Strauss: A Specification Miner

By

Glenn Ammons

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

2003

List of Figures

List of Tables

Contents

Chapter 7

Debugging Specifications

This dissertation is about methods for solving the specification problem, or the problem of finding specifications that program verification tools can use to find errors. In previous chapters, I have explained how Strauss attacks this problem by mining specifications from interaction traces: given a few interaction traces that record actual runs of one or more programs (Chapters 2 and 6), Strauss extracts scenarios (Chapters 3 and 4) and infers a scenario acceptor from them (Chapter 5).

Unsurprisingly, Strauss can mine buggy specifications: if some of the interaction traces are erroneous (as often happens), some of the scenarios are also erroneous, and the miner learns a SA that accepts erroneous interacton traces. Clearly, a solution to the specification problem also requires methods for debugging specifications.

Very small and well-understood specifications can be debugged by inspection. A natural way to debug a more difficult specification is by testing it. To test a specification, the specification author uses a program verification tool to check the specification against several programs. The tool finds inconsistencies between the program and the specification and reports them to the author. The author is supposed to look at each inconsistency and decide if the inconsistency is caused by a specification error. If the cause is a specification error, it should be fixed.

In particular, a temporal specification (such as a Strauss specification) can be expressed as a finite automaton that accepts some program execution traces and rejects others. A tool that verifies temporal specifications generates short *violation traces* [1] that appear to occur in the program but are not accepted by the FA. To debug a temporal specification by testing, the specification author looks at each trace and decides whether it demonstrates an error or not. If it is not erroneous, it should be added to the language of the FA.

Although Strauss specifications can be debugged this way, they also can be debugged more directly, without testing. Suppose that Strauss learns a buggy specification because some of the training scenarios are erroneous. To debug the specification without testing it, a specification expert looks at each scenario and decides whether it is erroneous or not. If the scenario is erroneous, then the expert tells the miner to ignore it when inferring a correct specification.

Both debugging methods are tedious and error-prone because a person must inspect many traces (violation traces or scenarios)—Strauss can generate tens of thousands of scenarios, and some program verification tools generate similar numbers of violation traces [1, 26, 6]. This chapter describes a novel method for debugging formal, temporal specifications that allows a person to take all of the traces into consideration without actually inspecting every trace. I have used the method to debug Strauss specifications, but in fact the method applies to any temporal specification.

In my method, an automatic tool finds similarities within a set of traces and uses

[1] Throughout this chapter, I use the word "trace" by itself to mean either violation traces or scenarios, as opposed to program execution traces or the interaction traces defined in Chapter 6.

concept analysis [52] to cluster similar traces together. The user inspects clusters of traces—summarized in various ways—instead of individual traces. For example, instead of looking at thousands of individual traces, a specification author can use my method to look at a few clusters of similar traces. For each cluster, the author views a summary of the cluster—such as a finite automaton that recognizes the cluster's traces—and decides en masse whether the cluster's traces are erroneous or not.

Concept analysis clusters hierarchically, producing a *concept lattice* of small clusters and big clusters, with small clusters contained within big clusters. Moreover, the traces in small clusters are more alike than the traces in big clusters.

Hierarchical clustering is essential. The ideal tool would divide the traces into two clusters: a cluster of erroneous traces and a cluster of correct traces. Unfortunately, this ideal can not be attained. Any real tool can produce *mixed clusters*, which contain both erroneous traces and correct traces. Hierarchical clustering solves this problem: a specification author who is presented with a mixed cluster can choose to look at the smaller clusters within it. These clusters are less likely to be mixed because they are smaller and because the traces within them are more similar.

Hierarchical clustering has benefits beyond handling mixed clusters:

- Small clusters are easier to understand and judge as correct or incorrect than large clusters, but it takes more small clusters than large clusters to cover the entire set of traces. Hierarchical clustering allows the user to choose to examine small clusters, large clusters, or a mixture of both.

- Clusters overlap, so the user can check his decisions by viewing summaries of the intersections and unions of clusters. For example, a specification author who believes he has found a number of erroneous traces can view a summary of all erroneous traces in a particular cluster: the summary should be consistent with his belief.

My method defines the similarity of a set of traces in terms of the transitions of a FA that recognizes traces. I regard traces that execute many transitions in common as more similar than traces that execute fewer transitions in common. This definition is flexible because the FA can be varied and intuitive because the user is debugging a specification which is itself expressed in terms of an FA. The definition also meshes nicely with concept analysis, which clusters objects with discrete attributes. In my case, objects represent traces and attributes represent FA transitions.

To test my method, I implemented a tool, Cable, for debugging specifications and used Cable to debug Strauss specifications. The corrected specifications found 199 bugs in widely distributed X11 programs, including serious race conditions and performance bugs. I found that using Cable to debug every specification requires less than one third as many user decisions as debugging every specification by examining all traces. In one case, using Cable required only 28 decisions, while debugging by examing all specifications required 224. I also found that concept analysis is affordable: it never took longer than about 22 seconds to run.

