defending the network
- Assess the effectiveness of "vigilant" strategies that attempt to defend the system dynamically, while the attack is in progress

Instead of trying to cover all the bases, we will present example analyses that focus on finding the best upgrade strategy when existing defenses are deemed inadequate. For our purposes it is necessary to distinguish between two different defensive strategies—*static* and *dynamic* defense.

Static Defense Analysis

A static defense measure is a one-shot change in the network configuration that reduces network exposure. Static measures include imposing a stricter password policy, applying security patches and upgrades, or tightening firewall rules. A single static measure corresponds to one modification of the attack graph: it reduces the number of scenarios that the intruder can execute, but does not present any additional obstacles to the attacker while the attack is underway. Generally, it is possible to perform analyses of static measures directly on the attack graphs, without referencing the underlying attack model.

When deciding on a static security upgrade, the system administrator may have a number of options and wish to evaluate their relative effectiveness. In Section 9.1 we show how to determine a minimal set of atomic attacks that must be prevented to guarantee that the intruder cannot achieve his goal in some particular attack graph. Appendix A discusses an alternative approach to static analysis that uses probabilistic models.

Dynamic Defense Analysis

Dynamic network defense measures are not yet common in modern network setups, with the notable exception of intrusion detection systems. A dynamic defensive mechanism continually monitors the network and attempts actively to prevent the attacker from succeeding. It is a safe prediction that in the future network software will become more aware of its own state, and active defense will play a correspondingly larger role in network security.

In the presence of dynamic defensive components an attack scenario becomes a game of moves by the attacker and counter-moves by the defender, so it is logical to employ a full game-theoretic treatment in analyzing effectiveness of dynamic defense. Appendix A contains a game-theoretic definition of attack models and attack graphs and discusses analysis of attack graphs based on the game-theoretic definitions.

9.1 Static Defense: Graph-Theoretic Exposure Minimization

Once we have an attack graph generated for a specific network with respect to a given security property, we can attempt to determine the best way to reduce the network exposure to attack. In this section we look at ways to choose the best security upgrade from a set of defensive *measures*, such as deploying additional intrusion detection tools, adding firewalls, upgrading software, deleting user accounts, changing access rights, etc.

We say that a measure *covers* an action if it renders the action ineffective for the intruder. Previous work [52, 80] considers the question of what measures a system administrator should deploy to make the system completely safe. We briefly review the key definitions and results and then extend them.

Assume that we have produced an attack graph G corresponding to the security property

$$P = G(\neg unsafe)$$

Since P is a safety property, we know that G is a safety scenario graph. Let A be the set of actions, and $G = (S, E, s_0, s_f, D)$ be the attack graph, where S is the set of states, $E \subseteq S \times S$ is the set of edges, $s_0 \in S$ is the initial state, $s_f \in S$ is the final state, and $D : E \rightarrow \mathcal{A} \cup \{\epsilon\}$ is a labeling function where $D(e) = a$ if an edge $e = (s \rightarrow s')$ corresponds to an action a, otherwise $D(e) = \epsilon$. Edges labeled with ϵ represent system transitions that do not correspond to an action.

Without loss of generality we assume that there is a single initial state and a single final state. It is easy to convert an attack graph with multiple initial states $s_0^1, \cdots, s_0^j$ and final states $s_f^1, \cdots, s_f^u$ into an attack graph that accepts the same set of scenarios but has a single initial state and a single final state. We simply add a new initial state s_0 and a new final state s_f with ϵ-labeled edges (s_0, s_0^m) $(1 \leq m \leq j)$ and (s_s, s_s^t) $(1 \leq t \leq u)$.

Suppose that we are given a finite set of measures $M = \{m_1, \cdots, m_k\}$ and a function $covers :$ $M \rightarrow 2^A$, where $a \in covers(m_i)$ if adopting measure m_i removes the action a from the intruder's arsenal. We want to find either a subset $M' \subseteq M$ or a subset $A' \subseteq A$ of smallest size, such that adopting the measures in the set M' or installing defenses against all attacks in A' will make the network safe. These two problems are related. To state them formally, we need a few definitions.

A *path* π in $G = (S, E, s_0, s_f, D)$ is sequence of states $s_1, \cdots, s_n$, such that $s_i \in S$ and $(s_i, s_{i+1}) \in E$, where $1 \le i \le n$. A *complete path* starts from the initial state s_0 and ends in the final state s_f. The label of a path $\pi = s_1, \cdots, s_n$ is a subset of the set of actions A. Abusing notation, we will denote the label of path π as $D(\pi)$.

$$\bigcup_{i=1}^{n-1} \{D(s_i, s_{i+1})\} \setminus \{\epsilon\} .$$

$D(\pi)$ represents the set of actions used on the path π. A subset of actions $A' \subseteq A$ is called *realizable* in the attack graph G iff there exists a complete path π in G such that $D(\pi) = A'$. In other words, an intruder can use the set of actions A' to reach the final state from the initial state. We denote the set of all realizable sets in an attack graph G as $Rel(G)$.

Given a state $s \in S$, a set of actions $C(s)$ is *critical* with respect to s if and only if the intruder cannot reach his goal from s when the actions in $C(s)$ are removed from his arsenal. Equivalently, $C(s)$ is critical with respect to s if and only if every path from s to the final state s_f has at least one edge labeled with an action $a \in C(s)$.

A critical set of actions $C(s)$ corresponding to a state s is *minimum* if there is no critical set $C'(s)$ such that $|C'(s)| < |C(s)|$. In general, there can be multiple minimum sets corresponding to a state s. Of course, all minimum critical sets must be of the same size. We say that a critical set of actions corresponding to the initial state s_0 is also a critical set of actions for the attack graph $G = (S, E, s_0, s_f, D)$.

Similarly, given a state $s \in S$, a set of measures $M_c(s)$ is *critical* with respect to s if and only if the intruder cannot reach his goal from s when the actions in $\bigcup_{m \in M_c(s)} covers(m)$ are removed from his arsenal. A critical set of measures $M_c(s)$ corresponding to a state s is *minimum* if there is no critical set $M'_c(s)$ such that $|M'_c(s)| < |M_c(s)|$. We can now state formally two problems:

Definition 9.1 *Minimum Critical Set of Actions (MCSA)*

Given an attack graph $G = (S, E, s_0, s_s, D)$ and a set of actions A, find a minimum critical subset of actions $C(s_0) \subseteq A$ for G.

Definition 9.2 *Minimum Critical Set of Measures (MCSM)*

Given an attack graph $G = (S, E, s_0, s_s, D)$, a set of actions A, and a set of defensive measures $M \subseteq \mathcal{P}(A)$, find a minimum critical subset of measures $M_c(s_0) \subseteq M$.

MCSA is NP-complete [52]. There is a trivial reduction from MCSA to MCSM; given an instance (G, A) of MCSA, we can construct an instance (G, A, M) of MCSM where $M = \{\{a\} \mid a \in A\}$. This reduction shows that MCSM is also NP-complete. To solve these problems efficiently, we employ heuristics.

MCSA and MCSM fall into the broad class of convex optimization problems. They are closely related to two other members of that class, Minimum Set Cover and Minimum Hitting Set.

Definition 9.3 *Minimum Set Cover (MSC)*

Given a collection $\mathcal{J}$ of subsets of a finite set C, find a minimum subcollection $\mathcal{J}' \subseteq \mathcal{J}$ such that $\bigcup_{S_i \in \mathcal{J}'} S_i = C$.

Input:
C, a finite set
$\mathcal{J}$, a collection of subsets of C
Output:
$\mathcal{J}' \subseteq \mathcal{J}$, a set cover of C

Algorithm `GREEDY-SET-COVER`$(C, \mathcal{J})$
$\mathcal{J}' \leftarrow \emptyset$;
$C' \leftarrow C$;
do
Pick $j \in \mathcal{J} \setminus \mathcal{J}'$ that maximizes $|j \cap C'|$;
$\mathcal{J}' \leftarrow \mathcal{J}' \cup \{j\}$;
$C' \leftarrow C' \setminus j$;
until C' is empty;
return $\mathcal{J}'$;

(a) `GREEDY-SET-COVER`

Input:
$G = (S, E, s_0, s_f, D)$, an attack graph
$\mathcal{A}$, an action set
Output:
$A' \subseteq \mathcal{A}$, a critical set of actions in G

Algorithm `GREEDY-CRITICAL-SET`$(G, \mathcal{A})$
$A' \leftarrow \emptyset$;
$G' \leftarrow G$;
do
Pick $a \in \mathcal{A} \setminus A'$ that maximizes $\mu_{G'}(a)$;
$A' \leftarrow A' \cup \{a\}$;
Remove all edges labeled with a from G';
until G' is empty;
return A';

(b) `GREEDY-CRITICAL-SET`

Figure 9.1: Greedy Approximation Algorithms

Definition 9.4 *Minimum Hitting Set (MHS)*

Given a collection $\mathcal{J}$ of subsets of a finite set C, find a minimum subset $H \subseteq C$ that has a nonempty intersection with every member of $\mathcal{J}$.

It has been shown that MSC and MHS are structurally equivalent [6].

Jha, Sheyner, and Wing show MCSA to be structurally equivalent to MHS and, by transitivity, to MSC [52]. An approximate solution to MCSA is found using a variant of the classic greedy MSC approximation algorithm [22], shown in Figure 9.1(a). The corresponding MCSA approximation algorithm `GREEDY-CRITICAL-SET` is specified in Figure 9.1(b). It finds a critical set of actions for an attack graph $G = (S, E, s_0, s_f, D)$. `GREEDY-CRITICAL-SET` makes use of the function $\mu_G : \mathcal{A} \rightarrow \aleph$, where $\mu_G(a)$ is the number of realizable sets in the attack graph G that contain the action a. Formally,

$$\mu_G(a) \equiv |\{A' | a \in A' \text{ and } A' \in Rel(G)\}| \ .$$

By structural equivalence, the approximation ratio R_{gcs} for `GREEDY-CRITICAL-SET` is identical to the ratio for the greedy set cover algorithm:

$$R_{gcs} = \ln n - \ln \ln n + \Theta(1)$$

where $n = |Rel(G)|$ [84].

To solve MCSM, we use measures instead of individual actions in another variant of the same greedy algorithm. To justify this approach, we show via a sequence of reductions that MCSM is structurally equivalent to MSC. Throughout the rest of this section, structure-preserving reduction from problem A to problem B is denoted $A \leq_S B$.

Definition 9.5 *Minimum Hitting Cover (MHC)*

Given two collections $\mathcal{J}, \mathcal{K}$ of subsets of a finite set C, find a minimum subcollection $\mathcal{K} \subseteq \mathcal{K}'$ such that $\forall j \in \mathcal{J}$, we have

Input:
  $G = (S, E, s_0, s_f, D)$, an attack graph
  A, an action set
  $M \subseteq \mathcal{P}(A)$, a set of security measures
Output:
  $M' \subseteq M$, a critical set of measures in G

Algorithm `GREEDY-MEASURE-SET`(G, M)
  $M' \leftarrow \emptyset$;
  $G' \leftarrow G$;
  do
    Pick $m \in M \setminus M'$ that maximizes $\nu_{G'}(a)$;
    $M' \leftarrow M' \cup \{m\}$;
    Remove all edges labeled
      with any attack in m from G';
  until G' is empty;
  return M';

(b) `GREEDY-MEASURE-SET`

Figure 9.2: Greedy Measure Set

$$\bigcup_{k \in \mathcal{K}} k \quad \cap \quad j \neq \emptyset.$$

Definition 9.6 *Minimum Powerset Cover (MPC)*

Let C be a finite set and $\Upsilon \subseteq \mathcal{P}(\mathcal{P}(C))$ a collection of sets of subsets of C. Find a minimum subcollection $\Upsilon' \subseteq \Upsilon$ such that

$$\bigcup_{\mathcal{K}_i \in \Upsilon'} \cup_{S_j \in \mathcal{K}_i} S_j = C.$$

Below we prove that MCSM $\leq_S$ MHC $\leq_S$ MPC $\leq_S$ MSC. The relationship between MCSM and MSC lets us use a variant of `GREEDY-SET-COVER` for approximating MCSM. The algorithm `GREEDY-MEASURE-SET` is shown in Figure 9.2. It finds a set of measures that cuts attack graph $G = (S, E, s_0, s_f, D)$. The algorithm uses the function $\nu_G : M \to \aleph$, where $\nu_G(m)$ is the number of realizable sets in the attack graph G that contain some attack from measure m:

$$\nu_G(m) \equiv |\{A' | m \cap A' \neq 0 \text{ and } A' \in \mathit{Rel}(G)\}| \,.$$

By structural equivalence, the approximation ratio R_{gms} for `GREEDY-MEASURE-SET` is identical to the ratio for the greedy set cover algorithm:

$$R_{gms} = \ln n - \ln \ln n + \Theta(1)$$

where $n = |\mathit{Rel}(G)|$ [84]. It remains to prove the reductions MCSM $\leq_S$ MHC $\leq_S$ MPC $\leq_S$ MSC.

Theorem 9.1 *MCSM $\leq_S$ MHC.*

Proof:

Let (G, A, M) be an instance of Minimum Critical Set of Measures. The goal of MCSM is to select a subcollection of measures that together hit every realizable set of attacks in G at least once. We construct an instance $(C, \mathcal{J}, \mathcal{K})$ of Minimum Hitting Cover with

$$C = A$$

$$\mathcal{J} = Rel(G)$$

$$\mathcal{K} = M$$

Every feasible solution $\mathcal{K}' \subseteq \mathcal{K}$ of Minimum Hitting Cover will ensure that every set in $\mathcal{J}$ is hit at least once. Equivalently, in the MCSM instance the same solution will ensure that every set in $Rel(G)$ is hit by at least one measure, as desired.

$\square$

Theorem 9.2 *MHC* $\leq_S$ *MPC.*

Proof:

MHC and MPC are duals of each other, and it is possible to turn one into the other with a simple substitution. Given an instance $(C, \mathcal{J}, \mathcal{K})$ of MHC, let f be a function that assigns a name from the name set *Nm* $= \{j_1, \ldots, j_n\}$ $(n = |\mathcal{J}|)$ to every set in the collection $\mathcal{J}$, so that $\mathcal{J} = \{f^{-1}(j_1), \ldots, f^{-1}(j_n)\}$.

We construct an instance of MPC by substituting for each element $c \in C$ the subset of names *Nm*' $\in$ *Nm* such that each element named in *Nm* contains c. That is, $\forall j \in$ *Nm*', we have $c \in f^{-1}(j)$.

This substitution changes $\mathcal{K}$ into a set Υ of sets of names from *Nm*, and the task of selecting a minimum subcollection of $\mathcal{K}$ that hits every member of collection $\mathcal{J}$ turns into the task of covering the set of names *Nm* with elements of Υ. In other words, (*Nm*, Υ) is an instance of MPC. A solution to the MPC (*Nm*, Υ) also gives the (dual) solution to the corresponding MHC $(C, \mathcal{J}, \mathcal{K})$.

$\square$

Theorem 9.3 *MPC* $\leq_S$ *MSC.*

Proof:

This reduction is simple. Let (C, Υ) be an instance of Minimum Powerset Cover. The goal is to pick a number of items from Υ that together cover C. Each item $\upsilon \in \Upsilon$ is a set of subsets of C. But υ covers exactly the same portion of C as its "flattened" version, $\bigcup_{j \in \upsilon} j$. So we can construct an instance of Minimum Set Cover $(C', \mathcal{J})$ by carrying C unchanged and flattening every item in Υ:

$$C' = C$$

$$\mathcal{J} = \{ \bigcup_{j \in \upsilon} j \mid \upsilon \in \Upsilon \}$$

A solution of MSC for $(C', \mathcal{J})$ is thus also an MPC solution for (C, Υ).

$\square$

Jha, Sheyner, and Wing have suggested a different, two-step approach for finding approximate solutions to MCSM [52]. In the first step, they find a critical set of actions A' using the approximation algorithm for MCSA. In the second step, the MSC approximation algorithm in Figure 9.1(a) takes the set of actions A' computed in the first step and covers them with measures. This approach does not offer any guarantee that the resulting set of measures is a good approximation to the MCSM. While the set of actions A' is a good approximation to the minimum critical set of actions, it may not be the right set of actions to cover with the available measures. In fact, it is possible that the entire set of available measures cannot cover A', but can nevertheless cover some other critical set of actions. Further, even when some subset of measures $M_c \subseteq M$ *can* cover A', it is possible that a smaller subset of measures $M'_c \subseteq M$ can cover some other, perhaps much larger, critical set of actions.

Chapter 10

Example Network

Figure 10.1 shows an example network. There are two target hosts, `Windows` and `Linux`, on an internal company network, and a Web server on an isolated "demilitarized zone" (DMZ) network. One firewall separates the internal network from the DMZ and another firewall separates the DMZ from the rest of the Internet. An intrusion detection system (IDS) watches the network traffic between the internal network and the outside world.

The `Linux` host on the internal network is running several services—Linux "I Seek You" (*LICQ*) chat software, *Squid* web proxy, and a *Database*. The *LICQ* client lets Linux users exchange text messages over the Internet. The *Squid* web proxy is a caching server. It stores requested Internet objects on a system closer to the requesting site than to the source. Web browsers can then use the local *Squid* cache as a proxy, reducing access time as well as bandwidth consumption. The host inside the DMZ is running Microsoft's Internet Information Services (IIS) on a Windows platform.

The intruder launches his attack starting from a single computer, which lies on the outside network. To be concrete, let us assume that his eventual goal is to disrupt the functioning of the database. To achieve this goal, the intruder needs root access on the database host `Linux`. The five actions at his disposal are summarized in the following table:

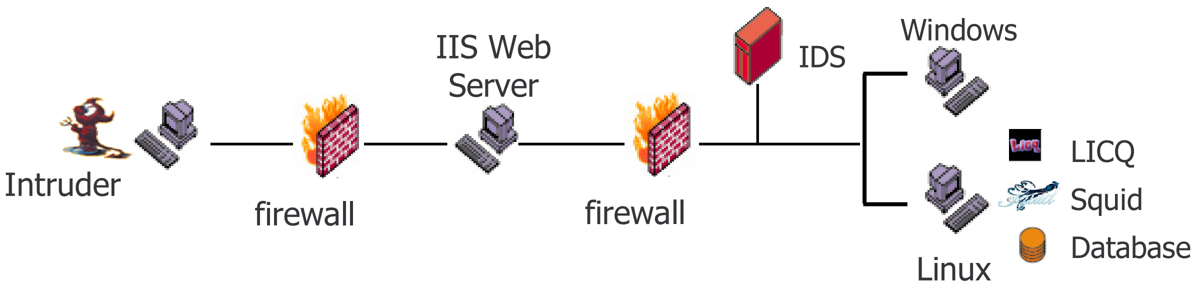


Figure 10.1: Example Network

Action	Effect	Example CVE ID
IIS buffer overflow	remotely get root	CAN-2002-0364
Squid port scan	port scan	CVE-2001-1030
LICQ gain user	gain user privileges remotely	CVE-2001-0439
scripting exploit	gain user privileges remotely	CAN-2002-0193
local buffer overflow	locally get root	CVE-2002-0004

Each of the five actions corresponds to a real-world vulnerability and has an entry in the Common Vulnerabilities and Exposures (CVE) database. CVE [96] is a standard list of names for vulnerabilities and other information security exposures. A CVE identifier is an eight-digit string prefixed with the letters “CVE” (for accepted vulnerabilities) or “CAN” (for candidate vulnerabilities).

The IIS buffer overflow action exploits a buffer overflow vulnerability in the Microsoft IIS Web Server to gain administrative privileges remotely.

The *Squid* action lets the attacker scan network ports on machines that would otherwise be inaccessible to him, taking advantage of a mis-configured access control list in the *Squid* web proxy.

The *LICQ* action exploits a problem in the URL parsing function of the *LICQ* software for Unix-flavor systems. An attacker can send a specially-crafted URL to the *LICQ* client to execute arbitrary commands on the client’s computer, with the same access privileges as the user of the *LICQ* client.