Contributions

This chapter makes these contributions:

- A novel method for debugging temporal specifications based on hierarchical clustering. The method applies not only to mined specifications, but also to temporal specifications from any source.
- A flexible, intuitive definition of similarity for traces that allows hierarchical clustering via concept analysis.
- A tool, Cable, that helps debug specifications by presenting users with a simple interface for exploiting the results of concept analysis.

Organization of this chapter

The rest of the chapter is organized as follows. Section 7.1 works through two examples, which demonstrate how to debug specifications by examining clusters of traces. Section 7.2 presents concept analysis and shows how to apply it to clustering traces. The Cable tool is described in Section 7.3, as are strategies for using it effectively. Section 7.4 evaluates the usefulness of Cable for debugging specifications mined by Strauss. Section 1.4 discusses related work. Section 7.5 concludes the chapter.

For all trace events matching

```
F = popen() | F = fopen()
```

with STM

```
return popen(def ?)
return fopen(def ?)
fread(use f)
fwrite(use f)
fclose(def use f)
pclose(def use f)
```

a scenario belongs to

```
(F = popen() [seed] | F = fopen() [seed]);
fread(f = F)*; fwrite(f = F)*;
fclose(f = F)
```

Figure 58: An incorrect temporal specification.

7.1 Two examples

This section presents my method for debugging temporal specifications with concept analysis and shows how to apply it by working through two examples. The first example demonstrates debugging with the aid of a a verification tool by testing a specification against a program, while the second example demonstrates debugging a Strauss specification by inspecting the scenarios from which Strauss inferred the specification.

7.1.1 Debugging by testing

Figure 58 shows a prototypical (and buggy) temporal specification. This example attempts to formalize a rule about the C stdio library. In that library, a call to `fopen` opens a file and returns a *file pointer* for reading and writing the file. The

```
0x10 = popen(); fread(f = 0x10); fwrite(f = 0x10); pclose(0x10)
0x20 = popen(); fread(0x20); fread(0x20); pclose(0x20)
0x30 = popen(); pclose(0x30)

0x40 = fopen(); fwrite(0x40)
0x50 = popen(); fread(0x50)
0x60 = fopen()

0x70 = fopen(); fread(0x70); fread(0x70); pclose(0x70)
0x80 = fopen(); fwrite(0x80); fwrite(0x80); pclose(0x80)
0x90 = fopen(); pclose(0x90)
```

Figure 59: Several violation traces that could be reported by verification of the specification in Figure 58.

file pointer should eventually be closed with a call to `fclose`. By contrast, a call to `popen` opens a pipe for communication with another process. Like `fopen`, `popen` returns a file pointer. Unlike `fopen`, the file pointer returned by `popen` should be closed with a call to `pclose`. The specification in Figure 58 gets this wrong: it allows a call to `fclose` on any file pointer, regardless of its source.

Suppose that a specification author is debugging this specification by testing it against a program. The author starts by using a program verification tool to find inconsistencies between the specification and the program. The tool analyzes the program and reports *violation traces*, which are program execution traces that demonstrate an apparent violation of the specification. Traditionally, the author looks at each violation trace, decides why it was reported, and takes an appropriate action. For the specification in Figure 58, the violation traces (see Figure 59 for examples) might include

- Traces that begin with a call to `popen` and end with a call to `pclose`. These traces are correct, so the author should change the specification to accept these traces.

- Traces that begin with a call to `fopen` or with a call to `popen` and end without a call to `fclose` or a call to `pclose`. These traces are erroneous, so the author should not change the specification.

- Traces that begin with a call to `fopen` and end with a call to `pclose`. Again, these traces are erroneous, so the author should not change the specification.

Unfortunately, the verification tool does not summarize the violation traces as neatly as I just did. Instead, the tool lists each trace with all of the calls it makes (not just the relevant calls I picked out in the above list), and in no particular order. For a simple example like the one in Figure 58, it may be easy for the author to inspect the violation traces and understand them well enough to decide how to fix the specification. However, if the violation traces are more complicated, inspecting each trace is both tedious and error-prone. If the tool reports thousands or tens of thousands of complicated violations (as some do [26, 6]), the problem is daunting.

Now let us see how the author would debug this specification with my method. My method has three steps. Step 1 automatically builds a concept lattice that neatly summarizes the violation traces, before the specification author sees them. This step has three substeps:

Step 1a This step finds a small finite automaton (FA) that recognizes the violation traces. For example, the Raman and Patrick's sk-string algorithm (see Section 5.3.2, page 117) could be used. A good FA learning algorithm is not

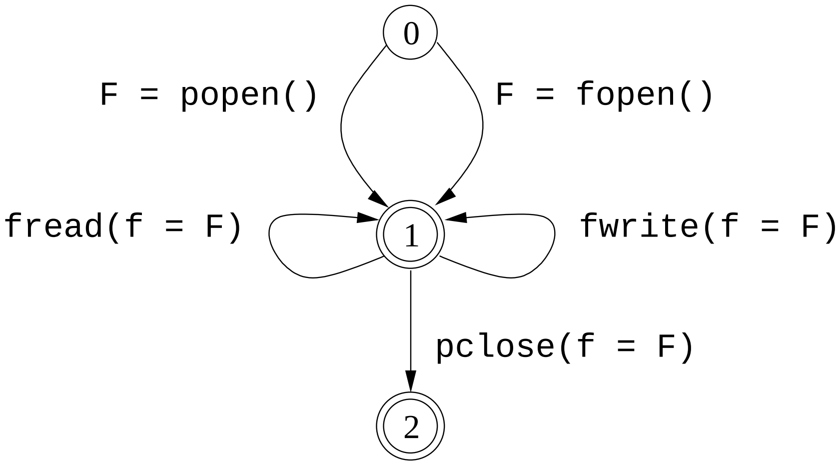

Figure 60: A small finite automaton that recognizes violation traces from verification of the specification in Figure 58.

essential. In fact, I have had success with FAs that recognize *all* possible traces. The key property I require is that erroneous and correct traces take different paths through the FA.

Figure 60 shows a small FA that recognizes violation traces from verification of the specification in Figure 58.

Step 1b This step defines a measure of similarity for violation traces. The FA recognizes a violation trace o iff there is a sequence of transitions $(a_0, \ldots, a_n)$ such that each transition a_i matches the ith event in o, the head of a_0 is a start state, and the tail of a_n is an accepting state. Call $(a_0, \ldots, a_n)$ an *accepting sequence of transitions for* o. If a transition a is on an accepting sequence of transitions for o, I say that o *executes* a. Finally, I define the *similarity* of a set of violation traces as the number of transitions that are executed by every sequence in the set.

Defining similarity with respect to an FA has two benefits. First, by varying

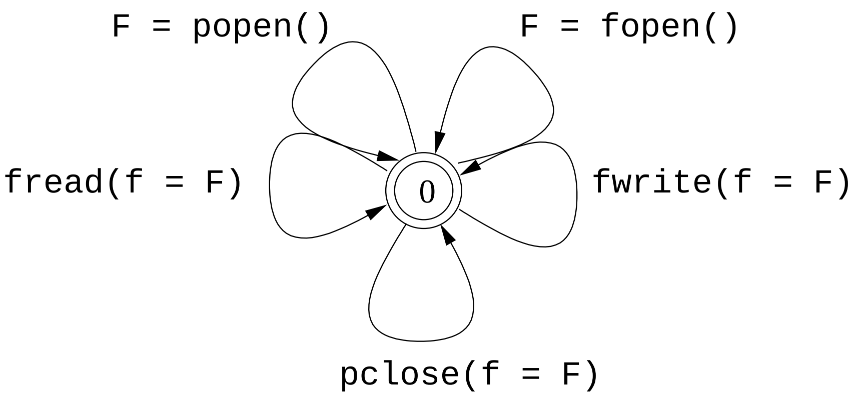

Figure 61: A very small finite automaton that recognizes violation traces from verification of the specification in Figure 58.

parameters of the FA-learning algorithm, the author can choose to use a large FA that makes very fine distinctions among traces or a smaller FA that makes coarser distinctions. For example, the FA in Figure 60 distinguishes between traces that call `popen` before calling `pclose` and traces that call `pclose` before calling `popen`. If the order didn't matter, a very small FA such as the one in Figure 61 could be used to induce a simpler concept lattice. On the other hand, if the order of calls to `fread` and `fwrite` also mattered, then a larger FA could be used to induce a concept lattice that distinguished different orders.

The second benefit is that, since the specification itself is expressed as an FA, summarizing violation traces with FAs makes it easier for the author to see how to fix the specification.

Step 1c This step uses concept analysis to build a *concept lattice*; the nodes of the lattice are called *concepts*. A concept pairs a set of violation traces with a set of FA transitions that are executed by every trace in the set. Concepts at