The scripting action lets the intruder gain user privileges on Windows machines. Microsoft Internet Explorer 5.01 and 6.0 allow remote attackers to execute arbitrary code via malformed Content-Disposition and Content-Type header fields that cause the application for the spoofed file type to pass the file back to the operating system for handling rather than raise an error message. This vulnerability may also be exploited through HTML formatted email. The action requires some social engineering to entice a user to visit a specially-formatted Web page. However, the action can work against firewalled networks, since it requires only that internal users be able to browse the Web through the firewall.

Finally, the local buffer overflow action can exploit a multitude of existing vulnerabilities to let a user without administrative privileges gain them illegitimately. For the CVE number referenced in the table, the action exploits buffer overflow in the *at* program. The *at* program is a Linux utility for queuing shell commands for later execution.

Some of the actions that we model have multiple instantiations in the CVE database. For example, the local buffer overflow action exploits a common coding error that occurs in many Linux programs. Each program vulnerable to local buffer overflow has a separate CVE entry, and all such entries correspond to the same action rule. The table lists only one example CVE identifier for each rule.

10.1 Example Network

Target Network

We specify the state of the network to include services running on each host, existing vulnerabilities, and connectivity between hosts. There are five boolean variables for each host, specifying whether any of the three services are running and whether either of two other vulnerabilities are present on that host:

variable	**meaning**
w3svc$_h$	IIS web service running on host h
squid$_h$	*Squid* proxy running on host h
licq$_h$	*LICQ* running on host h
scripting$_h$	HTML scripting is enabled on host h
vul-at$_h$	*at* executable vulnerable to overflow on host h

The model of the target network includes connectivity information among the four hosts. The initial value of the connectivity relation R is shown the following table. An entry in the table corresponds to a pair of hosts (h_1, h_2). IIS and *Squid* listen on port 80 and the *LICQ* client listens on port 5190, and the connectivity relation specifies which of these services can be reached remotely from other hosts. Each entry consists of three boolean values. The first value is 'y' if h_1 and h_2 are connected by a physical link, the second value is 'y' if h_1 can connect to h_2 on port 80, and the third value is 'y' if h_1 can connect to h_2 on port 5190.

Host	`Intruder`	`IIS Web Server`	`Windows`	`Linux`
`Intruder`	y,y,y	y,y,n	n,n,n	n,n,n
`IIS Web Server`	y,n,n	y,y,y	y,y,y	y,y,y
`Windows`	n,n,n	y,y,n	y,y,y	y,y,y
`Linux`	n,n,n	y,y,n	y,y,y	y,y,y

We use the connectivity relation to reflect the settings of the firewall as well as the existence of physical links. In the example, the intruder machine initially can reach only the Web server on port 80 due to a strict security policy on the external firewall. The internal firewall is initially used to restrict internal user activity by disallowing most outgoing connections. An important exception is that internal users are permitted to contact the Web server on port 80.

In this example the connectivity relation stays unchanged throughout an attack. In general, the connectivity relation can change as a result of intruder actions. For example, an action may enable the intruder to compromise a firewall host and relax the firewall rules.

Intrusion Detection System

A single network-based intrusion detection system protects the internal network. The paths between hosts `Intruder` and `Web` and between `Windows` and `Linux` are not monitored; the IDS can see the traffic between any other pair of hosts. There are no host-based intrusion detection components. The IDS always detects the *LICQ* action, but cannot see any of the other actions. The IDS is represented with a two-dimensional array of bits, shown in the following table. An entry in the table corresponds to a pair of hosts (h_1, h_2). The value is 'y' if the path between h_1 and h_2 is monitored by the IDS, and 'n' otherwise.

Host	`Intruder`	`IIS Web Server`	`Windows`	`Linux`
`Intruder`	n	n	y	y
`IIS Web Server`	n	n	y	y
`Windows`	y	y	n	n
`Linux`	y	y	n	n

The Intruder

The intruder's store of knowledge consists of a single boolean variable 'scan'. The variable indicates whether the intruder has successfully performed a port scan on the target network. For simplicity, we do not keep track of specific information gathered by the scan. It would not be difficult to do so, at the cost of increasing the size of the state space.

Initially, the intruder has root access on his own machine `Intruder`, but no access to the other hosts. The 'scan' variable is set to *false*.

Action Rules

There are five action rules corresponding to the five actions in the intruder's arsenal. Throughout the description, S is used to designate the source host and T the target host. $R(S, T, p)$ says that host T is reachable from host S on port p. The abbreviation *plvl*(X) refers to the intruder's current privilege level on host X.

Recall that a specification of an action rule has four components: *intruder preconditions*, *network preconditions*, *intruder effects*, and *network effects*. The *intruder preconditions* component places conditions on the intruder's store of knowledge and the privilege level required to launch the action. The *network preconditions* component specifies conditions on target host state, network connectivity, trust, services, and vulnerabilities that must hold before launching the action. Finally, the *intruder* and *network effects* components list the effects of the action on the intruder's state and on the network, respectively.

IIS Buffer Overflow

This remote-to-root action immediately gives a remote user a root shell on the target machine.

action IIS-buffer-overflow **is**
  intruder preconditions
    plvl$(S) \geq$ user — *User-level privileges on host S*
    plvl$(T) <$ root — *No root-level privileges on host T*
  network preconditions
    w3svc_T — *Host T is running vulnerable IIS server*
    $R(S, T, 80)$ — *Host T is reachable from S on port 80*
  intruder effects
    plvl$(T) :=$ root — *Root-level privileges on host T*
  network effects
    $\neg\text{w3svc}_T$ — *Host T is not running IIS*
end

Squid Port Scan

The *Squid* port scan action uses a mis-configured *Squid* web proxy to conduct a port scan of neighboring machines and report the results to the intruder.

action squid-port-scan **is**
  intruder preconditions

plvl$(S) =$ user	*User-level privileges on host S*
$\neg$scan	*We have not yet performed a port scan*
network preconditions	
squid$_T$	*Host T is running vulnerable* Squid *proxy*
$R(S, T, 80)$	*Host T is reachable from S on port 80*
intruder effects	
scan	*We have performed a port scan on the network*
network effects	
$\oslash$	*No changes to the network component*
end	

LICQ Remote to User

This remote-to-user action immediately gives a remote user a user shell on the target machine. The action rule assumes that a port scan has been performed previously, modeling the fact that such actions typically become apparent to the intruder only after a scan reveals the possibility of exploiting software listening on lesser-known ports.

action LICQ-remote-to-user **is**	
intruder preconditions	
plvl$(S) \geq$ user	*User-level privileges on host S*
plvl$(T) =$ none	*No user-level privileges on host T*
scan	*We have performed a port scan on the network*
network preconditions	
licq$_T$	*Host T is running vulnerable* LICQ *software*
$R(S, T, 5190)$	*Host T is reachable from S on port 5190*
intruder effects	
plvl$(T) :=$ user	*User-level privileges on host T*
network effects	
$\oslash$	*No changes to the network component*
end	

Scripting Action

This remote-to-user action immediately gives a remote user a user shell on the target machine. The action rule does not model the social engineering required to get a user to download a specially-created Web page.

action client-scripting **is**	
intruder preconditions	
plvl$(S) \geq$ user	*User-level privileges on host S*
plvl$(T) =$ none	*No user-level privileges on host T*
network preconditions	
scripting$_T$	*HTML scripting is enabled on host T*

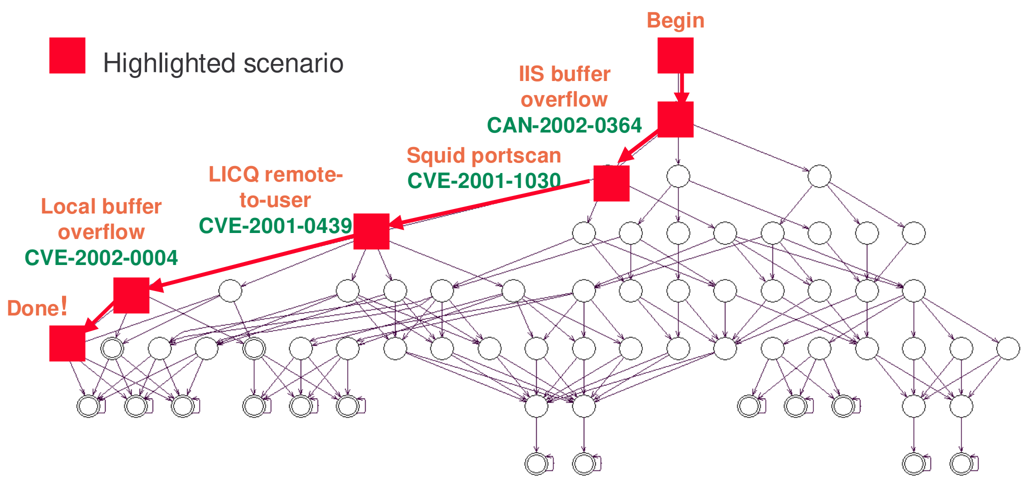


Figure 10.2: Example Attack Graph

$R(T, S, 80)$	*Host S is reachable from T on port 80*
intruder effects	
$plvl(T)$:= user	*User-level privileges on host T*
network effects	
$\oslash$	*No changes to the network component*
end	

Local Buffer Overflow

If the intruder has acquired a user shell on the target machine, this action exploits a buffer overflow vulnerability on a *setuid root* file (in this case, the *at* executable) to gain root access.

action local-setuid-buffer-overflow **is**	
intruder preconditions	
$plvl(T) =$ user	*User-level privileges on host T*
network preconditions	
vul-at$_T$	*There is a vulnerable* at *executable*
intruder effects	
$plvl(T)$:= root	*Root-level privileges on host T*
network effects	
$\oslash$	*No changes to the network component*
end	

10.2 Sample Attack Graphs

We have implemented an attack graph toolkit using scenario graph algorithms in Part I and network attack models defined in this chapter. We describe the toolkit in Chapter 11. Figure 10.2 shows a screenshot of the attack graph generated with the toolkit for the security property

$$\mathbf{G}(intruder.privilege[lin] < root)$$

which states that the intruder will never attain root privileges on the `Linux` host. In Figure 10.2, a sample attack scenario is highlighted with solid square nodes, with each attack step identified by name and CVE number. Since the external firewall restricts most network connections from the outside, the intruder has no choice with respect to the initial step—it must be a buffer overflow action on the IIS Web server. Once the intruder has access to the Web server machine, his options expand. The highlighted scenario is the shortest route to success. The intruder uses the Web server machine to launch a port scan via the vulnerable *Squid* proxy running on the Linux host. The scan discovers that it is possible to obtain user privileges on the Linux host with the *LICQ* exploit. After that, a simple local buffer overflow gives the intruder administrative control over the `Linux` machine. The last transition in the action path is a bookkeeping step, signifying the intruder's success.

Any information explicitly represented in the model is available for inspection and analysis in the attack graph. For instance, with a few clicks we are able to highlight portions of the graph "covered" by the intrusion detection system. Figure 10.3 shades the nodes where the IDS alarm has been sounded. These nodes lie on paths that use the *LICQ* action along a network path monitored by the IDS. It is clear that while a substantial portion of the graph is covered by the IDS, the intruder can escape detection and still succeed by taking one of the paths on the right side of the graph. Once such attack scenario is highlighted with square nodes in Figure 10.3. It is very similar to the attack scenario discussed in the previous paragraph, except that the *LICQ* action is launched from the internal `Windows` machine, where the intrusion detection system does not see it. To prepare for launching the *LICQ* action from the `Windows` machine, an additional step is needed to obtain user privileges in the machine. For that, the intruder uses the client scripting exploit on the Windows host immediately after taking over the Web machine.

10.2.1 Sample Attack Graph Analysis

To demonstrate some of the analysis techniques discussed in Chapter 9, we expanded the example from Section 10.1 with an extra host `User` on the external network and several new actions. An authorized user W of the internal network owns the new host and uses it as a terminal to work remotely on the internal `Windows` host. The new actions permit the intruder to take over the host `User`, sniff user W's login credentials, and log in to the internal `Windows` host using the stolen credentials. We omit the details of the new actions, as they are not essential to understanding the examples. Figure 10.4(a) shows the full graph for the modified example. The graph is significantly larger, reflecting the expanded number of choices available to the intruder.

Single Action Removal

A simple kind of analysis determines the impact of removing one action from the intruder's arsenal. Recall from Chapter 8 that each action is a triple (r, h_s, h_t), where $h_s \in H$ is the host from which the attack is launched, $h_t \in H$ is the host targeted by the attack, and r is an action rule. The user

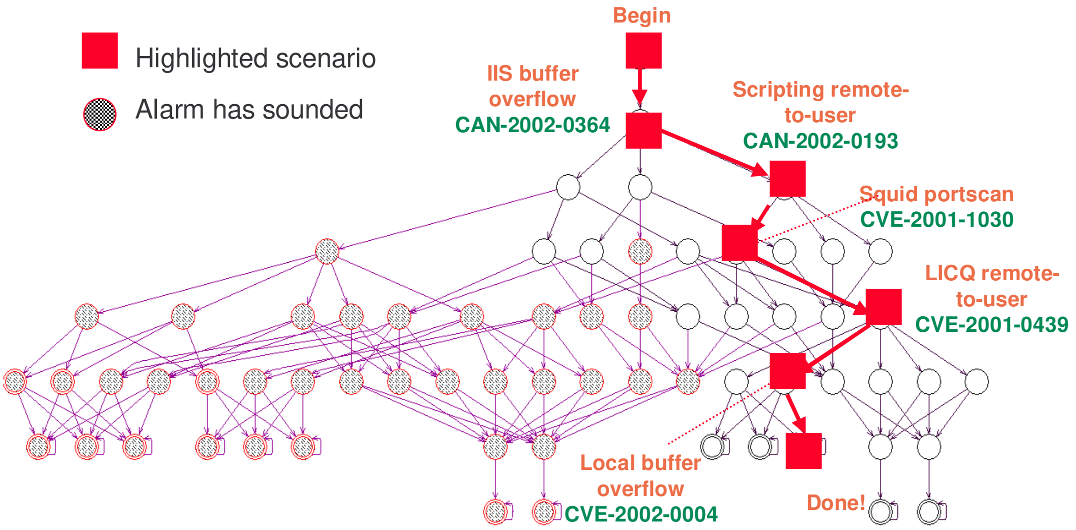


Figure 10.3: Alternative Attack Scenario Avoiding the IDS

specifies a set A_{rem} of action triples to be removed from the attack graph. The toolkit removes the transitions corresponding to each triple in the set A_{rem} and then reconstructs the attack graph using the regular explicit state algorithm `GenerateScenarioAutomaton`.

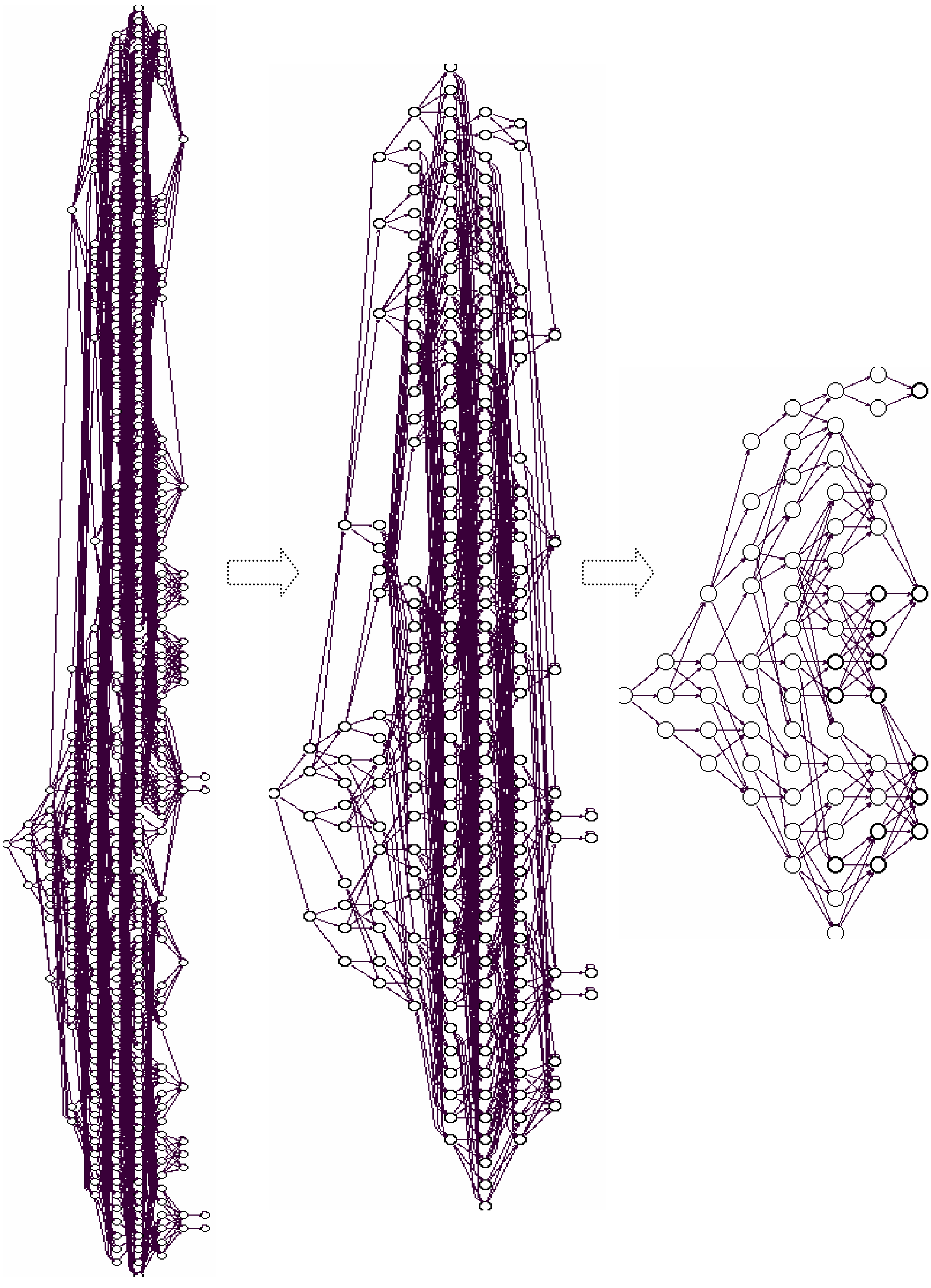
As demonstrated in Figure 10.4, this procedure can be repeated several times, reducing the size of the attack graph at each step. The full graph in Figure 10.4(a) has 362 states. Removing one of two ways the intruder can sniff user W's login credentials produces the graph in Figure 10.4(b), with 213 states. Removing one of the local buffer overflow actions produces the graph in Figure 10.4(c), with 66 states. At each step, the user is able to judge visually the impact of removing a single action from the intruder's arsenal.

Critical Attack Sets

Recall from Section 9.1 that once an attack graph is generated, the `GREEDY-CRITICAL-SET` algorithm can find an approximately-optimal set of actions that will completely disconnect the initial and final states of the graph. Similarly, the `GREEDY-MEASURE-SET` algorithm can find an approximately-optimal set of security measures that accomplish the same goal. With a single click, the user can invoke both of these exposure minimization algorithms.

The effect of `GREEDY-CRITICAL-SET` on the modified example attack graph is shown in Figure 10.5(a). The algorithm finds a critical action set of size 1, containing the port scan action exploiting the *Squid* web proxy. The graph nodes and edges corresponding to actions in the critical set computed by the algorithm are highlighted in the toolkit by shading the relevant nodes. The shaded nodes are seen clearly when we zoom in to inspect a part of the graph on a larger scale (Figure 10.5(b)).

Since the computed action set is always critical, removing every action triple in the set from the intruder's arsenal is guaranteed to result in an empty attack graph. In the example, we might patch the `Linux` machine with a new version of the *Squid* proxy, thereby removing every action triple that uses the *Squid* port scan rule on the `Linux` machine from the intruder's arsenal.