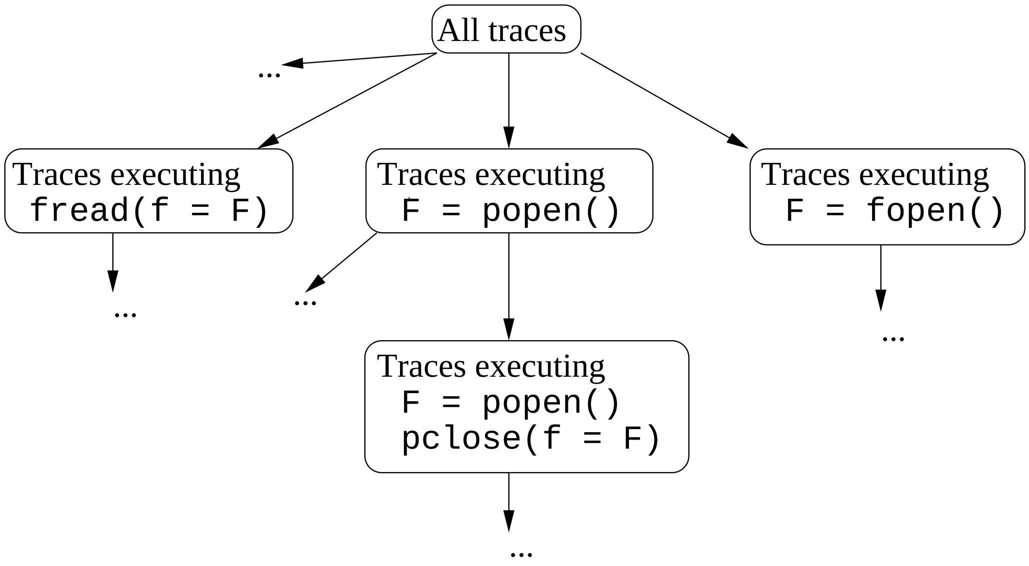

Figure 62: Part of a concept lattice that might be induced by violation traces from verification of the specification in Figure 58, with respect to the FA in Figure 60.

the top of the concept lattice contain more traces but fewer transitions than concepts at the bottom of the lattice. That is, according to my definition of similarity, the sets of traces in concepts get smaller but more similar as one moves down in the lattice.

Figure 62 shows part of a concept lattice that might be induced by violation traces from verification of the specification in Figure 58, with respect to the FA in Figure 60.

The concept lattice is a neat summary of the violation traces. In Step 2 of my method, the specification author uses Cable to display the lattice and to track his decisions about the traces in concepts. Step 2 has two substeps:

Step 2a In this step, the author records his decisions about traces by labeling traces. His goal is to partition the traces into correct traces, which should be accepted by the correct specification, and erroneous traces, which should not be accepted. The former he labels "good", while the latter he labels "bad".

The author is free to inspect concepts in any order, although a mostly top-down approach seems to work best in practice. Section 7.3 suggests several strategies, which are evaluated in Section 7.4.

Suppose that the author first looks at the concept that contains traces that execute `F = popen()`. The author asks Cable to display another finite automaton that is inferred from the traces in that concept. Because this automaton contains both erroneous traces and correct traces, the author decides to look at the concepts immediately below this concept. Suppose that the first concept he looks at contains just traces that execute both `F = popen()` and `pclose(F)`. These traces are correct, so the author labels them as "good". Finally, suppose that the author revisits the concept that contains traces that execute `F = popen()`. He asks Cable to display a finite automaton that is inferred from the unlabeled traces in that concept. These traces execute `F = popen()` but not `pclose(F)`, so they are erroneous. The author labels these traces as "bad". At this point, the author has come to a decision about all of the traces that execute `F = popen()`. The traces that execute `F = fopen()` remain, and the author labels these in a similar fashion.

Step 2b In this step, the author checks his labeling. Once all traces have been labeled, the author views a FA that is inferred from all "good" traces. These traces should be accepted by the correct specification. If the author made

For all trace events matching

```
F = popen() | F = fopen()
```

with STM

```
return popen(def ?)
return fopen(def ?)
fread(use f)
fwrite(use f)
fclose(def use f)
pclose(def use f)
```

a scenario belongs to

```
(F = popen() [seed];
 fread(f = F)*; fwrite(f = F)*; pclose(f = F))
| (F = fopen() [seed]);
   fread(f = F)*; fwrite(f = F)*; fclose(f = F))
```

Figure 63: The result of debugging the specification in Figure 58.

a mistake in his labeling, it will be revealed as the presence or absence of certain traces in the FA's language. Note that if the FA for all "good" traces is too complicated, the author can choose to view a FAs inferred from the "good" traces within concepts below the top of the lattice.

If there is a mistake, the author searches through the lattice for concepts that contain only traces that are incorrectly labeled "good", just as earlier he searched through the lattice for traces that should be labeled "good".

Once the author is satisfied that his labeling is correct, he fixes his specification so that it accepts all "good" traces and continues to reject all "bad" traces:

Step 3 In this step, the author fixes his specification. Note that although the author has not inspected every violation trace, he has taken every violation trace into consideration. Consequently, he can be more confident that he has the right fix for his specification. Figure 63 shows the result of fixing the specification in Figure 58.

To summarize, my method has the following benefits:

- The concept lattice neatly summarizes complicated traces that the verification tool lists in no particular order.
- Defining similarity with respect to an FA is flexible because the FA can be varied and intuitive because the specification itself is expressed in terms of an FA.
- The concept lattice allows the author to take every trace into consideration without inspecting every trace.
- The author can use the lattice to check that he has made the right decision about every trace.

7.1.2 Debugging a Strauss specification

Suppose that Strauss learns the buggy specification in Figure 58 from a set of scenarios that include

- Scenarios that begin with a call to `popen` and end with a call to `fclose`. These scenarios are erroneous, so they should not be included in the correct specification.

- Scenarios that begin with a call to `fopen` and end with a call to `fclose`. These scenarios are correct, and should be included in the correct specification.

An expert can produce a correct specification by rerunning Strauss's learner (Chapter 5) only on the latter of the two kinds of scenarios above. Unfortunately, the scenarios are not summarized so neatly as they are above. In general, it is tedious and error-prone for the expert to inspect every scenario.