(a) (b) (c)

Figure 10.4: Reducing Action Arsenal

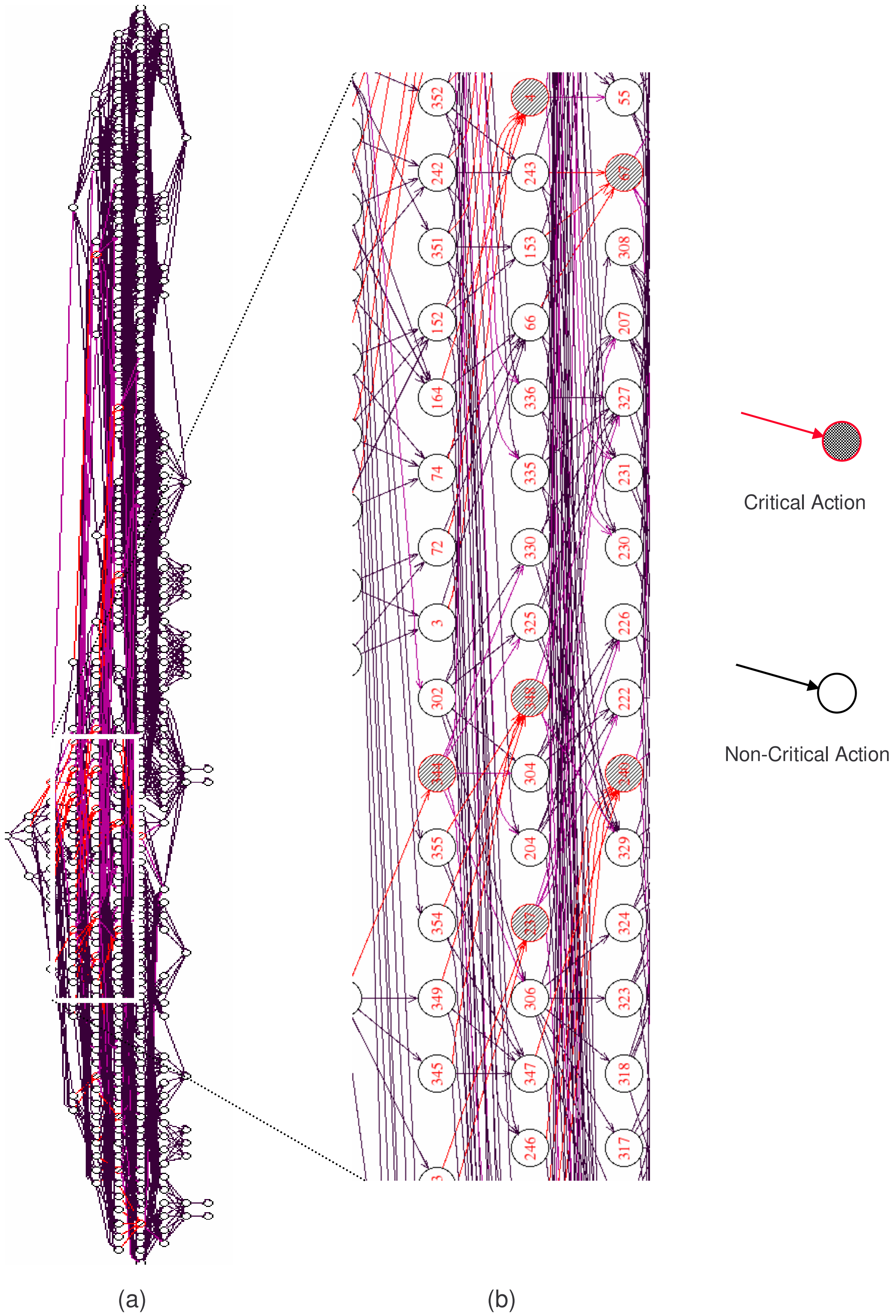


Figure 10.5: Finding Critical Action Sets

Chapter 11

Attack Graph Toolkit

We have implemented a toolkit for generating and exploring attack graphs, using scenario graph algorithms in Part I and network attack models defined Chapter 8. In this chapter we describe the toolkit and show several ways to integrate it with external data sources that supply information necessary to build a network attack model. Specifically, it is necessary to know the topology of the target network, configuration of the network hosts, and vulnerabilities present on the network. In addition, we require access to a database of attack rules to build the transition relation of the attack model. We could expect the user to specify all of the necessary information manually, but such a task is tedious, error-prone, and unrealistic for networks of more than a few nodes.

We recommend deploying the attack graph toolkit in conjunction with information-gathering systems that supply some of the data automatically. We integrated the attack graph generator with two such systems, MITRE Corp's Outpost and Lockheed Martin's ANGI. We report on our experience with Outpost and ANGI in Sections 11.4 and 11.5.

11.1 Toolkit Architecture

Figure 11.1 shows the architecture of the attack graph toolkit. There are three main pieces: a *network model builder*, a *scenario graph generator*, and a *graphical user interface* (GUI). The network model builder takes as input information about network topology, configuration data for each networked host, and a library of attack rules. It constructs a finite model of the network suitable for automated analysis. The model is augmented with a security specification, which spells out the security requirements against which the attack graph is to be built. The model and the security specification then go to the second piece, the scenario graph generator.

The scenario graph generator is a general-purpose tool based on the algorithms developed in Part I (Section 3.3). It takes any finite model and correctness specification and produces a graph composed of possible executions of the model that violate the correctness specification. The model builder constructs the input to the graph generator so that the output will be the desired attack graph. The graphical user interface lets the user display and examine the graph.

11.2 The Model Builder

Recall from Section 8.1 that a network attack model consists of six primary components:

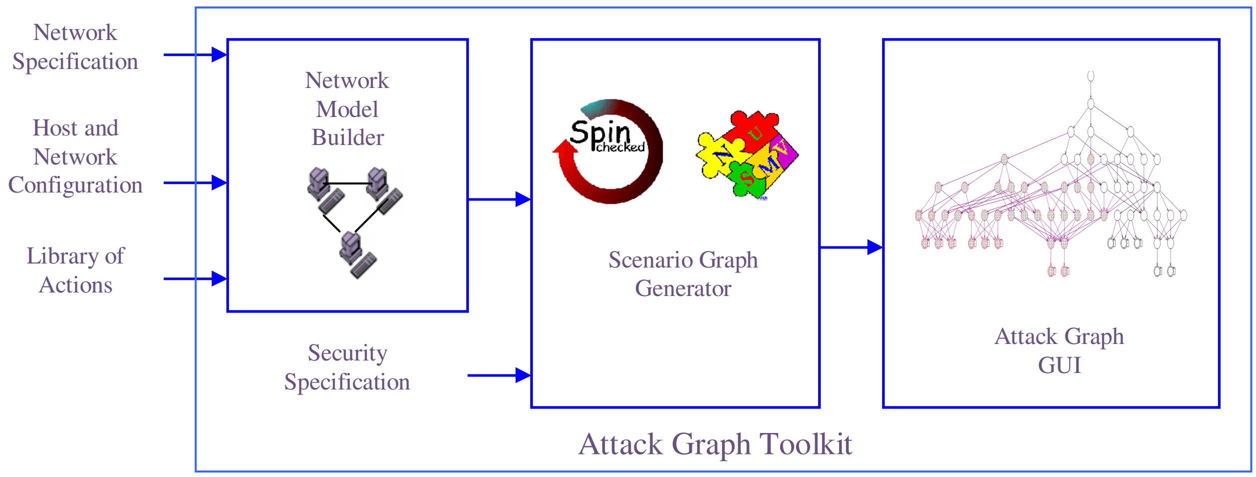


Figure 11.1: Toolkit Architecture

1. *H*, a set of hosts connected to the network
2. *C*, a connectivity relation expressing the network topology and inter-host reachability
3. *T*, a relation expressing trust between hosts
4. *I*, a model of the intruder
5. *A*, a set of individual attack actions
6. *Ids*, a model of the intrusion detection system

To construct each of the six components, the model builder needs to collect the following pieces of information. For the entire network, we need:

1. The set of hosts H
2. The network topology and firewall rules, which together induce the connectivity relation C
3. The initial state of the trust relation T: which hosts are trusted by other hosts prior to any intruder action

Several pieces of data are required for each host h in the set H:

4. A unique host identifier (usually name and network address)
5. Operating system vendor and version
6. Active network services with port numbers
7. Common Vulnerabilities and Exposures IDs of all vulnerabilities present on h
8. User-specific configuration parameters

Finally, for each CVE vulnerability present on at least one host in the set H, we need:

9. An attack rule with preconditions and effects

We designed an XML-based format covering all of the information that the model builder requires. The XML format lets the user specify each piece of information manually or indicate that the data can be gathered automatically from an external source. A typical description of a host in XML is as follows:

```
1  <host id="typical-machine" ip="192.168.0.1">
2
3    <services>
4      <ftp port="21"/>
5      <W3SVC port="80"/>
6    </services>
7
8    <connectivity>
9      <remote id="machine1" <ftp/> <W3SVC/> </remote>
10     <remote id="machine2"> <sshd/> <W3SVC/> </remote>
11     <remote id="machine3"> <sshd/> </remote>
12   </connectivity>
13
14   <cve>
15     <CAN-2002-0364/>
16     <CAN-2002-0147/>
17   </cve>
18
19 </host>
```

The example description provides the host name and network identification (line 1), a list of active services with port numbers (lines 3-6), the part of the connectivity relation that involves the host (lines 8-12), and names of CVE and CVE-candidate (CAN) vulnerabilities known to be present on the host (lines 14-17). Connectivity is specified as a list of services that the host can reach on each remote machine. Lines 9-11 each specify one remote machine; e.g., `typical-machine` can reach `machine1` on ports assigned to the *ftp* and *W3SVC* (IIS Web Server) services.

It is unrealistic to expect the user to collect and specify all of the data by hand. In Sections 11.3-11.5 we discuss three external data sources that supply some of the information automatically: the Nessus vulnerability scanner, MITRE Corp.'s Outpost, and Lockheed Martin's ANGI. Whenever the model builder can get a specific piece of information from one of these sources, a special tag is placed in the XML file. If Nessus, Outpost and ANGI are all available at the same time as sources of information, the above host description may look as follows:

```
<host id="typical-machine" ip="|Outpost|">

  <services source="|Outpost|"/>
  <connectivity source="|ANGI|"/>
  <cve source="|Nessus|"/>

</host>
```

The model builder gets the host network address and the list of running services from Outpost, connectivity information from ANGI, and a list of existing vulnerabilities from Nessus. Once all of the relevant information is gathered, the model builder creates a finite model and encodes it in the input language of the scenario graph generator. The scenario graph generator then builds the attack graph.

11.3 Attack Graphs with Nessus

A savvy attacker might use one of the many widely available vulnerability scanners [23] to discover facts about the network and construct an attack scenario manually. Similarly, an attack graph generator can use a scanner to construct such scenarios automatically. Our attack graph toolkit works with the freeware vulnerability scanner Nessus [30] to gather information about reachable hosts, services running on those hosts, and any known exploitable vulnerabilities that can be detected remotely.

The scanner has no internal knowledge of the target hosts, and will usually discover only part of the information necessary to construct a graph that includes every possible attack scenario. Using only an external vulnerability scanner to gather information can lead the system administrator to miss important attack scenarios.

Nevertheless, the administrator can run vulnerability scanners against his own network to find out what a real attacker would discover. In the future, sophisticated intruders are likely to use attack graph generators to help them devise attack scenarios. As a part of network security strategy, we recommend running a vulnerability scanner in conjunction with an attack graph generator periodically to discover avenues of attack that are most likely to be exploited in practice.

11.4 Attack Graphs with MITRE Outpost

MITRE Corporation's Outpost is a system for collecting, organizing, and maintaining security-related information on computer networks. It is a suite of inter-related security applications that share a common data model and a common data collection infrastructure. The goal of Outpost is to provide a flexible and open environment for network and system administrators to monitor, control, and protect computer systems.

At the center of the Outpost System is a data collection/probe execution engine that gathers specific configuration information from all of the systems within a network. The collected data is stored in a central database for analysis by the Outpost applications. Outpost collects data about individual hosts only, so it cannot provide information about network topology or attack rules. Since Outpost stores all of the data in a network-accessible SQL database, we retrieve the data directly from the database, without talking to the Outpost server, as shown in Figure 11.2.

Currently Outpost works with SQL databases supported by Microsoft and Oracle. Both of these packages use a proprietary Application Programming Interface. The model builder includes an interface to each database, as well as a generic module that uses the Open DataBase Connectivity interface (ODBC) and works with any database that supports ODBC. Furthermore, it is easy to add a capability to interface with other types of databases.

An Outpost-populated database contains a list of client hosts monitored by the Outpost server. For the model builder, the Outpost server can provide most of the required information about each individual host h, including:

1. A unique host identifier (usually name and network address)

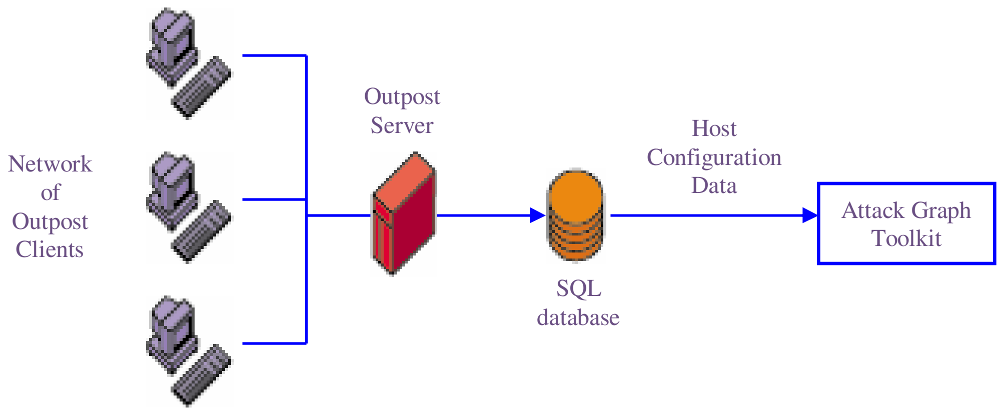


Figure 11.2: Outpost Architecture

2. Operating system vendor and version
3. Active network services with port numbers
4. Common Vulnerabilities and Exposures IDs of all vulnerabilities present on h
5. User-specific configuration parameters (e.g., is Javascript enabled for the user's email client?)

Outpost's lists of CVE vulnerabilities are usually incomplete, and it does not keep track of some of the user-specific configuration parameters required by the attack graph toolkit. Until these deficiencies are fixed, the user must provide the missing information manually.

In the future, the Outpost server will inform the attack graph toolkit whenever changes are made to the database. The tighter integration with Outpost will enable attack graph toolkit to re-generate attack graphs automatically every time something changes in the network configuration.

11.5 Attack Graphs with Lockheed's ANGI

Lockheed Martin Advanced Technology Laboratory's (ATL) Next Generation Infrastructure (ANGI) IR&D project is building systems that can be deployed in dynamic, distributed, and open network environments. ANGI collects local sensor information continuously on each network host. The sensor data is shared among the hosts, providing dynamic awareness of the network status to each host. ANGI sensors gather information about host addresses, host programs and services, and network topology. In addition, ANGI supports vulnerability assessment sensors for threat analysis.

Two distinguishing features of ANGI are the ability to discover network topology changes dynamically and focus on technologies for pro-active, automated repair of network problems. ANGI is capable of providing the attack graph model builder with network topology information, which is not available in Outpost and is not gathered by Nessus.

We tested our attack graph toolkit integrated with ANGI on a testbed of five hosts with combinations of the five CVE vulnerabilities specified for the example model in Chapter 10 (p. 72), and one adversary host. Figure 11.3 is a screenshot of the testbed network schematic. The intruder resides

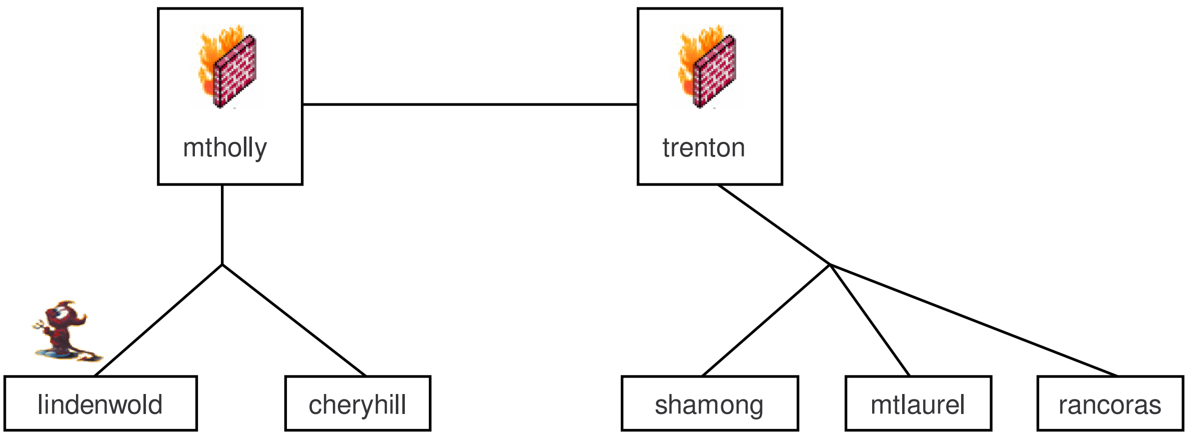


Figure 11.3: ANGI Network

on the host `lindenwold`. Hosts `trenton` and `mtholly` run firewalls, which are initially disabled. We assume that the target of the intruder is the host `shamong`, which contains some critical resource.

ANGI combines information about each host with data from firewall configuration files into a single XML document. To convert firewall rules into a reachability relation C accepted by the attack graph toolkit, ANGI uses a package developed at MITRE Corp. that computes network reachability from packet filter data [78]. The XML file specifies explicitly five attack rules corresponding to the CVE vulnerabilities present on the hosts. ANGI then calls the model builder with the XML document and a security property as inputs. The security property specifies a guarantee of protection for the critical resource host `shamong`:

$$\mathbf{G}(intruder.privilege[shamong] < root)$$

The attack graph generator finds several potential attack scenarios. Figure 11.4 shows the attack graph as it is displayed by the graphical user interface. The graph consists of 19 nodes with 28 edges.

Exploring the attack graph reveals that several successful attack scenarios exploit the *LICQ* vulnerability on the host `shamong`. One such attack scenario is highlighted in Figure 11.4. As indicated in the "Path Info" pane on the left of Figure 11.4, the second step of the highlighted scenario exploits the *LICQ* vulnerability on `shamong`. This suggests a possible strategy for reducing the size of the graph. Using the ANGI interface, we enable the firewall on the host `trenton`, and add a rule that blocks all external traffic at `trenton` from reaching `shamong` on the *LICQ* port. ANGI then generates a new XML model file reflecting this change. The new graph demonstrates a significant reduction in network exposure from this relatively small change in network configuration. The modification reduces graph size to 7 nodes and 6 edges. Figure 11.5 shows the graph for the modified network.