My solution is to summarize the scenarios neatly, before the expert sees them. The method is very similar to the method I discussed in Section 7.1.1. The differences are in Steps 1a and 3.

In Step 1a, the expert does not need to infer an acceptor for the traces. He already has one: namely, the scenario acceptor from Strauss's buggy specification. On the other hand, if Straus infers a SA that makes unnecessarily fine distinctions among scenarios, the expert may choose to use a different SA. But, in my experience, the inferred SA is usually a good starting point.

Steps 1b and 1c are just as in Section 7.1.1: the expert uses the SA to define a measure of similarity for scenarios and to build a concept lattice. Step 2 is the same, too: the expert uses Cable to label as "good" the scenarios that belong in the correct specification and to label as "bad" the scenarios that don't belong.

The expert fixes the specification in Step 3. In Section 7.1.1, the specification author did this manually. In Strauss, the expert just runs Strauss's learner on the scenarios that have been labeled "good".

There is a further problem, however. Strauss also *generalizes*: the specification accepts some scenarios that were not in its training set, but are similar to scenarios

```
0x10 = popen(); fread(0x10); fwrite(0x10); pclose(0x10)
0x20 = popen(); fread(0x20); fread(0x20); pclose(0x20)
0x30 = popen(); pclose(0x30)

0x40 = fopen(); fread(0x40); fwrite(0x40); fclose(0x40)
0x40 = fopen(); fread(0x40); fread(0x40); fclose(0x40)
0x40 = fopen(); fclose(0x40)
```

Figure 64: Several scenarios.

in the training set. For example, Strauss, given the "good" scenario traces in Figure 64, would ideally produce an SA that accepts any number of calls to `fread` and `fwrite` between calls to `popen` and `pclose` and between calls to `fopen` and `fclose`. Unfortunately, Strauss can make mistakes: in this case, Strauss might produce an SA that allows a call to `popen` to be followed by a call to `fclose`.

To address this problem, the expert can vary the parameters of Strauss's NFA learner (see Chapter 5), but a more frequently fruitful solution is to further subdivide the training set and apply Strauss separately to each division. If the expert observes that Strauss overgeneralizes, he can redo Step 2 and assign several different kinds of "good" labels. Here, the expert might assign a label "good_fopen" and another label "good_popen".

7.2 Applying concept analysis

Concept analysis [52] is a hierarchical clustering technique for objects with discrete attributes. This section reviews concept analysis and explains how to use it to cluster violation sequences and scenarios with respect to a temporal specification.

	4-legged	hairy	smart	marine	thumbed
cats	yes	yes			
dogs	yes	yes			
dolphins			yes	yes	
gibbons		yes	yes		yes
humans			yes		yes
whales			yes	yes	

Figure 65: A context where the objects are animals and the attributes are adjectives that describe animals.

In the process, I define a natural measure of the similarity of a set of traces and show that concept analysis builds a hierarchy of clusters of traces where small clusters are more similar than the large clusters that contain them. This property allows a user of Cable to choose between labeling many small and highly similar clusters and labeling a few larger but less similar clusters.

7.2.1 Concept analysis

The input to concept analysis is a set O of *objects*, a set A of *attributes*, and a *context* $R \subseteq O \times A$ that relates objects to attributes. Figure 65 shows an example where the objects are animals and the attributes are adjectives that describe animals. [2]

Given O, A, and R, concept analysis finds *concepts*. A concept pairs a set of objects X with a related set of attributes Y: Y is exactly the set of attributes enjoyed by all objects in X, and X is exactly the set of objects that enjoy all of the attributes in Y. In order to define concepts formally, I define two mappings

[2] I took this example from Michael Siff's thesis [44].

$\sigma_R : 2^O \to 2^A$ and $\tau_R : 2^A \to 2^O$. For any $X \subseteq O$ and $Y \subseteq A$,

$$\sigma_R(X) = \{a \in A | \forall x \in X.(x, a) \in R\}$$

$$\tau_R(Y) = \{o \in O | \forall y \in Y.(o, y) \in R\}$$

The formal definition of a concept is as follows: (X, Y) is a concept iff $\sigma_R(X) = Y$ and $\tau_R(Y) = X$. I call X the *extent* of the concept and Y the *intent* of the concept.

c0 = ({cats, dogs, dolphins, gibbons, humans, whales}, {})
c1 = ({cats, dogs, gibbons}, {hairy})
c2 = ({dolphins, gibbons, humans, whales}, {smart})
c3 = ({gibbons, humans}, {smart, thumbed})
c4 = ({cats, dogs}, {4−legged, hairy})
c5 = ({gibbons}, {hairy, smart, thumbed})
c6 = ({dolphins, whales}, {smart, marine})
c7 = ({}, {4−legged, hairy, smart, marine, thumbed})

$$\begin{tikzcd}
 & c0 \arrow[dl] \arrow[dr] & \\
c1 \arrow[dd] \arrow[ddr] & & c2 \arrow[dl] \arrow[dd] \\
 & c3 \arrow[d] & \\
c4 \arrow[dr] & c5 \arrow[d] & c6 \arrow[dl] \\
 & c7 &
\end{tikzcd}$$

Figure 66: Concept lattice for Figure 65.

The choice of O, A, and R uniquely defines a set of concepts. I define a partial order $\leq_R$ on these concepts: $(X_0, Y_0) \leq_R (X_1, Y_1)$ iff $X_0 \subseteq X_1$. This partial order induces a complete lattice on concepts, called the *concept lattice*. Figure 66 shows the concept lattice for the example in Figure 65.

By definition, the concept lattice is a subset lattice on objects. In fact, the concept lattice is also a superset lattice on attributes. That is, $(X_0, Y_0) \leq_R (X_1, Y_1)$ iff $Y_0 \supseteq Y_1$. I can use this fact to define a measure of similarity that increases as one moves down in the concept lattice.

Define the *similarity* of $X \subseteq O$ by $\mathrm{sim}(X) = |\sigma_R(X)|$. That is, the similarity of X is simply the number of attributes shared by all objects in X. Because the concept lattice is a superset lattice on attributes, if (X_0, Y_0) and (X_1, Y_1) are concepts with $X_0 \subseteq X_1$, then $\mathrm{sim}(X_0) \geq \mathrm{sim}(X_1)$.

Efficiency of concept analysis

There are several algorithms for building concept lattices. The algorithm I use is due to Godin and others [23] (I use their Algorithm 1). Let k be an upper bound on $|\sigma_R(\{o\})|$, where $o \in O$. That is, k is an upper bound on the number of attributes enjoyed by any object in O. Then, their algorithm runs in time

$$\mathbf{O}(2^{2k}|O|)$$

In my experiments, k was typically less than ten, while $|O|$ ranged up to the hundreds. My measurements (see Section 7.4) show that the algorithm is very practical, terminating in less than 22 seconds on my largest data set.

7.2.2 Clustering traces

In order to cluster traces with concept analysis, I need to define O, A, and R:

O The objects are the traces themselves.

A I either have in hand or can infer a FA that recognizes the traces. The attributes are this FA's transitions.

R Let AS(o) be the set of all accepting sequences of transitions for the trace o. I define R by

$$R = \{(o, a) \in O \times A | \exists s \in \mathrm{AS}(o).s = (\ldots, a, \ldots)\}$$

That is, $(o, a) \in R$ iff the transition a can be executed while accepting the trace o. R can be computed by simulating each trace on the finite automaton.

This is a natural choice of R. Intuitively, traces that exercise many of the same transitions are more alike than traces that do not. With this choice of R, the similarity of a set of traces $X \subseteq O$ is equal to the number of transitions that the traces exercise in common. Also, R provides a direct connection between traces and the specification that the user is debugging. This connection is useful for answering questions such as "Which parts of the specification matter for these traces?" and "Which traces would be affected by a change to this part of the specification?".

Of course, there are other ways to define similarity with respect to an FA. For example, my definition ignores the order in which transitions are taken. There are two good reasons to ignore the order of transitions. First, the number of possible orders grows exponentially with the amount of history that is tracked. Second, tracking transition ordering is unnecessary: any benefit can be had simply by transforming the FA.

7.3 Cable

The section describes Cable, my tool for debugging specifications. Cable displays the concept lattices defined in Section 7.2.2 and enables a specification author or other expert to view summaries of concepts and to decide en masse whether the traces in a concept are erroneous or not. The rest of this section explains Cable's interface, discusses strategies for using Cable effectively, and explains on which lattices Cable works best.

7.3.1 The Cable interface

Cable, which is based on the Dotty [20] graph visualization tool, displays the concept lattice and allows the user to view summaries of concepts and to decide en masse whether the traces in a concept are erroneous or not.

The user records his decision about a set of traces by *labeling* the traces in the set. His goal is to partition the traces into a set of erroneous traces, labeled "bad", and a set of correct traces, labeled "good".

For example, if a specification author decides that certain violation traces do not demonstrate a program error, he gives those traces the label "good". On the other hand, the author gives violation traces that do demonstrate program errors the label "bad". The author's goal is to label every trace; when he is done, he uses Cable to view the traces labeled "good" and fixes his specification accordingly.

An expert uses Cable to debug a Strauss specification in a similar fashion. If the expert decides that certain scenarios are not erroneous, he labels them "good". Scenarios that are erroneous are labeled "bad". After the expert has labeled every scenario, he uses Cable to view the scenarios labeled “good” and reruns the miner on those scenarios. Labels are a flexible mechanism: as Section 7.1.2 discussed, the expert can avoid problems with overgeneralization by assigning several different kinds of “good” labels and running the miner several times.

A user of Cable can label all of a concept’s traces at once. Because concepts belong to a lattice, labeling the traces in one concept affects the labels on traces in other concepts. Labeling *all* of the traces in a descendant concept also labels *some* of the traces in an ancestor concept, labeling *all* of the traces in an ancestor concept also labels *all* of the traces in a descendant concept, and labeling *all* of the traces in a cousin or sibling concept might label *some* of the traces in another cousin or sibling concept.