Looking at the scenarios in the new graph, we discover that the attacker can still reach `shamong` by first compromising the web server on `cherryhill`. Since we do not want to disable the web server, we enable the firewall on `mtholly` and add a rule specifically blocking `cherryhill`'s access to the *LICQ* client on `shamong`. Yet another invocation of the attack graph generator on the

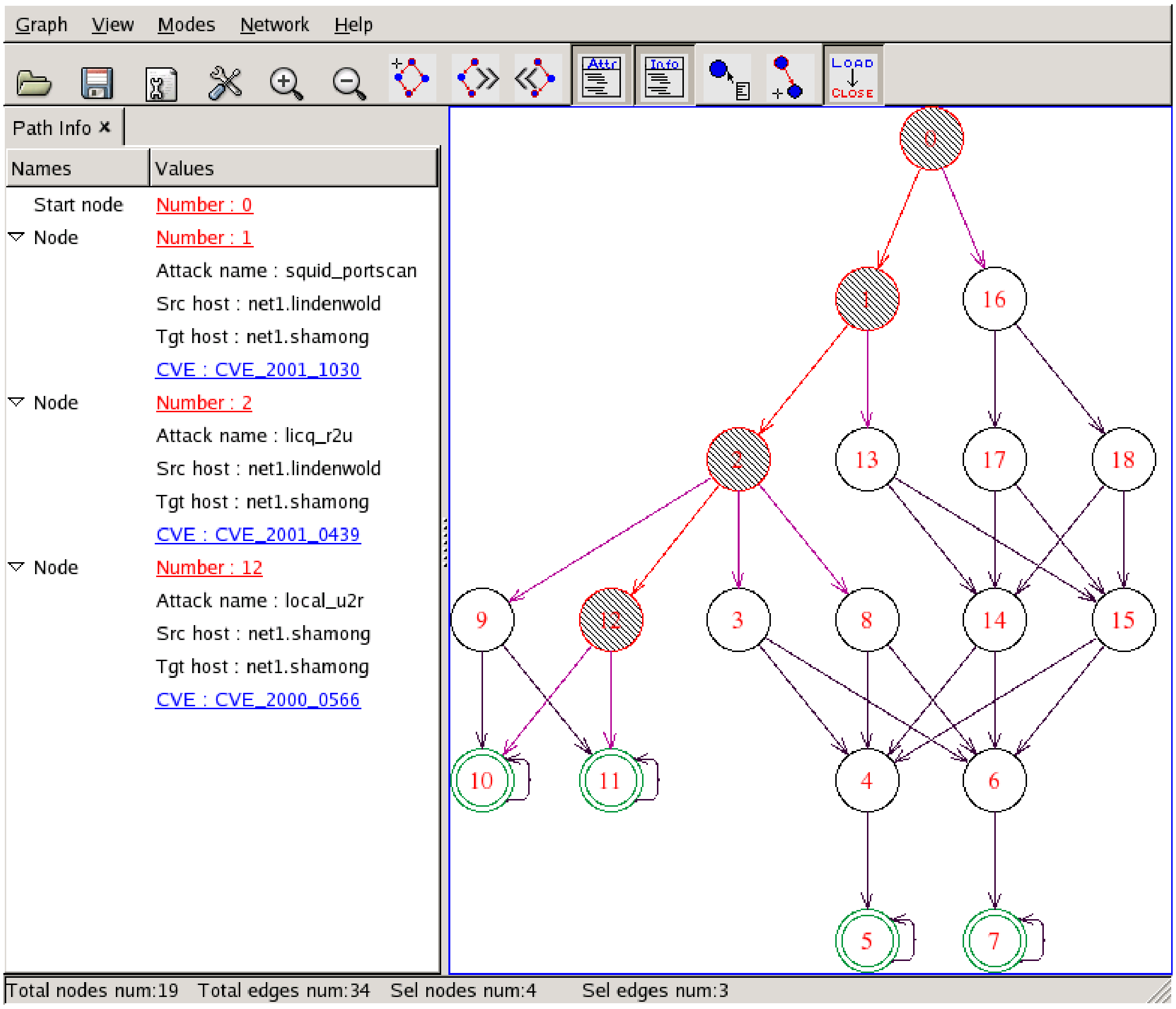


Figure 11.4: ANGI Attack Graph - No Firewalls

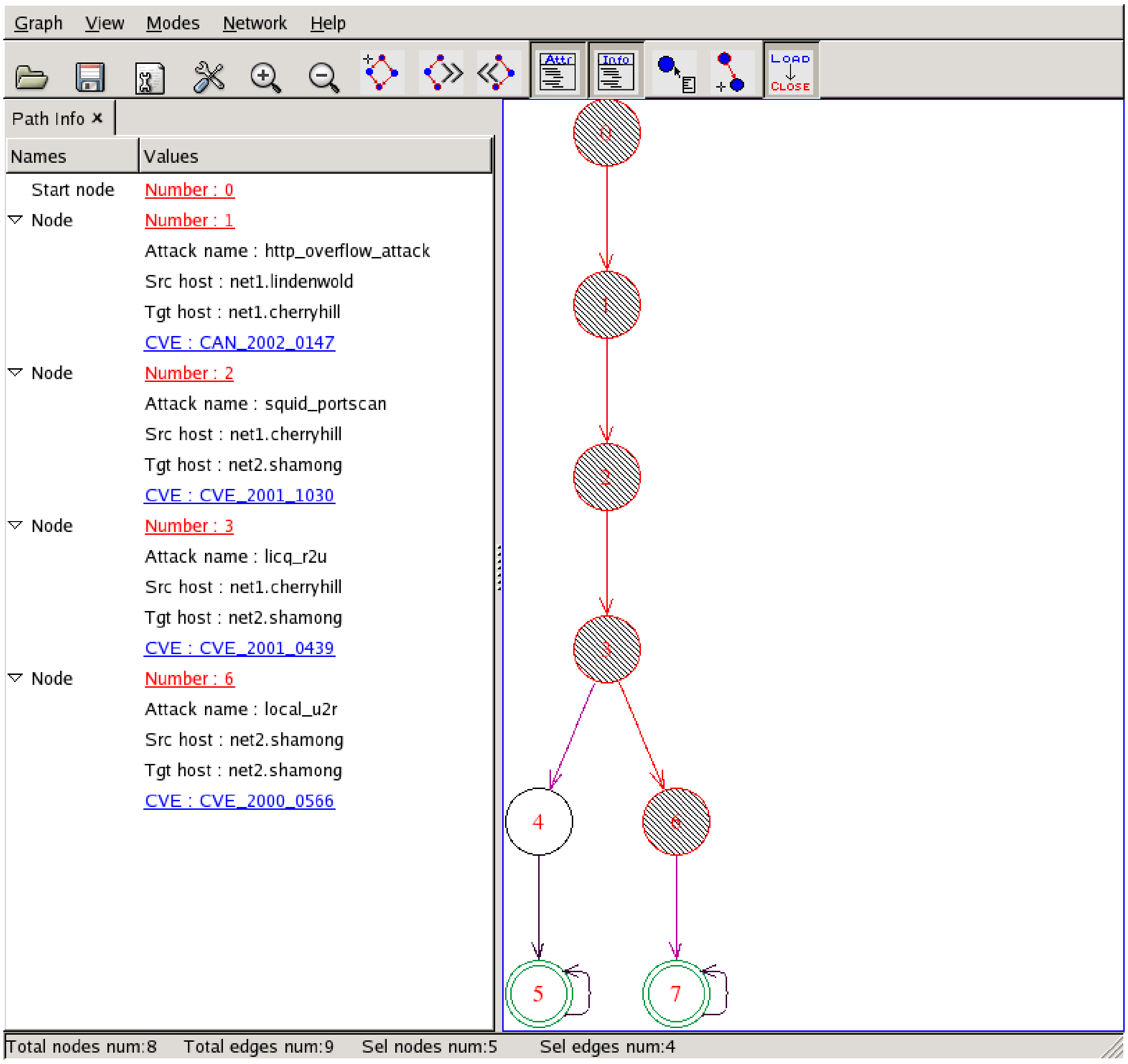


Figure 11.5: ANGI Attack Graph - One Firewall

modified model produces an empty attack graph and confirms that we have successfully safeguarded `shamong` while retaining the full functionality of the network.

Chapter 12

Attack Graphs: Related Work

Many of the ideas that we investigate in Part II have been suggested or considered in existing work in the computer security field. In this chapter we survey the related work.

Vulnerability scanning tools form a necessary part of any complete system for network vulnerability analysis, as they provide the input to build the network model. There are a variety of commercial and open source tools that scan single hosts for known vulnerabilities. Several prominent ones are COPS [1], CyberCop [5], and Nessus [30]. Each host running network services exposes vulnerabilities found by these scanners to hosts outside the network. Such exposures could be benign or hazardous in nature. However, plugging all the vulnerabilities based on the output of a scanning tool may render a network unusable to bona-fide users.

The focus of the work in Part II of this thesis is on chaining together the vulnerabilities uncovered by such tools to uncover end-to-end attack scenarios and thus discover the nature of the exposed vulnerabilities. Two distinct lines of work relate to our contributions to network security research: constructing multi-stage penetration scenarios, and building and analyzing entire attack graphs. Both lines of research have employed model checking technology. We start with the work on single attack scenarios.

Templeton and Levitt [91] propose a *requires/provides* model for modeling chains of network exploits. The model links exploits into scenarios, with earlier exploits supplying the prerequisites for the later ones. The *requires* part corresponds to our action preconditions, and the *provides* part corresponds to our action effects. They describe a language for specifying exploits and point out that relating seemingly innocuous system behavior to known attack scenarios can help discover new exploits.

Templeton and Levitt do not consider combining their attack scenarios into attack graphs; however, their ideas on reducing the complexity of the attack tree can be used within our model to reduce the size of the attack graph. We already abstract multiple vulnerabilities on a host, with similar effects, to a generic vulnerability and hence simplify the analysis. In future work, we would like to abstract a group of hosts with similar abstract vulnerabilities as a single node to further reduce model complexity.

Ramakrishnan and Sekar use a model checker to analyze single host systems with respect to combinations of unknown vulnerabilities [77]. Going further, Ritchey and Ammann use a model checker to construct single attack scenarios to test heterogeneous multi-host networks [79] with respect to known exploits, such as ones found on Bugtraq or ICAT Metabase. They use the (unmodified) model checker SMV [85] to construct the scenarios. Both Ramakrishnan and Sekar and Ritchey and Ammann obtain only one counter-example; that is, only one attack scenario ending in a state that violates the security specification.

Cuppens and Ortalo [26] propose a declarative language (LAMBDA) for specifying attacks in terms of pre- and post-conditions. LAMBDA is a superset of the simple language we used to model attacks in our work. The language is modular and hierarchical; higher-level attacks can be described using lower-level attacks as components. LAMBDA also includes intrusion detection elements. Attack specifications includes information about the steps needed to detect the attack and the steps needed to verify that the attack has already been carried out. Using a database of attacks specified in LAMBDA, Cuppens and Miege [25] propose a method for alert correlation based on matching post-conditions of some attacks with pre-conditions of other attacks that may follow. In effect, they exploit the fact that alerts about attacks are more likely to be related if the corresponding attacks can be a part of the same attack scenario. It would be interesting to apply similar alert correlation techniques to full scenario graphs.

In the references listed so far, the output is limited to a single attack scenario ending in a state that violates the security specification. Their focus is on building the network model and on creating flexible and portable languages for specifying attack steps. In contrast, we use model checkers NuSMV and SPIN to produce full attack graphs, representing all possible attack scenarios. We also describe, in Chapter 9 and elsewhere [52, 80], post-generation analyses of attack graphs that help administrators make decisions about improving network security. These analyses cannot be used on single attack scenarios.

Graph-based data structures have been used in network intrusion detection systems, such as *NetSTAT* [94]. There are two major components in NetSTAT, a set of probes placed at different points in the network and an analyzer. The analyzer processes events generated by the probes and generates alarms by consulting a network fact base and a scenario database. The network fact base contains information (such as connectivity) about the network being monitored. The scenario database has a directed graph representation of various atomic attacks. For example, the graph corresponding to an IP spoofing attack shows various steps that an intruder takes to mount that specific attack. The authors state that "in the analysis process the most critical operation is the generation of all possible instances of an attack scenario with respect to a given target network." As stated in the introduction to Part II, this is an important rationale for our application of model checking to attack graphs.

The earliest work in producing full attack graphs is the Kuang system [7], and its extension to a network environment, the NetKuang system [99]. In these systems a backward, goal-based search scans UNIX systems for poor configurations. The output is a combination of operations that lead to compromise.

Dacier [27] proposes a concept of *privilege graphs*. Each node in the privilege graph represents a set of privileges owned by the user; edges represent vulnerabilities. Privilege graphs are then explored to construct *attack state graphs*, which represents different ways in which an intruder can reach a certain goal, such as root access on a host. Dacier also defines a metric, called the *mean effort to failure* or METF. METF is a is probabilistic metric based on assigning likelihoods to attacks. Building on this work, Dacier et al [28] also use graph analysis for network security evaluation. Later, Ortalo et al. describe an experimental evaluation of a security framework based on these ideas [69].

Phillips and Swiler [72] propose the concept of attack graphs that is closest to the one described in Part II. If we have a finite state model of the system and restrict our attention to safety properties, an attack graph may be constructed by doing simple forward search. Starting with the initial states of the model M, we use a graph traversal procedure (e.g., depth first search) to find all reachable *fail* states where the security property P is violated. The attack graph is then the union of all paths from initial states to fail states. Swiler et. al. [89] use this strategy in their tool for generating attack graphs. Their tool constructs the attack graph by forward exploration starting from the initial state. The advantage of using model checking instead of forward search is that the technique can handle any security property expressible in temporal logic, including liveness properties.

Swiler et. al. eliminate redundant paths from their attack graphs with what amounts to an assumption of monotonicity, namely that the order of attacks does not matter. Their approach is fundamentally related to partial-order reduction techniques [92, 40, 45, 65, 71]. While it does not eliminate the fundamental exponential character of attack graphs, it does help control the visited parts of the state space. Swiler et. al. also consider cost functions, which we do not. It would be interesting to apply their techniques to our model.

Dawkins et. al. [29] specify another language for modeling exploits, and they use this language to provide a hierarchical view of attack trees. The hierarchy helps present information to the user in a more manageable way, but the approach does not have the goal directed aspect of the other references listed here. That is, their procedure results in a graph representing all possible compromises, and not just those of interest to a specific intruder goal.

On the surface, the notion of attack graphs proposed by Dacier and the one used by Phillips and Swiler are similar to ours. However, both of those attack graph definitions take an "attack-centric" view of the world. As pointed out above, our attack graphs are more general. Since we work with a general modeling language, the actions in our model can be both seemingly benign system events (such as failure of a link) and malicious events (such as attacks). Furthermore, in the future it will be easy to incorporate routine behavior of network users into the model and explore the interaction between legitimate and malicious activity.

Ammann et. al. address the exponential nature of the attack graphs common to Dacier, Phillips and Swiler, and our work. They present an implicit, scalable attack graph representation that avoids the exponential blow-up of explicit attack graphs [4]. Attack graphs are encoded as dependencies among exploits and security conditions, under the assumption of monotonicity. Informally, monotonicity means that no action an intruder can take interferes with the intruder's ability to take any other actions. The authors treat vulnerabilities, intruder access privileges, and network connectivity as atomic boolean attributes, and actions as atomic transformations that, given a set of preconditions on the attributes, establish a set of postconditions. In this model, monotonicity means that (1) once a postcondition is satisfied, it can never become 'unsatisfied', and (2) the negation operator cannot be used in expressing action preconditions.

The authors show that under the monotonicity assumption it is possible to construct an efficient (low-order polynomial) attack graph representation that scales well. They present an efficient algorithm for extracting minimal attack scenarios from the representation, and suggest that a standard graph algorithm can produce a critical set of actions that disconnects the goal state of the intruder from the initial state.

This approach is less general than our treatment of attack graphs. In addition to the monotonicity requirement, it can handle only simple safety properties. Further, the compact attack graph representation is less explicit, and therefore harder for a human to read. The advantage of their approach is that it has a polynomial worst-case bound on the size of the graph, and therefore in theory can scale better to large networks. It is also possible to perform certain useful analyses without instantiating the full attack graph.

Lye and Wing [62] present a game-theoretic method for analyzing the security of computer networks. Their model is similar to the one sketched in Appendix A. They view the interactions between an attacker and the administrator as a two-player general-sum stochastic game and construct a model for the game. Using a non-linear program, they compute Nash equilibria or best-response strategies for both players (attacker and administrator). Their work is complementary to ours: they report using our method for automatically generating attack graphs to replicate their example, which was originally generated manually. Thus, our model-checking algorithm can be used as the first step in their analysis, to generate the game tree automatically.

Chapter 13

Conclusions and Future Work

In this chapter we summarize our scientific contributions and disscuss avenues for further research.

13.1 Summary of Contributions

Traditional model checkers give users a single counterexample, expecting the user to fix the problem and run the model checker again. Sometimes, however, repeatedly checking for and fixing individual flaws does not work well. We developed formal techniques that give the user access to scenario graphs that represent the full set of faulty behaviors. Scenario graphs give the user more control over the design refinement process.

In Part I we specified scenario graphs formally and gave definitions for sound, exhaustive, and succinct scenario graphs. We presented two algorithms for automatically generating full scenario graphs from finite models. We have further specified a related algorithm that produces smaller graphs by replacing subsets of scenarios with representative prefixes. All of the algorithms have been shown to produce sound, exhaustive, and succinct graphs. The algorithms' performance is linear in the size of the reachable state space of the model.

Our implementation of the scenario graph algorithms exhibited linear performance in empirical tests. We also defined and implemented the traceback mode, a new technique for explicit state space search. In contrast with existing methods for reducing memory requirements, the traceback mode retains collision detection functionality in the hash table of states. Instead, memory savings are achieved at the cost of performance.

In Part II we applied our formal concepts to the security domain. Building on the foundation established in Part I, we defined attack graphs, an application of scenario graphs to represent ways in which intruders attack computer networks. We constructed a finite model of computer networks and applied scenario graph algorithms to the model to generate attack graphs. We showed how system administrators can use attack graphs to check how vulnerable their systems are and select additional measures to improve security.

We built an attack graph toolkit to support our generation and analysis algorithms. The toolkit has an easy-to-use graphical user interface. We integrated our tools with external sources to populate our network attack model with host and vulnerability data automatically. The toolkit has been used in collaboration with Lockheed Martin Advanced Technologies Laboratory for a small case study.

Any formal modeling technique has advantages and disadvantages when applied to a particular domain. The primary difficulty in our approach is in collecting the information used to build the network model. The attack graphs we produce are only as good as the inputs to the model builder. Every approach to information gathering that we covered in Chapter 11 has a negative side: the Nessus scanner provides incomplete information, Outpost and ANGI place onerous requirements on the software infrastructure, and hand-built models consume too many man-hours of labor.

A related difficulty is in modeling the actions of the attacker. New actions are devised all the time, and if we omit an attacker action, then the attack graph will give the user an incomplete picture of network vulnerability. These limitations are shared by other formal modeling techniques. Our formal framework at least gives system administrators a formal basis for making decisions relative to the accuracy of the input model.

13.2 Future Work

In this section we mention a few areas of interest where attack graph technology could be applied.

13.2.1 Library of Actions

Kannan, Sridhar, and Wing [54] plan to specify a library of actions based on a vulnerability database provided to us by the Computer Emergency Response Team (CERT). This database has over 150 actions representing many published CVEs. Kannan et al. used a subset of 30 of these actions as input to the model builder, allowing them to produce attack graphs with over 300 nodes and 3000 edges in a few minutes. With their growing library of actions, they intend to perform more systematic experiments: on different network configurations, with different subsets of actions, and for different attacker goals. Their preliminary results indicate that it is difficult to get useful information from large graphs by manual inspection. It is therefore essential to expand and implement our repertoire of automated analyses, and to explore techniques for reducing graph size.

13.2.2 Alert Correlation

We have already touched upon the necessity of correlating low-level intrusion detection alerts in Chapter 6. Ning, Cui, and Reeves [67] propose an approach for constructing individual attack scenarios by correlating alerts on the basis of prerequisites and consequences of actions. Their method correlates alerts by (partially) matching the consequence of some previous alerts and the prerequisite of some later ones.

By using full attack graphs instead of individual scenarios, we can further raise the level of reasoning about events in a network. Instead of looking at individual events in the network and trying to determine if one or a few of them might constitute good evidence that a particular attack is in progress, we focus instead on the intentions of the intruder. Recognizing that a series of alerts corresponds to a series of steps that could lead to system compromise could be a guide to distinguishing false alarms from real attacks.

The next step would be building a proof-of-concept intrusion detection system based on attack graphs. Given a database of graphs describing all likely attack scenarios, the system would match individual alerts to actions in the graphs. Matching successive alerts to individual paths in an attack graph dramatically increases the likelihood that the network is in fact under attack. The IDS would aggregate alarms to reduce the volume of alert information waiting for analysis, reduce the rate of false alarms, and try to predict intruder goals. Knowledge of intruder goals and likely next steps would help guide defensive response.