Consequently, Cable keeps track of which traces have been labeled, ensures that each trace receives no more than one label, and gives the user visual feedback that makes it obvious which concepts still have unlabeled traces. At any time, each concept in the lattice is in one of three states:

Unlabeled The concept has unlabeled traces, and no traces that are labeled. Cable displays Unlabeled concepts in green.

PartlyLabeled The concept has some unlabeled traces and some labeled traces. Cable displays PartlyLabeled concepts in yellow.

FullyLabeled The concept has no unlabeled traces. Cable displays FullyLabeled concepts in red.

Note that an empty concept (a concept with no traces) is always FullyLabeled.

Cable's "Label traces" command allows the user to assign labels to selected traces in a concept:

Label traces If some traces already have labels, then Cable asks the user which traces to label: the user may choose to label all of the traces, only the unlabeled traces, or only the traces with a given label. Then, Cable prompts the user for a label and gives that label to the selected traces. Because no trace may have more than one label, the new label replaces any existing labels.

If no traces have labels, then Cable prompts the user for a label and gives that label to all of the concept's traces.

A Cable user bases his labeling decisions on summary views of concepts. Cable supports the following summary views:

Show FA Cable uses an FA learner to construct a FA that summarizes a concept's traces and then prints this FA to the terminal. This is the most frequently used summary because the FA is often short and clear. My implementation uses the sk-strings learner (see Section 5.3.2, page 117)

If the concept is Unlabeled, then the FA that Cable prints summarizes all of the concept's traces. If the concept is PartlyLabeled or FullyLabeled, then Cable lets the user choose which traces to include: the user may choose to include all of the traces, only the unlabeled traces, or only the traces with a given label. The last option is particularly useful once all concepts are FullyLabeled: the user can obtain a FA for all traces with a particular label by telling Cable to show the FA for the traces with that label in the concept at the top of the lattice.

Show transitions Cable prints the transitions in the concept. This is the second most frequently used summary because the user often knows that the label for a trace depends on whether the trace executes a certain set of transitions in the reference FA or not.

As above, the user may choose to view the transitions for a subset of the concept's traces if some of the traces have labels.

Show traces Cable prints the traces in the concept. This summary is not used very often because it usually generates more output than the user can understand.

As above, the user may choose to view a subset of the concept's traces if some of the traces have labels.

Finally, the user can choose a new FA and use it to cluster concept traces:

Focus Cable starts a sub-session, which focuses on a single concept's traces. Cable prompts the user for a FA to use for the session, and clusters the traces in the focused session with that FA. The user can end a focused session at any time, at which time any labels that he assigned are automatically merged into the original session.

In my experiments, I always started Cable with a cluster determined by Strauss's inferred SA. If this cluster appeared complicated, I started a focused session. The SAs that I used for focusing followed one of the following three templates:

Unordered SAs that follow this template distinguish among scenarios based on which events appear in scenarios, while completely ignoring the order in which events appear:

```
(event0 | event1 | ...  | eventN)*
```

where `event0` through `eventN` are the events that occur on transitions in the inferred SA. SAs that follow the unordered template work well when correct scenarios and erroneous scenarios often contain different events.

Name projection SAs that follow this template distinguish scenarios based on a single name, say X, which occurs in the inferred SA:

```
(event0(...  X ...)
 | event1(...  X ...)
 | ...
 | eventN(...  X ...)
 | wildcard)*
```

where `event0` through `eventN` are events that occur on transitions in the inferred SA, and *wildcard* matches any event. Name projections work well when the inferred SA mentions several names, because they allow the user to check correctness with respect to one name at a time. The template above pays no attention to the order of events; more generally, name projections can be any FA (which accepts the scenarios) whose transitions are all labeled by *wildcard* or by an event that refers to X.

Seed order SAs that follow this template distinguish among scenarios based on which events appear before the seed and which events appear after the seed:

```
(event0 | event1 | ...  | eventN)*;
event [seed];
(event0 | event1 | ...  | eventN)*
```

where `event0` through `eventN` are the events that occur on transitions in the inferred SA. Because SAs that follow the seed order template pay attention to ordering, they can distinguish scenarios that can not be distinguished by an unordered SA. However, the ordering is very limited, so the concept lattice stays small.

7.3.2 Strategies for using Cable

Cable allows the user to inspect and label concepts in any order. Some orders are better than others, however. To use Strauss effectively, a user should have some strategy for selecting concepts to inspect and label.

An important question is "how much does the user's choice of strategy matter?". In order to answer that question, this section defines several common-sense strategies whose cost can be measured automatically, given a reference labeling for the traces, and Section 7.4 measures the relative performance of these strategies on several labelings.