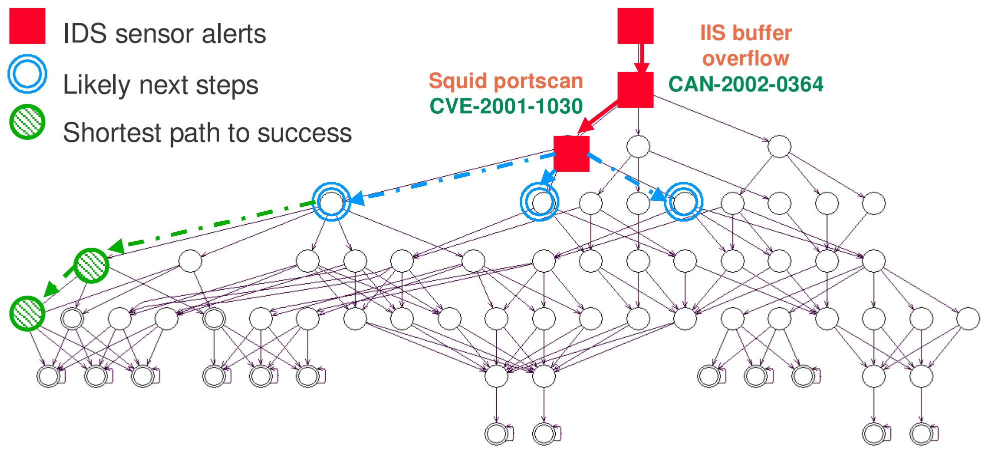


Figure 13.1: Alert Correlation with Attack Graphs

13.2.3 New Attack Discovery

By matching the parameters of seemingly innocuous system configuration with action prerequisites in known attack scenarios, it is possible to discover that the configuration can be turned into an exploit that will effectively plug into the attack scenario. Templeton and Levitt [91] cite an example of this strategy in action. By analyzing the peculiarities of Ethernet MAC control packets and the ARP protocol they discovered an exploit that could be used in conjunction with one of their individual attack scenarios. It would be interesting to explore this technique in conjunction with full attack graphs.

13.2.4 Prediction and Automated Response

Most of the attacks carried out today are scripted, with an entire scenario carried out in quick succession. With existing intrusion detection systems, the attack is usually over by the time the system administrator investigates the alarms. Only automated attack response can hope to prevent such attacks before they finish.

Prediction is a natural extension of alert correlation with attack graphs. If we are able to determine with a reasonable degree of certainty that the intruder has brought the system to a particular state in the attack graph, we can predict his immediate actions and, quite possibly, his ultimate goals. Such information can be an important aid as the system administrator deals with the intrusion.

13.2.5 Example: Alert Correlation, Prediction, and Response

We illustrate with an example a three-step strategy for correlating low-level alerts and then responding to the attack. Going back to the network we considered in Chapter 10, assume that we have compiled a database of attack graphs corresponding to various intruder goals. We add intrusion sensors to all hosts in the DMZ and on the internal network. The sensors watch the network traffic and report activity that could indicate that an attack is in progress. Such sensors often have high false alarm rates, and it is not feasible to investigate every alarm. However, armed with pre-computed attack graphs, we can make good use of the noisy sensor data.

Suppose, for instance, that we receive an alert indicating that a remote buffer overflow action may have been executed against the IIS Web Server, followed by an alert about a possible Squid portscan attack on the `Linux` host. The first step is to match the alerts to the attack graphs in our database. The alerts correspond to attacks that form the first two steps of an attack scenario in the attack graph that we saw in Section 10.2. These steps are indicated with boxed nodes in Figure 13.1. Despite low confidence in the accuracy of the individual sensor alerts, we now have a strong indication that a real intrusion is in progress.

Next, we can use the attack graph to form hypotheses about the intruder's current progress, his likely next action, and his ultimate goal. Assuming that the intruder's progress corresponds to the lower of the three boxed states in Figure 13.1, we can focus our attention on the subset of attack scenarios reachable from that state. The intruder's likely next step corresponds to one of the transitions out of the lower boxed state to a double-circle state.

We can further hypothesize that the intruder is likely to take the shortest route to his goal, meaning that his most likely next step is the leftmost transition (that is, a remote compromise of the *LICQ* client of the `Linux` host). The IDS software can now attempt to disrupt the intruder with an appropriate counter-action: disabling the *LICQ* client on the `Linux` host or closing the internal firewall to cut off the intruder's link to the host.

13.3 Final Word

Attack graphs have been a subject of study by security researchers for several years, and they served as the original motivation for our work on scenario graphs. Based on the inquiries received in the course of our work, we believe that the attack graph application will continue to generate interest in the future. However, the field is wide open for more applications of scenario graphs. We believe that other kinds of systems can benefit from comprehensive modeling of faulty behavior and hope to see more research in this area in the near future.

Appendix A

Attack Graphs and Game Theory

In this appendix we develop game-theoretic models for describing systems under attack and show how to generate attack graphs for such models with respect to particular security properties.

A.1 Background: Extensive-Form Games

Game theory uses the *extensive* form to describe games where the players make multiple moves and base their strategy at each turn on information about moves that have already occurred. Our treatment of extensive-form games mostly follows [37], Chapter 3; exceptions are noted explicitly. Extensive form games contain the following information:

1. the set of players, denoted by $\mathcal{I}$,
2. the set of all possible actions (moves), denoted by A,
3. the order of moves (who moves when),
4. what the players' available actions are when it's their turn to move,
5. the players' payoffs as a function of the moves that were made,
6. what each player knows when he makes his choices, and
7. the probability distributions over any exogenous events, represented as moves by a special player *Nature* and denoted by $N \in \mathcal{I}$.

A finite *game tree* T represents the order of play and the players' available choices at each stage in the game (points 3 and 4). Using a finite tree to represent game histories rules out infinite games. In addition, in a tree each node (except the root node) has exactly one predecessor, which means that distinct game histories cannot converge at a single node. Each game tree node $s \in S$ is thus a complete description of the history up to that point in the game.

To complete the specification of points 3 and 4, in addition to the game tree we introduce two associated functions. The state-ownership map $o : S \to \mathcal{I}$ specifies which player may perform an action in each state. We say that the player $\mathtt{o}(s)$ *owns* state s. The *action assignment* function $a : S \to A$ labels each non-initial node s with the last action taken to reach it. We require that a be one-to-one on the set of immediate successors of each node, so that different successors correspond to different actions. We say that $A(s)$ is the set of actions available at s. Thus, $A(s)$ is the range of a on the set of immediate successors of s.

The players' payoffs are represented by utility functions $u_i : S \rightarrow R$. In a slight departure from [37], we assign a payoff to every node (instead of just the terminal nodes), with $u_i(s)$ being player i's payoff if the game is terminated at node s.

Point 6, the information players have when choosing their actions, is the most subtle part of extensive form games. This information is represented using *information sets* $h \in H$, which partition the set of nodes in the tree - that is, every node is in exactly one information set. The interpretation of the information set $h(s)$ containing node s is that the player who is choosing an action at s is uncertain if he is at s or at some other node $s' \in h(s)$. To prevent ambiguity with move ordering, we require that the same player must move in all nodes of an information set, and further that the player must have identical action choices in all nodes of an information set. Thus, we can speak of an action set $A(h)$ of an information set h. In the special case of games of *perfect information* all information sets are singletons - the player knows at all times exactly where in the game tree the current configuration is located. However, in our intended area of application it is unrealistic to assume perfect information for either the intruder or the defender of a network.

In summary, an extensive form game is a tuple $\mathcal{G} = (\mathcal{I}, A, T, o, a, \{u_i | i \in \mathcal{I}\}, H)$ incorporating the set of players, the set of actions, the game tree, the state ownership and action assignment functions, utility functions for each player, and the information sets.

A.2 Game-Theoretic Attack Models

In game-theoretic terms, an *attack model* $\mathcal{W} = (\mathcal{I}, A, T, o, a, \{u_i | i \in \mathcal{I}\}, H)$ is an extensive-form game with three players $\mathcal{I} = \{a, \mathtt{D}, \mathtt{S}\}$ representing an attacker, a defender, and the system, respectively. With each player $\mathtt{P} \in \mathcal{I}$ we associate actions in the set $A_{\mathtt{P}}$, so that the total set of action in the game is $A = \bigcup_{\mathtt{P} \in \mathcal{I}} A_{\mathtt{P}}$. In general, the attacker's actions move the system "toward" some undesirable (from the system's point of view) state, and the defender's actions attempt to counteract that effect.

The game tree T, with node set S and transition relation $\tau : S \times S$, can take any shape, subject to the restrictions imposed by the extensive form game formalism. There is a single root state s_0 representing the initial state of each player before any action has taken place.

For the purpose of generating attack graphs with the techniques developed in Part I, an attack model $\mathcal{W}$ so defined may be recast as a Büchi model $M_{\mathcal{W}} = (S, \tau, \{s_0\}, S, S, D)$, with the node set and transition relation of the game tree interpreted, respectively, as the state space and transition relation of the Büchi model. The labeling D of the Büchi model encodes the state ownership and action assignment functions: for all states s, $D(s) = \{owner = o(s), action = a(s)\}$.

At the end of the game the attacker and the defender get a payoff that generally depends on the degree of attacker's success. In traditional game-theoretic terminology, the system plays the role of *Nature* and does not get payoffs. In contrast, the attacker and the defender are *active* players. Each active player $\mathtt{P}$ has a total payoff function $u_{\mathtt{P}} : S \rightarrow R$, associating a payoff with every state $s \in S$.

The simplest case has two possible payoffs for each active player, $\{$*succ = 1, fail = -1*$\}$. More complicated situations with multiple prioritized attacker targets would have commensurately more complex payoff functions. The defender may likewise have a more complicated payoff matrix. For instance, it may be possible for the defender to close the firewall and prevent further incursions by the attacker, but the closed-firewall outcome is less desirable than the status-quo since legitimate users are also prevented from accessing the network. This difference will be reflected in the defender's payoffs.

A.3 Attack Graphs

The full game tree is typically very large for any attack model that describes a non-trivial system. As we have explained previously, for the purposes of vulnerability assessment and defense planning we are particularly interested in games where the attacker is successful. Intuitively, an attack graph is a subgraph of the game tree where in each final state the attacker's payoff is *succ* and the defender's payoff is *fail*. The difference between the game tree and the attack graph is that the latter can coalesce identical subtrees into a single branch, and can represent infinite behaviors with cycles. More formally, an attack graph is a safety scenario graph with an appropriately defined safety property (Section 2.1.8):

Definition A.1 *Given an attack model $\mathcal{W}$, define a safety property P with the logical predicate $Q_P : S \rightarrow bool$ such that $Q_P(s)$ if and only if $u_A(s) =$* fail. *An* attack graph *of $\mathcal{W}$ is a safety scenario graph $G_a = (S_a, \tau_a, \{s_0\}, S_f, D)$, where $S_a \subseteq S$, $\tau_a \subseteq \tau$, and $S_f = \{s : S_a | \neg Q_P(s)\}$.*

Using the game-theoretic definitions of attack models and attack graphs, we can model situations where the network actively defends itself against attack. This is an interesting research direction that we hope will be explored in the future. In Sections A.5 and A.6 we offer examples of future research directions.

A.4 Behavior Strategies

Let $S_i \equiv \{s \in S \mid o(s) = i\}$ be the set of states owned by player i. Let $A_i \equiv \bigcup_{s \in S_i} A(s)$ be the set of all actions for player i. A *pure strategy* for player i is a map $ps_i : S_i \rightarrow A_i$, with $ps_i(s) \in A(s)$ for all states s. Player i's space of pure strategies, PS_i, is simply the space of all such ps_i. A *strategy profile* assigns a strategy to each player in the game.

Given a pure strategy for each player i and the probability distribution over Nature's (in our case, the Network's) moves, we can compute a probability distribution over outcomes and thus assign expected payoffs $u_i(pf)$ to each strategy profile pf.

A.4.1 Strategic Equilibria - Pure Strategies

A pure-strategy Nash equilibrium for an extensive-form game is a strategy profile pf^* such that each player i's strategy pf_i^* maximizes his expected payoff given the strategies of his opponents. Many complex games do not admit pure-strategy equilibria, but pure strategies suffice for games of perfect information.

Theorem A.1 *(Zermelo 1913; Kuhn 1953) A finite game of perfect information has a pure-strategy Nash equilibrium.*

The proof of this theorem doubles as a backward-induction algorithm for constructing the equilibrium. Since the game is finite, it has a set of penultimate nodes – i.e., nodes whose immediate

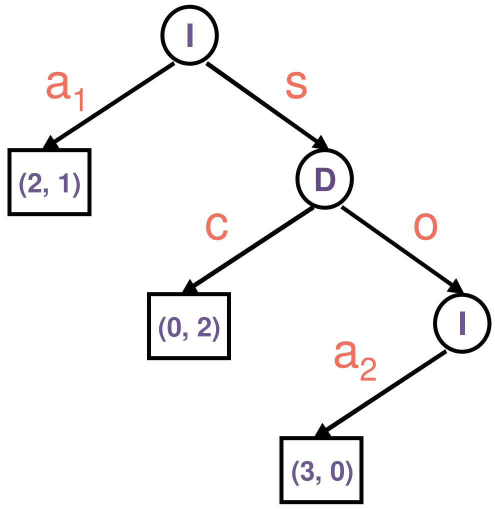


Figure A.1: Risk of Discovery Game Tree

successors are terminal nodes. Specify that the player who can move at each such node chooses whichever strategy leads to the successive terminal node with the highest payoff for him. Now specify that each player at nodes whose immediate successors are penultimate nodes choses the action that maximizes her payoff over the feasible successors, given that players at the penultimate nodes play as we have specified in the previous step. We can now roll back through the tree, specifying actions at each node. When we are done, we will have specified a strategy for each player, and it is easy to check that these strategies form a Nash equilibrium.

A.4.2 Strategic Equilibria - Mixed Strategies

Since it is unreasonable to expect either the attacker or the defender to have complete knowledge of the state of the network during an intrusion, we must consider strategic equilibria for games of imperfect information. Such games require players to "mix" their strategies to come out ahead (poker is a well-known example). Game theory devotes much research to so-called *mixed strategies* that specify a probability distribution over actions at each state.

Recall that extensive-form games model incomplete information with a collection of player-specific sets of information sets $\{H_i | i \in \mathcal{I}\}$. The player choosing an action at state s is uncertain if he is at s or at some other node $s' \in h(s)$. Let $\delta(A(h))$ denote the space of all possible probability distributions on actions available at information set h.

A *behavior strategy for player i*, denoted b_i, is an element of the Cartesian product $\times_{h_i \in H_i} \delta(A(h_i))$. That is, a behavior strategy specifies a probability distribution over possible actions for each information set of the player. The probability distributions at different information sets are independent. In a pure strategy, the distribution is degenerate in each information set: one action has probability one, and the rest probability zero. If we specify a behavior strategy for each player, we get a *strategy profile* $b = (b_1, \ldots, b_{\mathcal{I}})$, which generates a probability distribution over outcomes and thus gives rise to an expected payoff for each player. A Nash equilibrium in behavior strategies is a profile such that no player can increase his expected payoff by using a different behavior strategy.

A.5 Dynamic Analysis: Risk of Discovery

Consider a game-theoretic attack model with the game tree depicted in Figure A.1. The attacker (I) wants to get at a database on a protected machine. He would prefer to get administrative access to the machine (payoff=3) but may content himself with reading some of the data (payoff=2). The attacker has available two alternative approaches. He may go for a simple one-step penetration (action a_1), perhaps accessing the database with a password obtained via a little social engineering. Or he may choose a more sophisticated route, running a vulnerability scan (s) followed by an appropriately-prepared break-in to gain full control of the target database (a_2). The former approach nets his some data out of the database, but the latter gives him a shot at the optimal payoff (3).

The snag is that the scanning action may be detected by the defender, who may opt to close the firewall (c) and prevent action a_2. In that case the attacker gets nothing (payoff 0). Since this is a game of perfect information, the backward-induction algorithm is applicable and yields the Nash strategy profile $((a_1, a_2), c)$. Knowing that the scan will alert the defender and lead to a firewall closure, the attacker must reject the sophisticated break-in and choose the lower-payoff approach.

Despite its simplicity, the example highlights one of the major insights of game theory—that the threat of counter-action by an opposing agent has a fundamental significance in the construction of an optimal strategy for any game player. In the context of network security this means that the utility of actively vigilant defense (perhaps involving intrusion detection and dynamic adjustment of security settings) goes beyond prevention of specific attacks. The mere threat of counter-measures is sufficient to alter attacker's strategy, perhaps channeling it in less destructive directions.

A.6 Dynamic Network Defense Evaluation Methodology

The impact of a static network defense measure can be evaluated by including the measure in the model of the network and re-running the security analysis. A similar approach can work for dynamic defense solutions, but it calls for a game-theoretic treatment that accounts for attacker's strategies designed to cope with the active defense. In general outline the method consists of the following steps:

1. Model the network and attacker behavior with the defensive actions included as full-fledged transitions of the state machine.
2. Run attack graph generator to produce the game tree of the attack model.
3. Apply backward induction (or other appropriate game-theoretic algorithm) to compute optimal strategies for both the attacker and the defender.

The outcomes achieved by the attacker and the defender with their respective optimal strategies give an indication of the effectiveness of a particular dynamic defense mechanism. As a side benefit, the analysis produces a concrete strategy that can be programmed into the defensive software.

Appendix B

Probabilistic Scenario Graphs

When we are modeling a system designed to operate in an uncertain environment, certain transitions in the model represent the system's reaction to changes in the environment. We can think of such transitions as being outside of the system's control—they occur when triggered by the environment. When no empirical information is available about the relative likelihood of such environment-driven transitions, we can model them only as nondeterministic "choices" made by the environment. However, sometimes we have empirical data that make it possible to assign probabilities to environment-driven transitions. We would like to take advantage of such information to quantify the probabilistic behavior of scenario graphs. In this section we show how to incorporate probabilities into scenario graphs and demonstrate a correspondence between the resulting construction and Markov Decision Processes [3] (MDPs). The correspondence lets us take advantage of the existing body of algorithms for analyzing MDPs.

One way to incorporate probabilities into scenario graphs is to choose a subset of states and make transitions out of those states probabilistic. Suppose that the graph has a state s with only two outgoing transitions. In a regular scenario graph the choice of which transition to take when the system is in s is nondeterministic. However, we may have some empirical data that enables us to estimate that whenever the system is in state s, on average it will take one of the transitions four times out of ten and the other transition the six remaining times. We can place probabilities 0.4 and 0.6 on the corresponding edges in the scenario graph. We call a state with known probabilities for outgoing transitions *probabilistic*. When we have assigned all known probabilities in this way, we are left with a scenario graph that has some probabilistic and some nondeterministic states in it. We call such mixed scenario graphs *probabilistic scenario graphs*.

B.1 Probabilistic Scenario Graphs

For the rest of the appendix, we assume that the transition relation τ is total in both the Büchi model and the scenario graph in question. Since the scenario graph includes only those scenarios that violate the correctness property, it excludes some states and transitions that exist in the input Büchi model M. These excluded transitions can have non-zero probability, so that in their absence the sum of probabilities of transitions from a probabilistic state will be less than 1. To address this problem, we add a catch-all *escape* state s_e to the scenario graph to model the excluded part of M. The escape state is neither final nor accepting. A probabilistic state s in the scenario graph will have a transition to s_e if and only if in M there is a transition from s to some state that does *not* appear in the scenario graph. The probability of going from s to s_e is computed by subtracting from one the sum of the probabilities of going to all other states represented in the scenario graph. There are no transitions out of s_e except a self-loop (preserving the totality of τ).

We call scenarios terminating in s_e *escape scenarios*, to distinguish them from failing scenarios. A scenario graph augmented with an escape state in this manner still accepts only the set of failing scenarios; however, in the following discussion it will be handy to have a name for escape scenarios.

Definition B.1 *A* probabilistic scenario graph *is a tuple $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_a, S_f, D)$, where S_n is a set of nondeterministic states, S_q is a set of probabilistic states, $s_e \in S_n$ is a nondeterministic escape state ($s_e \notin S_a$ and $s_e \notin S_f$), $S = S_n \cup S_q$ is the set of all states, $\tau \subseteq S \times A \times S$ is a transition relation, $\pi : S_q \to S \to \Re$ are transition probabilities, $S_0 \subseteq S$ is a set of initial states, $S_a \subseteq S$ is a set of acceptance states, $S_f \subseteq S$ is a set of final states, and $D : S \to 2^A (A = 2^{AP})$ is a labeling of states with sets of alphabet letters.*

A probabilistic scenario graph (PSG) distinguishes between nondeterministic states (set S_n) and probabilistic states (set S_q). The sets of nondeterministic and probabilistic states are disjoint ($S_n \cap S_q = \oslash$). The function π specifies probabilities of transitions from probabilistic states, so that for all transitions $(s_1, s_2) \in \tau$ such that $s_1 \in S_q$, we have $\mathbf{Prob}(s_1 \to s_2) = \pi(s_1)(s_2) > 0$. Thus, $\pi(s)$ can be viewed as a probability distribution on next states. Intuitively, when the system is in a nondeterministic state s_n, we have no information about the relative probabilities of the possible next transitions. When the system is in a probabilistic state s_q, it will choose the next state according to probability distribution $\pi(s_q)$.

A common objection to modeling software systems with probabilities is that these probabilities are difficult or impossible to obtain, and nobody has a reason to try. These questions are legitimate, but they are beyond the scope of this work. The chicken-and-egg cycle must be broken somehow or other, and if we can propose compelling ways of incorporating probabilities into reliability analysis of systems, that may provide impetus for other researchers to obtain these probabilities in practice.

B.1.1 Alternating Probabilistic Scenario Graphs and Markov Decision Processes

In this section we show that probabilistic scenario graphs are equivalent to Markov Decision Processes (without the cost function). We then demonstrate how a *benefit* function can be assigned to probabilistic scenario graphs such that standard MDP algorithms compute answers to interesting quantitative questions about the corresponding PSG.

Definition B.2 *[3] A Markov Decision Process is a tuple $(\mathbf{X}, A, \mathcal{P}, c)$ where*

- *$\mathbf{X}$ is a finite state space. Generic notation for MDP states will be x, y, z.*
- *A is a finite set of actions. $A(x) \subseteq A$ denotes the actions that are available at state x. The set $\mathcal{K} = \{(x, a) : x \in \mathbf{X}, a \in A(x)\}$ is the set of state-action pairs. A generic notation for an action will be a.*
- *$\mathcal{P} : \mathbf{X} \times A \times \mathbf{X}$ are the transition probabilities; thus, $\mathcal{P}(x, a, y)$ (also written as $\mathcal{P}_{xay}$) is the probability of moving from state x to y if action a is chosen.*
- *$c : \mathcal{K} \to \Re$ is an immediate cost. The cost may be equivalently viewed as a negative benefit. We will freely use the term* benefit *to mean the negative cost, and vice versa.*

An execution fragment of an MDP is a sequence $x_0 a_1 x_1 \ldots a_n x_n$ of alternating states and actions such that the sequence begins and ends with a state, and for all $0 < k \leq n$, $0 < \mathcal{P}(x_{k-1}, a_k, x_k) \leq 1$.

It is possible to convert a probabilistic scenario graph into an MDP such that the behaviors of the PSG and the MDP are identical. To explain the conversion procedure, we define a restricted kind of probabilistic scenario graph.

Definition B.3 *An* alternating probabilistic scenario graph *is a tuple $G = (S_n, S_q, s_e, S, \tau_n, \tau_q, \pi, S_0, S_a, S_f, D)$, where S_n is a set of nondeterministic states, S_q is a set of probabilistic states, $s_e \in S_n$ is a nondeterministic escape state, $S = S_n \cup S_q$ is the set of all states, $\tau_n \subseteq S_n \times S_q$ is a set of nondeterministic transitions, $\tau_q \subseteq S_q \times S_n$ is a set of probabilistic transitions, $\pi : S_q \to S_n \to \Re$ are transition probabilities, $S_0 \subseteq S$ is a set of initial states, $S_a \subseteq S$ is a set of acceptance states, $S_f \subseteq S$ is a set of final states, and $D : S \to A (A = 2^{AP})$ is is a labeling of states with sets of alphabet letters.*

An alternating probabilistic scenario graph (APSG) does not have any transitions between two consecutive nondeterministic or two consecutive probabilistic states. In other words, a nondeterministic state has transitions to probabilistic states only, and vice versa. An execution of an APSG will always have strictly alternating nondeterministic and probabilistic states.

The algorithm shown in Figure B.1 converts a PSG $G_p = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_a, S_f, D)$ into an APSG $G'_p = (S'_n, S'_q, s_e, S, \tau_n, \tau_q, \pi', S_0, S_a, S_f, D')$ that has equivalent behaviors. The algorithm works by adding *hidden* states and transitions to the graph such that every execution becomes strictly alternating, yet does not change its *observable* (non-hidden) components.

We start with $S'_n = S_n$, $S'_q = S_q$, $\tau'_n := \oslash$, $\tau'_q := \oslash$, $\pi' := 0.0$, and $D' = D$. Next,

1. Whenever τ has a transition from probabilistic state s_1 to nondeterministic state s_2, we add the transition to τ_p and its probability to π'.

2. Whenever τ has a transition from nondeterministic state s_1 to probabilistic state s_2, we add the transition to τ_n.

3. Whenever τ has a transition between two nondeterministic states s_1 and s_2, we add a hidden probabilistic state s_h to S'_q, an observable transition $s_1 \to s_h$ to τ_n, and a hidden transition $s_h \to s_2$ to τ_p, assigning the latter probability 1 in π' (Figure B.2a). We also set $D'(s_h) = D(s_1)$.

4. Whenever τ has a transition between two probabilistic states s_1 and s_2, we add a hidden nondeterministic state s_h to S'_n, a hidden transition $s_h \to s_2$ to τ_n, and an observable transition $s_1 \to s_h$ to τ_p, assigning the latter the original probability p of going from s_1 to s_2 (Figure B.2b). We also set $D'(s_h) = D(s_1)$.

Let e be an execution of G'. We define $obs(e) = e^{obs}$ by removing hidden states and hidden transitions from e.

Lemma B.1 *If e is an execution of G', then e^{obs} is an execution of G.*

Proof:

Input:
$G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_a, S_f, D)$ – a PSG
Output:
$G' = (S'_n, S'_q, s_e, S, \tau_n, \tau_q, \pi', S_0, S_a, S_f, D')$ – an APSG

Algorithm: PSG-to-APSG$(S_n, S_q, s_e, S, \tau, \pi, S_0, S_a, S_f, D)$
$S'_n := S_n$; $S'_q := S_q$
$\tau'_n := \oslash$; $\tau'_q := \oslash$;
$\forall(s_q, s_n) : \pi'(s_q, s_n) := 0.0$;
$D' := D$

foreach transition $(s_1, D(s_1), s_2)$ **in** τ **do**
  if s_1 is probabilistic, s_2 is nondeterministic **then** *// Case 1*
    $\tau_p := \tau_q \cup (s_1, s_2)$; $\pi'(s_1)(s_2) := \pi(s_1)(s_2)$
  elseif s_1 is nondeterministic, s_2 is probabilistic **then** *// Case 2*
    $\tau_n := \tau_n \cup (s_1, s_2)$
  elseif both s_1 and s_2 are nondeterministic **then** *// Case 3*
    create hidden probabilistic state s_h
    create observable transition (s_1, s_h)
    create hidden transition (s_h, s_2)
    $\tau_n := \tau_n \cup (s_1, s_h)$
    $\tau_q := \tau_q \cup (s_h, s_2)$
    $\pi'(s_h)(s_2) := 1$
    $S'_q := S'_q \cup s_h$
    $D'(s_h) := D(s_1)$
  elseif both s_1 and s_2 are probabilistic **then** *// Case 4*
    create hidden nondeterministic state s_h
    create observable transition (s_1, s_h)
    create hidden transition (s_h, s_2)
    $\tau_n := \tau_n \cup (s_h, s_2)$
    $\tau_q := \tau_q \cup (s_1, s_h)$
    $\pi'(s_1)(s_h) := \pi(s_1)(s_2)$
    $S'_n := S'_n \cup s_h$
    $D'(s_h) := D(s_1)$
  end
end

Figure B.1: Converting PSG to APSG

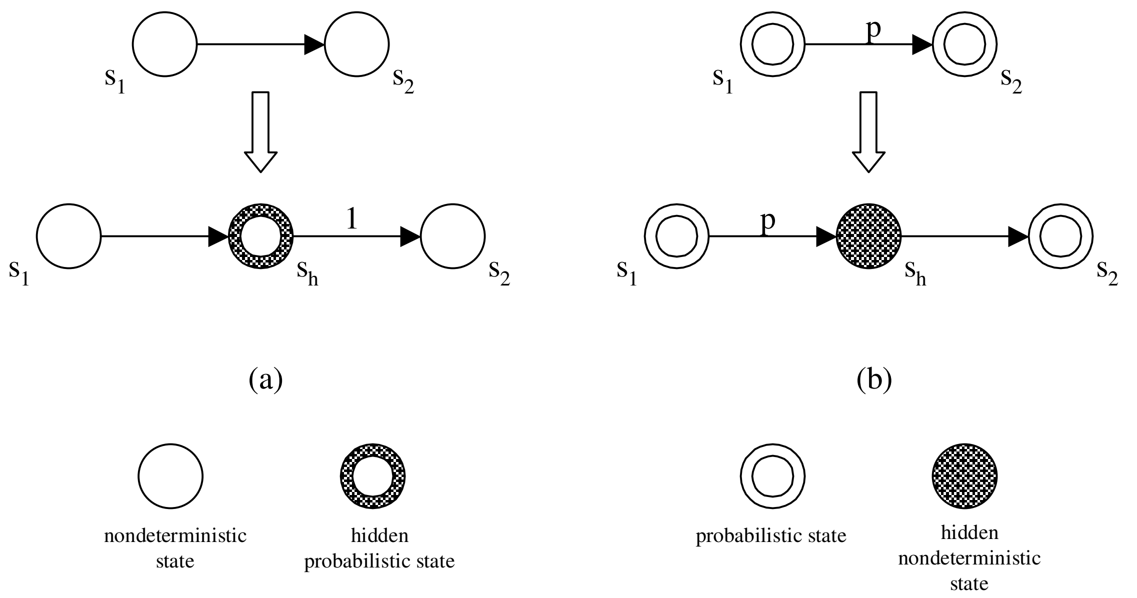


Figure B.2: Converting PSG to APSG

e can be represented as $e_0 r_1 h_1 t_1 e_1 \dots e_{n-1} r_n h_n t_n e_n$, where $h_0, \dots, h_n$ are hidden states, $r_0, \dots, r_n$ are transitions into hidden states, $t_0, \dots, t_n$ are transitions out of hidden states, and $e_0, \dots, e_n$ are executions consisting only of observable states.

We observe that $e_0, \dots, e_{n+1}$ is also an execution in G. We can see that this is so by inspecting how the algorithm adds observable states and transitions to $G^{'}$. All observable states are added from G at initialization, so that each observable state in $G^{'}$ also occurs in G. All transitions between two observable states are added by cases 1 or 2 of the algorithm in Figure B.1, so that each such transition also occurs in G.

The proof proceeds by induction on n. For $n = 0$, $e = e^{obs} = e_0$, which is an execution fragment of G. For $n = k$, $e = e_p r_k h_k t_k e_k$, where the prefix fragment e_p is $e_0 r_1 h_1 t_1 e_1 \dots e_{k-1}$. There are two cases:

Case 1: h_k is a hidden probabilistic state. h_k was added to G by case 3 of the algorithm. So t_k is a hidden transition, r_k is an observable transition, and $e^{obs} = e_p^{obs} r_k e_k^{obs}$.

e_k^{obs} is an execution fragment in G. By the induction hypothesis, e_p^{obs} is an execution in G. Further, by the precondition of case 3 of the algorithm G must have a nondeterministic transition between the last state of e_p^{obs} and the first state of e_k^{obs}. Therefore, e^{obs} is an execution in G.

Case 2: h_k is a hidden nondeterministic state. h_k was added to G by case 4 of the algorithm. So t_k is a hidden transition, r_k is an observable transition, and $e^{obs} = e_p^{obs} r_k e_k^{obs}$.

e_k^{obs} is an execution in G. By the induction hypothesis, e_p^{obs} is an execution in G. Further, by the precondition of case 4 of the algorithm in Figure B.1, G must have a probabilistic transition between the last state of e_p^{obs} and the first state of e_k^{obs}, with the same probability as r_k. Therefore, e^{obs} is an execution in G.

$\square$

Lemma B.2 *obs is a 1-to-1 correspondence between successful executions of G' and G.*

Proof:

onto Let e be a successful execution in G. We can construct a pre-image e_h of e as in the algorithm for converting PSGs to APSGs. Replace each transition between two consequent nondeterministic states with two transitions, to and from a hidden probabilistic state (Figure B.2a). Replace each transition between two consequent probabilistic states with two transitions, to and from a hidden nondeterministic state (Figure B.2b). It is easy to prove by induction that the resulting sequence e_h is a successful execution in G', and that $obs(e_h) = e$.

1-to-1 The inductive proof is easier if we rewrite subscripts as follows: $e_h = s_0 t_1 s_2 \ldots t_{n-1} s_n$. To prove that obs is 1-to-1, suppose there is an execution $e'_h = s'_0 t'_1 s'_2 \ldots t'_{n-1} s'_n$ in G' such that $obs(e'_h) = e$. We show that $e'_h = e_h$ by induction on subscript n.

Letting $n = 0$, we have $e_h = s_0$ and $e'_h = s'_0$. s_0 and s'_0 are both initial states in G', therefore they are observable. We have $e = obs(e_h) = s_0$ and $e = obs(e'_h) = s'_0$. So $s_0 = s'_0$, and $e_h = e'_h$.

Letting $n = k$ (where $k > 0$), we have $e_h = e_{k-1} x_k$, where x_k is a state when k is even and a transition when k is odd, and e_{k-1} is the prefix $s'_0 t'_1 \ldots x_{k-1}$. Similarly, $e'_h = e'_{k-1} x'_k$. Note that by inductive hypothesis $e'_{k-1} = e_{k-1}$, and since $obs(e'_h) = obs(e_h)$, we must have $obs(x'_k) = obs(x_k)$.

We would like to show that $x'_k = x_k$. There are three cases.

Case 1: x'_k is a state (hidden or observable). This state is uniquely determined by transition t'_{k-1}. Since by inductive hypothesis $t'_{k-1} = t_{k-1}$, x'_k must be the same state as x_k.

Case 2: x'_k is an observable transition. Since $obs(x'_k) = obs(x_k)$, x'_k must be the same observable transition as x_k.

Case 3: x'_k is an hidden transition. By construction of G' s'_{k-1} must be a hidden state. By inductive hypothesis $s'_{k-1} = s_{k-1}$. Since a hidden state has exactly one outgoing transition, x'_k must be the same transition as x_k.

□

Definition B.4 *Define the* behavior *of a scenario graph G as the set of observable successful executions of G:* $Beh(G) = \{e : \exists e' \in L(G) \,.\, e = obs(e')\}$.

Theorem B.1 $Beh(G') = Beh(G)$.

Proof: Follows immediately from Lemma B.2 □

We have shown that APSGs have the same expressive power as PSGs, so hereafter we consider them interchangeable.

An APSG $G = (S_n, S_q, s_e, S, \tau_n, \tau_p, \pi, S_0, S_a, S_f, D)$ has a direct interpretation as an MDP $M_G = (\mathbf{X}, A, \mathcal{P}, c)$. Let $\mathbf{X} = S_n$, $A = \tau_n$. That is, each action represents a transition from a nondeterministic to a probabilistic state. Further, let $x, y \in \mathbf{X}$ and $a \in A(x)$, so that a represents a transition from x to some probabilistic state $s_q \in S_q$ in the APSG. Then we have $\mathcal{P}(x, a, y) = \pi(s_q)(y)$. For now, we leave the cost function c uninterpreted.

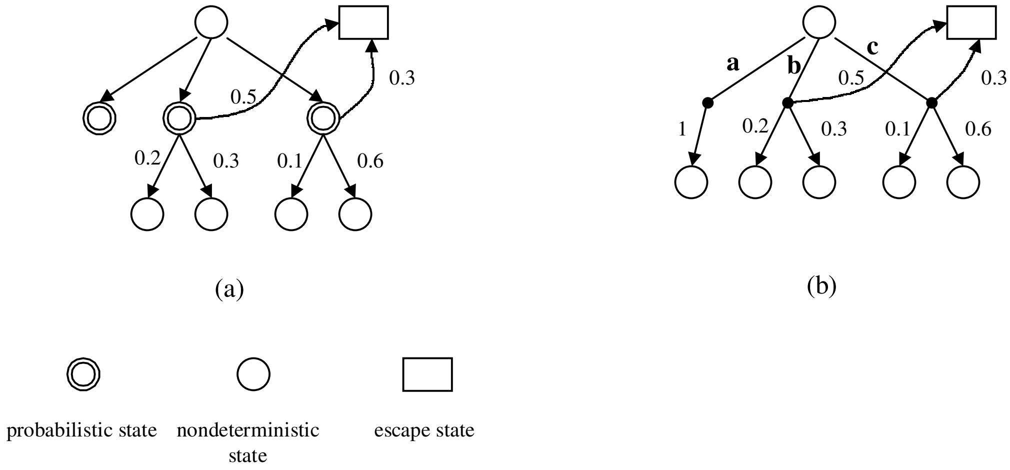


Figure B.3: Converting APSG to MDP

It is preferable to have all APSG acceptance and final states represented explicitly as MDP states, so that we can reason about failing scenarios in the MDP context. For this reason, we add a hidden nondeterministic state and a transition to it to every probabilistic fail state in the APSG. We omit proofs of equivalence of an APSG before and after this modification.

Figure B.3a shows an example APSG, with the corresponding MDP shown in Figure B.3b. The nondeterministic transitions from the root node in the APSG are represented by the MDP actions **a**, **b**, and **c**. The leftmost leaf in the APSG is a probabilistic final state; in the MDP it is represented by the appended hidden nondeterministic state.

APSG components S_0, S_a, S_f, D play no role in the construction of the MDP; they do, however, play a role in our interpretation of results obtained through MDP algorithms. Finally, we can choose the cost function c depending on the questions we are trying to answer. In Section B.2, we will see how particular cost assignments can help us compute interesting probabilities.

Let $e = s_0^n t_1^n s_1^p t_1^p s_1^n \dots t_n^p s_n^n$ be an execution of APSG G, where s_k^n and t_k^n represent nondeterministic states and transitions, respectively, and s_k^p and t_k^p represent probabilistic states and transitions. Let mdp : *executions of G* $\rightarrow$ *executions of* M_G be a function that takes an execution e of APSG G and strips probabilistic states and transitions, leaving a sequence of nondeterministic states and transitions. That is, if $e = s_0^n t_1^n s_1^p t_1^p s_1^n \dots t_n^p s_n^n$, then $mdp(e) = s_0^n t_1^n s_1^n \dots s_n^n$.

Lemma B.3 *$mdp(e)$ is an execution of the MDP M_G.*

Proof:

$mdp(e)$ is an alternating sequence of states and actions, beginning and ending in a state. Let $0 < k \leq n$. By construction, $\mathcal{P}(s_{k-1}^n t_k^n s_k^n) = \pi(s_k^p)(s_k^n) > 0$, fulfilling all requirements for an MDP execution.

∎

Lemma B.4 *mdp is a 1-to-1 correspondence between executions of G that begin and end in a nondeterministic state and executions of M_G.*

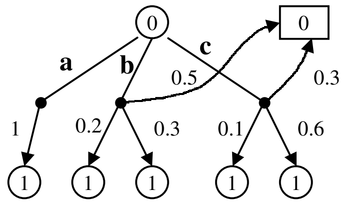


Figure B.4: Initial Value Assignments

Proof:

To prove that mdp is onto, let $e_m = s_0^n t_1^n s_1^n \dots s_n^n$ be an execution of M_G. In G transition t_1^n ends in a probabilistic state. Call this state s_1^p.

Since $s_0^n t_1^n s_1^n$ is a segment of execution e_m, we must have $\mathcal{P}(s_0^n t_1^n s_1^n) > 0$. So in G we have $\pi(s_1^p)(s_1^n) > 0$, and there is a transition t_1^p from s_1^p to s_1^n. We can repeat this process and reconstruct s_k^p and t_k^p for all $0 < k \leq n$. It is easy to see that $e = s_0^n t_1^n s_1^p t_1^p s_1^n \dots t_n^p s_n^n$ is an execution of G and $e_m = mdp(e)$.

To prove that mdp is 1-to-1, let e and e' be two successful executions of G that begin and end in a nondeterministic state, such that $mdp(e) = mdp(e')$. We prove that $e = e'$ by induction on the length l of e and e'.

When $l = 1$, we have $e = mdp(e) = s_0$. Since $mdp(e') = mdp(e)$, we must also have $e' = s_0$.

When $l = k$ ($l > 1$), we have $e = e_{k-1}x_k$ and $e' = e_{k-1'}x'_k$. By inductive hypothesis $e_{k-1} = e'_{k-1}$. There are three cases to consider:

Case 1 x_k is a nondeterministic state or a nondeterministic transition. Since $mdp(x_k) = x_k$ and $mdp(x'_k) = mdp(x_k)$, we must have $x_k = x'_k$.

Case 2 x_k is a probabilistic state. x_k is uniquely determined by nondeterministic transition t_k^p that immediately precedes it. Since by inductive hypothesis $t_k^p = t_{k'}^p$, we must have $x_k = x'_k$.

Case 3 x_k is a probabilistic transition. By inductive hypothesis both x_k and x'_k start in the same probabilistic state. But they cannot end in different nondeterministic states, since that would violate the assumption $mdp(e) = mdp(e')$. So we must have $x_k = x'_k$.

$\square$

B.2 Probabilistic Analysis

Static analysis techniques discussed in Chapter 9 take a binary (yes/no) approach to assessing defense effectiveness. In this appendix we sketch a quantitative, probabilistic analysis that relies on empirical information about attack frequencies and detection probabilities. The analysis uses probabilistic scenario graph ideas presented in Appendix B.

When empirical information about the likelihood of certain events in the system is available, we can use well-known graph algorithms to answer quantitative questions about an attack graph. Suppose we know the probabilities of some transitions in the attack graph. After appropriately annotating the graph with these probabilities, we interpret it as a Markov Decision Process, as discussed in Section B.1.1.

The standard MDP *value iteration algorithm* [75] computes the optimal policy for selecting actions in an MDP that results in maximum benefit (or minimum cost) for the decision-maker. Value iteration can compute the worst case probability of reaching a fail state in a probabilistic scenario graph as follows. We assume that the decision-maker's goal is to maximize the probability of reaching a fail state, and assign all fail states the benefit value of 1. All other nodes get the benefit value of 0 (see Figure B.4 for an example). Then we run the value iteration algorithm. The algorithm finds the optimal transition selection policy for the decision-maker and assigns the expected benefit value resulting from that policy to each state in the scenario graph. The expected value is a fraction of 1, and it is equivalent to the probability of getting to a fail state from that node, assuming the decision-maker always follows the optimal policy. In the example in Figure B.4 the expected benefit at the initial (top) state will come out to be 1, as we can simply take action a, which always leads to a bad state.

We implemented the value iteration algorithm as a scenario graph post-processor and ran it on a modified version of the example from Chapter 10. In the modified example each action has both detectable and stealthy variants. The intruder chooses which action to try next, and he has a certain probability of picking a stealthy or a detectable variant.

In this setup, the value iteration algorithm computes the probability that the intruder will succeed without raising an alarm, and his best attack strategy. The system administrator can use this technique to evaluate effectiveness of various security fixes. For instance, it is possible to compute the probability of intruder success after installing an additional IDS component to monitor the network traffic between hosts `Windows` and `Linux` and after installing a host-based IDS on host `Linux`. Looking at the computed probability in each case gives an indication of which option is more effective.

It is worthwhile to point out that the probabilistic reliability analysis discussed here is useful only for exhaustive and succinct scenario graphs. In the absence of either property the results lose meaning. If the graph is not succinct, it has spurious states which invalidate the analysis methodology itself; if the graph is not exhaustive, the analysis will miss viable executions that violate the property under consideration and thus underestimate the danger.

Appendix C

Converting LTL to Büchi Automata

Traditional automata-theoretic model checkers convert correctness properties specified in temporal logic into automata that accept infinite sequences only. In this appendix, we sketch a modification to the classic algorithm for converting LTL formulas [39] to Büchi automata. The modified algorithm outputs Büchi automata that accept both finite and infinite executions.

C.1 LTL to Büchi Conversion Algorithm

An LTL formula P induces language $\mathcal{L}(P)$ of propositional sequences over the alphabet $A = 2^{AP}$:

$$\mathcal{L}(P) = \{\xi \in A^\omega \cup A^* \mid \xi \models P\}$$

Given P, our goal is to construct a Büchi automaton N_P such that $\mathcal{L}(N_P) = \mathcal{L}(P)$. The basic algorithm for converting LTL formulas to Büchi automata is due to Gerth, Peled, Vardi, and Wolper [39]. They define LTL semantics and generate Büchi automata with respect to infinite executions only. We modify the algorithm slightly so it works for both finite and infinite executions. Except for the changes related to finite executions, our presentation follows [39].

The algorithm builds a graph that defines the states and transitions of the output automaton. The nodes of the graph are labeled by sets of sub-formulas of the input formula. The algorithm constructs the nodes by decomposing formulas according to their Boolean structure and by expanding the temporal operators in order to separate what has to be true immediately from what has to be true in the next state and thereafter. The fundamental identity used for expanding temporal operators is

$$\mu \,\mathbf{U}\, \psi \equiv \psi \vee (\mu \wedge \mathbf{X}(\mu \,\mathbf{U}\, \psi))$$

Before describing the graph construction algorithm, we introduce the data structure used to represent the graph nodes. Throughout the description we refer to the LTL formula P that is being converted to a Büchi automaton.

ID is the unique identifier for the node.

Incoming is the list of predecessor nodes.

New is a list of sub-formulas of P. It represents the set of temporal formulas that must hold at the current state and have not yet been processed by the algorithm.

Old is the set of sub-formulas of P that must hold in the node and have already been processed by the algorithm. As the algorithm processes the sub-formulas that must hold in the node, it moves them from *New* to *Old*, eventually leaving *New* empty.

Next is the set of temporal formulas that must hold in all states that are immediate successors of the node.

Father is the field that will contain the name of the node from which the current one has been split. We use this field for reasoning about the correctness of the algorithm, and it is not important for the construction.

We keep a list of nodes *Nodes_Set* whose construction was completed. We denote the field *New* of the node q by *New*(q), etc.

The fields *Old*, *New*, and *Next* describe temporal properties of execution suffixes that start in the current node. The sub-formulas in these fields contain information about a suffix α^i of a computation α when the following condition holds:

Invariant C.1 *α^i satisfies all of the sub-formulas in either* Old *or* New*, and α^{i+1} satisfies all of the sub-formulas in* Next*.*

Without loss of generality, we assume that the formula P has been converted into negation normal form: the formula consists of temporal operators **U**, **R**, and **X**, and propositional operators $\vee$, $\wedge$, and $\neg$. In negation normal form, the operator $\neg$ is only applied to propositional variables.

The line numbers in the following descriptions refer to the algorithm that appears in pseudocode in Figure C.1. The algorithm starts with a single node with the formula P in its *New* field and a special node *init* in its *Incoming* set (lines 2-3). By the end of the construction, a node will be initial in the Büchi automaton if and only if it contains the label *init* in its *Incoming* set.

The algorithm processes a node q by checking if there are unprocessed sub-formulas left in the set *New*(q) (line 13). If not, the node q is fully processed and ready to be added to the list of nodes *Nodes_Set*. If there is already a node in *Nodes_Set* with the same sub-formulas in both its *Old* and *Next* fields (line 14), the algorithm merely adds the incoming edges of q to the copy that already exists (line 15).

If no node equivalent to q already exists in *Nodes_Set*, then the algorithm adds q to *Nodes_Set* and forms a new successor r to q as follows (lines 17-18):

- There is initially one edge from q to the new node r.
- The set *New* of r is initially set to the *Next* field of q.
- The sets *Old* and *Next* of r are initially empty.

While list *New*(q) is not empty, the algorithm picks a formula η out of *New*(q) (line 19) and either splits or modifies the node q according to the main operator of η.

Input: P – an LTL formula
Output: Generalized Büchi automaton $N_P = (S, \tau, S_0, \mathcal{S}_a, S_f, D)$
datatype graph_node = [*Name*: string, *Father*: string, *Incoming*: set of string,
New: set of formula, *Old*: set of formula, *Next*: set of formula];

Algorithm: LTLtoBüchi(P)
Nodes $\leftarrow$ expand([*Name* $\Leftarrow$ *Father* $\Leftarrow$ NewName(), *Incoming* $\Leftarrow \{init\}$,
New $\Leftarrow \{P\}$, *Old* $\Leftarrow \oslash$, *Next* $\Leftarrow \oslash$], $\oslash$));
$S \leftarrow$ *Nodes*;
$\tau \leftarrow \{(q, q') \mid q \in Incoming(q')\}$;
$S_0 \leftarrow \{q \mid init \in Incoming(q)\}$;
for each subformula $\eta = \mu \, \mathbf{U} \, \psi$ of P, let $S_a^\eta = \{q \mid \psi \in Old(q) \ \vee \ \mu \, \mathbf{U} \, \psi \notin Old(q)\}$;
$\mathcal{S}_a \leftarrow \{S_a^\eta \mid \eta$ is a subformula of the form $\mu \, \mathbf{U} \, \psi\}$;
$S_f \leftarrow \{q \mid$ for each subformula $\mu \, \mathbf{U} \, \psi$ of P, either $\psi \in Old(q)$ or $\mu \, \mathbf{U} \, \psi \notin Old(q)\}$;
$D \leftarrow \lambda q \in S \, . \, \{X \subseteq AP \mid (X \supseteq Old(q) \cap AP) \wedge (X \cap \{\eta \in AP \mid \neg\eta \in Old(q)\} = \oslash)\}$;
return($N_P \leftarrow (S, \tau, S_0, \mathcal{S}_a, S_f, D)$);

function expand(q, *Nodes_Set*)
if $New(q) = \oslash$ **then**
if $\exists r \in$ *Nodes_Set* such that $Old(r)$=$Old(q) \wedge Next(r)$=$Next(q)$ **then**
$Incoming(r) \leftarrow$ Incoming$(r) \cup Incoming(q)$;
return(*Nodes_Set*);
else return(expand([*Name* $\Leftarrow$ *Father* $\Leftarrow$ NewName(), *Incoming* $\Leftarrow \{Name(q)\}$,
New $\Leftarrow \{Next(q)\}$, *Old* $\Leftarrow \oslash$, *Next* $\Leftarrow \oslash$], $\{q\} \cup$*Nodes_Set*));
else let $\eta \in New(q)$;
$New(q) \leftarrow New(q) \setminus \{\eta\}$;
if $(\eta = A) \ \vee \ (\eta = \neg A) \ \vee \ (\eta = \mathtt{T}) \ \vee \ (\eta = \mathtt{F})$ **then**
if $(\eta = \mathtt{F}) \ \vee \ (\mathtt{Neg}(\eta) \in Old(q))$ **then** *– q contains a contradiction*
return(*Nodes_Set*); *– discard q*
else $Old(q) \leftarrow Old(q) \cup \{\eta\}$;
return(expand(q, *Nodes_Set*));
elseif $(\eta = \mu \, \mathbf{U} \, \psi) \ \vee \ (\eta = \mu \, \mathbf{R} \, \psi) \ \vee \ (\eta = \mu \ \vee \ \psi)$ **then**
$r_1 \leftarrow$ [*Name* $\Leftarrow$ NewName(), *Father* $\Leftarrow Name(q)$, *Incoming* $\Leftarrow Incoming(q)$,
New $\Leftarrow New(q) \cup (\{\mathtt{New1}(\eta)\} \setminus Old(q))$,
Old $\Leftarrow Old(q) \cup \{\eta\}$, *Next* $\Leftarrow Next(q) \cup \{\mathtt{Next1}(\eta)\}$];
$r_2 \leftarrow$ [*Name* $\Leftarrow$ NewName(), *Father* $\Leftarrow Name(q)$, *Incoming* $\Leftarrow Incoming(q)$,
New $\Leftarrow New(q) \cup (\{\mathtt{New2}(\eta)\} \setminus Old(q))$,
Old $\Leftarrow Old(q) \cup \{\eta\}$, *Next* $\Leftarrow Next(q)$];
return(expand(r_2, expand(r_1, *Nodes_Set*)));
elseif $\eta = \mu \ \wedge \ \psi$ **then**
return(expand([*Name* $\Leftarrow Name(q)$, *Father* $\Leftarrow Father(q)$,
Incoming $\Leftarrow Incoming(q)$, *New* $\Leftarrow New(q) \cup (\{\mu, \psi\} \setminus Old(q))$,
Old $\Leftarrow Old(q) \cup \{\eta\}$, *Next* $\Leftarrow Next(q)$], *Nodes_Set*));
elseif $\eta = \mathbf{X} \, \mu$ **then**
return(expand([*Name* $\Leftarrow Name(q)$, *Father* $\Leftarrow Father(q)$,
Incoming $\Leftarrow Incoming(q)$, *New* $\Leftarrow New(q)$,
Old $\Leftarrow Old(q) \cup \{\eta\}$, *Next* $\Leftarrow Next(q) \cup \{\eta\}$], *Nodes_Set*));
end expand;

Figure C.1: Converting LTL formulas to Büchi Automata

- η is a proposition p, the negation of a proposition $\neg p$, or a boolean constant `T` or `F`. If η is `F` or $\neg\eta$ is in *Old*, we discard the node since it contains a contradiction (lines 22-23). Otherwise, the node is modified by placing η into *Old* if not already there.

  When η is not a proposition, the node q can be split into two (lines 26-33) or modified in-place (lines 34-41). The exact actions depend on the form on η:

- $\eta = \mu\,\mathbf{U}\,\psi$. Because $\mu\,\mathbf{U}\,\psi$ is equivalent to $\psi \vee (\mu \wedge \mathbf{X}(\mu\,\mathbf{U}\,\psi))$, the node q is split into two nodes r_1 and r_2. In node r_1, μ is added to *New* and $\mu\,\mathbf{U}\,\psi$ to *Next*. In node r_2, ψ is added to *New*.

- $\eta = \mu\,\mathbf{R}\,\psi$. $\mu\,\mathbf{R}\,\psi$ is equivalent to $\psi \wedge (\mu \vee \mathbf{X}(\mu\,\mathbf{R}\,\psi))$, which in turn is equivalent to $(\psi \wedge \mu) \vee (\psi \wedge \mathbf{X}(\mu\,\mathbf{R}\,\psi))$. So the node is split into two nodes r_1 and r_2, ψ is added to *New* of both r_1 and r_2, μ is added to *New* of r_1, and $\mu\,\mathbf{R}\,\psi$ is added to *Next* of r_2.

- $\eta = \mu \vee \psi$. The node is split, with μ added to *New* of r_1 and ψ added to *New* of r_2.

- $\eta = \mu \wedge \psi$. The node is modified, with μ and ψ both added to *New*.

- $\eta = \mathbf{X}\,\mu$. The node is modified, with μ added to *Next*.

In the algorithm listing, the function `NewName()` generates a new string for each successive call. The symbol A stands for a proposition in AP. The function $\texttt{Neg}(\eta)$ is defined as follows: $\texttt{Neg}(A) = \neg A$, $\texttt{Neg}(\neg A) = A$, $\texttt{Neg}(\texttt{T}) = \texttt{F}$, $\texttt{Neg}(\texttt{F}) = \texttt{T}$. The functions $\texttt{New1}(\eta)$, $\texttt{New2}(\eta)$, $\texttt{Next1}(\eta)$ are defined in the following table:

η	$\texttt{New1}(\eta)$	$\texttt{Next1}(\eta)$	$\texttt{New2}(\eta)$
$\mu\,\mathbf{U}\,\psi$	$\{\mu\}$	$\{\mu\,\mathbf{U}\,\psi\}$	$\{\psi\}$
$\mu\,\mathbf{R}\,\psi$	$\{\psi\}$	$\{\mu\,\mathbf{R}\,\psi\}$	$\{\mu, \psi\}$
$\mu \vee \psi$	$\{\mu\}$	$\oslash$	$\{\psi\}$

The list of nodes constructed by the above procedure can be converted to a generalized Büchi automaton with the following components:

- The alphabet is 2^{AP}.

- The set of states S is the set nodes generated by the algorithm (line 4).

- $(q, q') \in \tau$ if and only if $q \in \mathit{Incoming}(q')$ (line 5).

- The initial states are those that have *init* in their incoming set (line 6).

- The acceptance set $\mathcal{S}_a$ (lines 7-8) contains a separate set of states S_a^i for each subformula of the form $\mu\,\mathbf{U}\,\psi$; S_a^i contains all the states q such that either $\psi \in \mathit{Old}(q)$ or $\mu\,\mathbf{U}\,\psi \notin \mathit{Old}(q)$. Thus, S_a^i guarantees that if $\mu\,\mathbf{U}\,\psi$ holds at some state in some accepting run, then ψ must hold later on the same run.

- The set of final states S_f contains all states that do not have unfulfilled eventuality obligations (line 9). Specifically, $q \in S_f$ if and only if for each subformula of P of the form $\mu\,\mathbf{U}\,\psi$, we have $\psi \in \mathit{Old}(q)$ or $\mu\,\mathbf{U}\,\psi \notin \mathit{Old}(q)$.

C.2 Correctness of LTLToBüchi

In this section we sketch elements of a correctness proof proof for the `LTLToBüchi` algorithm. The proof is incomplete; in particular, Lemma C.9 has not been proven for finite sequences.

Let $\Delta(q)$ denote the value of *Old*(q) at the point where the construction of the node q is finished, i.e. where q is added to *Nodes_Set*, at line 25 of the algorithm. Let $\bigwedge \Psi$ denote the conjunction of a set of formulas Ψ, where the conjunction of an empty set is equal to `T`. Let N_P be the output generalized Büchi automaton of `LTLToBüchi`.

Recall that a *propositional sequence* $\xi = x_0x_1\ldots$ is a sequence in $(2^{AP})^* \ \cup \ (2^{AP})^\omega$. Let $\alpha = q_0q_1\ldots$ be a sequence of states of N_P such that for each $i \geq 0$, $(q_i, q_{i+1}) \in \tau_P$. As before, ξ^i denotes the suffix $x_ix_{i+1}x_{i+2}\ldots$ of the sequence ξ.

Lemma C.1 *Let $\alpha = q_0q_1\ldots$ be a path in N_P, and let $\mu\, \boldsymbol{U}\, \psi \in \Delta(q_0)$. Then one of the following holds:*

1. $\forall\, 0 \leq i < |\alpha| : \mu \in \Delta(q_i)$ *and* $\mu\, \boldsymbol{U}\, \psi \in \Delta(q_i)$ *and* $\psi \notin \Delta(q_i)$.
2. $\exists\, 0 \leq j < |\alpha|$ *s.t.* $\forall\, 0 \leq i < j : \mu \in \Delta(q_i)$ *and* $\mu\, \boldsymbol{U}\, \psi \in \Delta(q_i)$ *and* $\psi \in \Delta(q_j)$.

Proof: Follows by straightforward induction on the length of α.

∎

Lemma C.2 *When a node q is split during the construction in lines 26-32 into two nodes r_1 and r_2, the following LTL identity holds:*

$$(\bigwedge \mathrm{Old}(q)\ \wedge\ \bigwedge \mathrm{New}(q)\ \wedge\ \boldsymbol{X} \bigwedge \mathrm{Next}(q)) \Leftrightarrow$$

$$((\bigwedge \mathrm{Old}(r_1)\ \wedge\ \bigwedge \mathrm{New}(r_1)\ \wedge \boldsymbol{X} \bigwedge \mathrm{Next}(r_1)) \vee (\bigwedge \mathrm{Old}(r_2)\ \wedge\ \bigwedge \mathrm{New}(r_2)\ \wedge \boldsymbol{X} \bigwedge \mathrm{Next}(r_2)))$$

Similarly, when a node q is updated to become a new node q', as in lines 34-41, the following holds:

$$(\bigwedge \mathrm{Old}(q)\ \wedge\ \bigwedge \mathrm{New}(q)\ \wedge\ \boldsymbol{X} \bigwedge \mathrm{Next}(q)) \Leftrightarrow (\bigwedge \mathrm{Old}(q')\ \wedge\ \bigwedge \mathrm{New}(q')\ \wedge\ \boldsymbol{X} \bigwedge \mathrm{Next}(q'))$$

Proof: Follows directly from inspecting the cases at lines 26-41 the algorithm and the definition of LTL. ∎

Using the field *Father* we can link each node to the one from which it was split. Thereby we define an ancestor relation R, where $(p, q) \in R$ iff *Father*$(q) =$ *Name*(p). Let R^* be the transitive closure of R. We call a node p *rooted* if $(p, p) \in R$. A rooted node can arise in one of the following two circumstances:

1. p is the initial node with which the search starts at lines 2-3. Then, p has *New*$(p) = \{P\}$.
2. p is obtained at lines 17-18 from some other node q whose construction is finished. Then, *New*(p) is set to *New*(q).

Lemma C.3 *Let p be a rooted node, and $q_1, q_2, \ldots, q_n$ be all of p's direct descendants, so that $(p, q_k) \in R$ for all $1 \leq k \leq n$. Let Ψ be the set in formulas in* New(p) *when p is first created. Let* Next(q_k) *be the values of the* Next *field for q_k at the end of the construction. Then, the following LTL identity holds:*

$$\bigwedge \Psi \Leftrightarrow \bigvee_{1\leq k\leq n} (\bigwedge \Delta(q_k) \ \wedge \ \boldsymbol{X} \bigwedge \text{Next}(q_k))$$

Moreover, if $\xi \models \bigvee_{1\leq k\leq n}(\bigwedge \Delta(q_k) \ \wedge \ \boldsymbol{X} \bigwedge \text{Next}(q_k))$, then there exists some $1 \leq k \leq n$ such that $\xi \models \bigwedge \Delta(q_k) \ \wedge \ \boldsymbol{X} \bigwedge \text{Next}(q_k)$ and for each $\mu \, \boldsymbol{U} \, \psi \in \Delta(q_k)$ with $\xi \models \psi$, ψ is also in $\Delta(q_k)$.

Proof: By induction on the construction, using Lemma C.2 ∎

Lemma C.4 *Let q be a node and ξ a propositional sequence such that $\xi \models \bigwedge \Delta(q) \wedge \boldsymbol{X} \bigwedge \text{Next}(q)$. Let $\Gamma = \{\psi \mid \mu \, \boldsymbol{U} \, \psi \in \Delta(q) \ \wedge \ \psi \notin \Delta(q) \ \wedge \ \xi^1 \models \psi\}$. Then there exists a transition (q, q') such that $\xi^1 \models \bigwedge \Delta(q') \ \wedge \ \boldsymbol{X} \bigwedge \text{Next}(q')$ and $\Gamma \subseteq \Delta(q')$.*

Proof:

When the construction of node q is finished, the algorithm generates a node r with $\mathit{New}(r) = \mathit{Next}(q) = \Psi$ (line 10). Lemma C.3 then guarantees the existence of the required successor.

∎

Lemma C.5 *For every initial state $q \in S_0$ of the automaton N_P, we have $P \in \Delta(q)$.*

Proof: Follows immediately by construction. ∎

Lemma C.6 *For an automaton N_P constructed from property P, the following holds:*

$$P \Leftrightarrow \bigvee_{q\in S_0} (\bigwedge \Delta(q) \ \wedge \ \boldsymbol{X} \bigwedge \text{Next}(q)).$$

Proof: From Lemma C.3, since Ψ in that lemma is initially $\{P\}$. ∎

Lemma C.7 *Let $\alpha = q_0 q_1 q_2 \ldots$ be an execution of M_P that accepts the propositional sequence ξ. Then $\xi \models \bigwedge \Delta(s_0)$.*

Proof:

By induction on the size of the formulas in $\Delta(s_0)$. The base case if for formulas of the form A, $\neg A$, where $A \in AP$. We will show only the case $\mu \, \mathbf{U} \, \psi \in \Delta(q_0)$; the other cases are treated similarly. According to Lemma C.1 there are two cases:

1. $\forall\, 0 \leq i < |\alpha| : \mu \in \Delta(q_i)$ and $\mu \, \mathbf{U} \, \psi \in \Delta(q_i)$ and $\psi \notin \Delta(q_i)$.
2. $\exists\, 0 \leq j < |\alpha|$ s.t. $\forall\, 0 \leq i < j : \mu \in \Delta(q_i)$ and $\mu \, \mathbf{U} \, \psi \in \Delta(q_i)$ and $\psi \in \Delta(q_j)$.

If α is finite, its terminal state satisfies the final state conditions of N_P constructed at line 9. Those conditions rule out case 1. If α is infinite, it satisfies the acceptance conditions of N_P constructed at lines 7-8. Those conditions also rule out case 1. Therefore, only case 2 is possible. Then, by the induction hypothesis $\xi \models \psi$ and for each $0 \leq i < j$, $\xi^i \models \mu$. Thus, by the semantic definition of LTL, $\xi \models \mu \, \mathbf{U} \, \psi$. ∎

Lemma C.8 *Let $\alpha = q_0q_1q_2\ldots$ be an execution of the automaton N_P, constructed for P, that accepts the propositional sequence ξ. Then $\xi \models P$.*

Proof: The node q_0 is an initial state. From Lemma C.7 it follows that $\xi \models \bigwedge \Delta(s_0)$. By Lemma C.5 we also have $P \in \Delta(s_0)$. Therefore, $\xi \models P$. ∎

Lemma C.9 *Let $\xi \models P$. Then there exists an execution α of N_P that accepts ξ.*

Proof:

By Lemma C.6 there exists a node $q_0 \in S_0$ such that $\xi \models \bigwedge \Delta(q_0) \wedge \mathbf{X} \bigwedge \mathit{Next}(q_0)$. We construct the desired execution $\alpha = q_0q_1q_2\ldots$ by repeated application of Lemma C.4, starting at node q_0. If $\xi^i \models \bigwedge \Delta(q_i) \wedge \mathbf{X} \bigwedge \mathit{Next}(q_i)$, then choose q_{i+1} according to Lemma C.4 as a successor of q_i that satisfies two conditions:

(C1) $\xi^{i+1} \models \bigwedge \Delta(q_{i+1}) \wedge \mathbf{X} \bigwedge \mathit{Next}(q_{i+1})$, and

(C2) for any subformula $\mu \,\mathbf{U}\, \psi \in \Delta(q_i)$, if $\psi \notin \Delta(q_i)$ and $\xi^{i+1} \models \psi$, then $\psi \in \Delta(q_{i+1})$.

This procedure gives an inductive definition for α. At each step of this procedure we pick a state q_{i+1} that (by condition 1) satisfies the requirements of Lemma C.4, so the definition of α is well-formed. It remains to show that α is accepted by N_P.

Suppose that ξ and α are infinite. N_P accepts α if at least one member of each set of acceptance states computed at lines 7-8 occurs infinitely often in α. So let $\eta = \mu \,\mathbf{U}\, \psi$ be a subformula of P, and let $S_a^\eta = \{q \mid \psi \in \Delta(q) \vee \mu \,\mathbf{U}\, \psi \notin \Delta(q)\}$ be the set of acceptance states associated with η on line 7. Further, let q_i be any state in α. It is sufficient to show that there exists $j \geq i$ such that $q_j \in S_a^\eta$.

If $\psi \in \Delta(q_i)$ or $\mu \,\mathbf{U}\, \psi \notin \Delta(q_i)$, then $q_i \in S_a^\eta$, so assume that $\psi \notin \Delta(q_i)$ and $\mu \,\mathbf{U}\, \psi \in \Delta(q_i)$. From condition (C1) we know that $\xi^i \models \bigwedge \Delta(q_i) \wedge \mathbf{X} \bigwedge \mathit{Next}(q_i)$, so $\xi^i \models \mu \,\mathbf{U}\, \psi$. Therefore, there must be some minimal $j \geq i$ such that

(A1) $\xi^j \models \psi$

By Lemma C.1 we know that $\mu \,\mathbf{U}\, \psi$ will propagate to successors of q_i up to and including q_{j-1}, so we have

(A2) $\mu \,\mathbf{U}\, \psi \in \Delta(q_{j-1})$

Now, if $\psi \in \Delta(q_{j-1})$ then $q_{j-1} \in S_a^\eta$ as desired, so assume that

(A3) $\psi \notin \Delta(q_{j-1})$

Assertions (A1), (A2), and (A3) together satisfy the requirements of condition (C2), so we get $\psi \in \Delta(q_j)$ and therefore $q_j \in S_a^\eta$, as desired.

∎

Theorem C.1 *The automaton N_P constructed for a property P by algorithm `LTLToBüchi` accepts exactly the executions over $(2^{AP})^* \cup (2^{AP})^\omega$ that satisfy P.*

Proof: Follows directly from Lemmas C.8 and C.9. ∎

Bibliography

[1] Computer Oracle and Password System (COPS). ftp.cert.org/pub/tools/cops.

[2] S. Aggarwal, R. P. Kurshan, and K. Sabnani. A calculus for protocol specification and validation. In H. Rudin and C. H. West, editors, *Protocol Specification, Testing and Verification*, pages 19–34. North Holland, 1983.

[3] Eitan Altman. *Constrained Markov Decision Processes*. Chapman & Hall/CRC, 1999.

[4] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *9th ACM Conference on Computer and Communications Security*, pages 217–224, 2002.

[5] Network Associates. Cybercop scanner. www.nai.com/asp_set/products/tns/ccscanner_intro.asp.

[6] G. Ausiello, A. D'Atri, and M. Protasi. Structure preserving reductions among convex optimization problems. *Journal of Computational System Sciences*, 21:136–153, 1980.

[7] R. Baldwin. Kuang: Rule based security checking. Technical report, MIT Lab for Computer Science, May 1994.

[8] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[9] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. On actl formulas having deterministic counterexamples, 1999.

[10] J. R. Buchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Method, and Philosophy of Sciences*, pages 1–12. Stanford University Press, 1962.

[11] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[12] J. R. Burch, E. M. Clarke, D. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transations on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, April 1994.

[13] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the 1991 International Conference on Very Large Scale Integration*, August 1991.

[14] Shawn Butler. *Security Attribute Evaluation Method*. PhD thesis, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, May 2003.

[15] T. Cattel. Modelization and verification of a multiprocessor realtime os kernel. In *Proceedings of the 7th FORTE Conference*, pages 35–51, Bern, Switzerland, 1994.

[16] J. Chaves. Formal methods at AT&T, an industrial usage report. In *Proceedings of the 4th FORTE Conference*, pages 83–90, Sydney, Australia, 1991.

[17] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*, January 1983.

[18] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[19] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 2000.

[20] Edmund M Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *IEEE Symposium on Logic in Computer Science (LICS) 2002*, page 11, July 2002.

[21] R. W. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In J. Sifakis, editor, *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 24–37. Springer-Verlag, 1989.

[22] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1985.

[23] Cotse.net. Vulnerability Scanners. *http://www.cotse.com/tools/vuln.htm*.

[24] O. Coudert, C. Berhet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the 1989 International Workshop on Automatic Vefification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[25] Frederic Cuppens and Alexandre Miege. Alert correlation in a cooperative intrusion detection framework. In 23^{rd} *IEEE Symposium on Security and Privacy*, pages 187–200, May 2002.

[26] Frederic Cuppens and Rodolphe Ortalo. Lambda: A language to model a database for detection of attacks. In *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID)*, number 1907 in LNCS, pages 197–216. Springer-Verlag, 2000.

[27] M. Dacier. *Towards Quantitative Evaluation of Computer Security*. PhD thesis, Institut National Polytechnique de Toulouse, December 1994.

[28] M. Dacier, Y. Deswartes, and M. Kaaniche. Quantitative assessment of operational security models and tools. Technical Report Research Report 96493, LAAS, May 1996.

[29] J. Dawkins, C. Campbell, and J. Hale. Modeling network attacks: Extending the attack tree paradigm. In *Workshop on Statistical and Machine Learning Techniques in Computer Intrusion Detection*. Johns Hopkins University, June 2002.

[30] Renaud Deraison. Nessus Scanner. *http://www.nessus.org*.

[31] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32nd Design Automation Conference (DAC 95)*, pages 427–432, San Francisco, CA, USA, 1995.

[32] E.M.Clarke and E.A.Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1981.

[33] E. A. Emerson. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs*. PhD thesis, Harvard University, 1981.

[34] E. A. Emerson and C. L. Lei. Temporal model checking under generalized fainess constraints. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, 1985.

[35] E. A. Emerson and C.L. Lei. Modalities for model checking: Branching time strikes back. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM Press, January 1985.

[36] M. Fujita, H. Tanaka, and T. Moto-oka. Logic design assistance with temporal logic. In *Proceedings of the IFIP WG10.2 International Conference on Hardware Description Languages and their Applications*, 1985.

[37] Drew Fundenberg and Jean Tirole. *Game Theory*. MIT Press, 1991.

[38] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM, 1980.

[39] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[40] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the 6th Symposium on Logic in Computer Sciece*, pages 406–415, Amsterdam, July 1991.

[41] R. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 1996 Workshop on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer-Verlag, 1996.

[42] Z. Har'El and R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, Jan.-Feb. 1990.

[43] R. Hojati, R. K. Brayton, and R. P. Kurshan. Bdd-based debugging of designs using language containment and fair ctl. In C. Courcoubetis, editor, *Computer Aided Verification: Proc. of the 5th International Conference CAV'93*, pages 41–58. Springer, Berlin, Heidelberg, 1993.

[44] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editions, Englewood Cliffs, NJ, 1991.

[45] G. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proceedings of the 12th International Symposium on Protocol Specification, Testing, and Vefification*, Lake Buena Vista, Florida, June 1992. North-Holland.

[46] G. J. Holzmann. Proving the value of formal methods. In *Proceedings of the 7th FORTE Conference*, pages 385–396, Bern, Switzerland, 1994.

[47] G. J. Holzmann. The theory and practice of a formal method: Newcore. In *Proceedings of the 13th IFIP World Computer Congress*, Hamburg, Germany, 1994.

[48] G. J. Holzmann. An analysis of bitstate hashing. In *Proceedings of the 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314, Warsaw, Poland, 1995. Chapman & Hall.

[49] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Pearson Educational, 1997.

[50] Gerard J. Holzmann. An improved protocol reachability analysis technique. *Software - Practice and Experience*, 18(2):137–161, 1988.

[51] G.J. Holzmann. On limits and possibilities of automated protocol analysis. In H. Rudin and C. West, editors, *Proceedings of the 6th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Zurich, Sw., June 1987.

[52] Somesh Jha, Oleg Sheyner, and Jeannette M. Wing. Minimization and reliability analyses of attack graphs. Technical Report CMU-CS-02-109, Carnegie Mellon University, February 2002.

[53] Somesh Jha and Jeannette Wing. Survivability analysis of networked systems. In *Proceedings of the International Conference on Software Engineering*, Toronto, Canada, May 2001.

[54] Arvind Kannan, Meera Sridhar, and Jeannette Wing. Personal communication, January 2004.

[55] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[56] O. Kupferman and O. Grumberg. Buy one, get one free!!! *Journal of Logic and Computation*, 6(4):523–539, 1996.

[57] R. P. Kurshan. Analysis of discrete event coordination. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1989.

[58] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[59] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[60] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 97–107. ACM Press, 1985.

[61] F.J. Lin. Specification and validation of communications in client/server models. In *Proceedings of the 1994 Int. Conference on Network Protocols ICNP*, pages 108–116, Boston, Mass., 1994.

[62] Kong-Wei Lye and Jeannette Wing. Game strategies in network security. In *Foundations of Computer Security Workshop*, July 2002.

[63] Monika Maidl. The common fragment of CTL and LTL. In *IEEE Symposium on Foundations of Computer Science*, pages 643–652, 2000.

[64] Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. In H. T. Kung, B. Sproull, and G. Steele, editors, *VLSI Systems and Computations*. Computer Science Press, 1981.

[65] K. McMillan. Using unfolidng to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of the 4th Workshop on Computer Aided Verification*, Montreal, June 1992.

[66] D. E. Muller. Infinite sequences and finite machines. In *Proceedings of the 4th IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 3–16, 1963.

[67] Peng Ning, Yun Cui, and Douglas S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 245–254. ACM Press, 2002.

[68] NuSMV. NuSMV: A New Symbolic Model Checker. `http://afrodite.itc.it:1024/~nusmv/`.

[69] 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, September/October 1999.

[70] R. Peikert. ω-regular languages and propositional temporal logic. Technical Report 85-10, ETH, Zurich, 1985.

[71] D. Peled. Combining partial order reductions with on-the-fly model checking. In D. L. Dill, editor, *Proceedings of the 1994 Workshop on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 1994.

[72] C.A. Phillips and L.P. Swiler. A graph-based system for network vulnerability analysis. In *New Security Paradigms Workshop*, pages 71–79, 1998.

[73] C. Pixley. A computational theory and implementation of sequential hardware equivalence. In R. Kurshan and E. Clarke, editors, *Proceedings of CAV Workshop*, Rutgers University, NJ, June 1990.

[74] Amir Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[75] M. Puterman. *Markov Decision Processes*. John Wiley & Sons, New York, NY, 1994.

[76] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[77] C. Ramakrishnan and R. Sekar. Model-based vulnerability analysis of computer systems. In *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation*, September 1998.

[78] John Ramsdell. Frame propagation. MITRE Corp., 2001.

[79] R.W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–165, May 2001.

[80] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2002.

[81] A. Sistla, M. Vardi, and P. Wolper. The complementation problem for buchi automata with applications to temporal logic. *Theoretical Computer Science*, 47:217–237, 1987.

[82] A. P. Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, Harvard University, 1983.

[83] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[84] Petr Slavík. A tight analysis of the greedy algorithm for set cover. In *28th ACM Symposium on Theory of Computing (STOC)*, pages 435–441, 1996.

[85] SMV. SMV: A Symbolic Model Checker. `http://www.cs.cmu.edu/~modelcheck/`.

[86] Peter Stephenson. Using formal methods for forensic analysis of intrusion events - a preliminary examination. White Paper, available at http://www.imfgroup.com/Document Library.html.

[87] U. Stern and D. Dill. Improved probabilistic verification by hash compaction. In *Proceedings of the IFIP WG 10.5 Advanced Research Working Conf. on Correct Hardware Design and Verification Methods*, pages 206–224, 1995.

[88] U. Stern and D. Dill. A new scheme for memory-efficient probabilistic verification. In *Proceedings of the IFIP TC6/WG6.1 Joint Int. Conf. on Formal Description Techn. for Distr. Systems and Comm. Protocols, and Protocol Spec., Testing, and Verification, FORTE/PSTV96*, pages 333–348. North-Holland Publ., 1996.

[89] L.P. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, June 2000.

[90] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[91] Steven Templeton and Karl Levitt. A requires/provides model for computer attacks. In *Proceedings of the New Security Paradigms Workshop*, Cork, Ireland, 2000.

[92] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Rutgers, June 1990.

[93] Moshe Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *First IEEE Symp. on Logic in Computer Science*, pages 322–331, 1986.

[94] G. Vigna and R.A. Kemmerer. Netstat: A network-based intrusion detection system. *Journal of Computer Security*, 7(1), 1999.

[95] G. von Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, C-31(3), March 1982.

[96] Common Vulnerabilities and Exposures. http://www.cve.mitre.org.

[97] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, Tucson, AZ, 1983.

[98] Pierre Wolper and Denis Leroy. Reliable hashing without collision detection. In *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, June 1993.

[99] D. Zerkle and K. Levitt. Netkuang - a multi-host configuration vulnerability checker. In *6th USENIX Unix Security Symposium*, San Jose, CA, 1996.

Index