I measure the cost of a strategy by counting the number of Cable operations—inspecting concepts and labeling traces—that the strategy performs. I include the cost of inspecting concepts, because otherwise an optimal strategy could inspect every concept and use that perfect information to minimize the number of labeling operations. Including the cost of labeling is not as essential, but I include it because I want an optimal strategy to generate a shortest sequence of labeling operations. Note that I do not allow a strategy to label a concept without inspecting it first.

The strategies are

Top-down This strategy makes a series of lattice traversals. Each traversal visits the Unlabeled and PartiallyLabeled concepts in breadth-first order, starting at the top concept and proceeding downwards. At each concept, the strategy marks it as visited and inspects its unlabeled traces. If all unlabeled traces should receive the same label, then the strategy labels them.

The advantage of this strategy is that it does not waste time visiting concepts with a common parent whose traces can be assigned a label. The disadvantage of this strategy is that it may visit many concepts that can not be labeled because they include traces that should receive different labels.

Bottom-up This strategy loops over the lattice concepts until all concepts are FullyLabeled, visiting concepts in a bottom-up order. At each iteration, the strategy visits a concept that is not FullyLabeled but whose children are all FullyLabeled.

The advantage of this strategy is that it never visits concepts that can not be labeled because they are too general. The disadvantage is that it misses most opportunities to label many traces at once.

Random This strategy visits concepts in random order, never visiting FullyLabeled concepts and stopping when all concepts are FullyLabeled.

Optimal This strategy visits concepts in an optimal order. An optimal order is an order that minimizes the cost.

Real users do not follow any of these strategies exactly. One reason is that a real user is limited: for example, even when all of a concept's traces should receive the same label, the user might need to inspect the concept's subconcepts to convince himself of that fact. Another reason is that a real user makes heuristic decisions: for example, he may realize that the children of a certain concept should be visited first because the parent has an interesting transition in its attribute set.

7.3.3 Well-formed lattices

Because Cable only allows the user to label the traces in concepts en masse (with the "Label traces" command), a bad concept lattice can make it impossible for the user to give traces a desired labeling. I say that such lattices are not *well-formed* for the desired labeling.

A well-formed lattice for a labeling is a lattice where every concept is well-formed for that labeling. I define a well-formed concept recursively; a concept c is well-formed for a labeling iff one of the following cases holds:

1. The labeling gives the same label to every trace in c.

2. All of the children of c are well-formed for the labeling, and the labeling gives the same label to every trace in c that is not in a child of c.

Intuitively, a concept c is well-formed for a labeling if there is a trace of "Label traces" commands that puts c in the FullyLabeled state with the desired labeling. The first case says that c can be put in the FullyLabeled state simply by labeling its traces. The second case says that c can be put in the FullyLabeled state by putting all of its children in the FullyLabeled state and then labeling the unlabeled traces of c.

If the concept lattice is not well-formed, it is impossible to label all of the traces with Cable, without changing the FA. In particular, none of the above strategies would succeed.

It is easy to see how lattices that are not well-formed could arise. For example, suppose that it is correct to call a routine `foo` an even number of times and incorrect to call `foo` an odd number of times. Consider a buggy specification whose FA has one accepting state and one transition to and from that state on `foo`. This specification accepts all traces of `foo` calls. The concept lattice induced by this specification and any set of traces would put all traces of calls to `foo` in the same concept, since they all exercise the sole FA transition. That concept, and hence the whole lattice, would not be well-formed for the labeling that labels correct traces as "good" and incorrect traces as "bad".

The user does have some options when presented with a lattice that is not well-formed. One option is to change the FA and construct a better concept lattice, using Cable's "Focus" command. Another option is to label concepts that are not well-formed as "mixed"; the user can label the traces in those concepts by hand, or use my method again, with a different FA and with the set of traces restricted to the "mixed" traces.

7.4 Experimental Results

This section evaluates the usefulness of Cable for debugging specifications mined by Strauss.

I analyzed traces from full runs of 72 programs that use the Xlib and X Toolkit Intrinsics libraries for the X11 windowing system; in all, I collected 90 traces. The programs came from the X11 distribution, the X11 contrib directory, and from the programs installed for general use at the University of Wisconsin. Each trace records events for all X library calls and for all callbacks from the X library to client code. See Chapter 6 for more information about these traces.

I mined 17 specifications from these traces and used Cable to debug them. The debuged specifications found a total of 199 bugs. Chapter 9 discusses the debugged specifications in detail; a short English description of each specification is in Appendix A.

All of these specifications are fairly simple, and none of them contain loops. Consequently, the longest scenario through each SA is very short, usually less than ten events long. Debugging specifications that accept such short scenarios is actually a worst case for my method because when Strauss's front end generates short scenarios, it does not generate very many unique scenarios. So, it is relatively easy to debug these specifications simply by looking at a representative from each set of identical traces. Nonetheless, the results in Section 7.4.2 show that my

Spec.	Scenarios	FA		Lattice		Time (ms)
		|Q|	|T|	|C|	|E|	
PrsAccelTbl	9	3	10	12	19	3.2
PrsTransTbl	3	3	4	6	7	1.6
Quarks	48	63	93	135	254	80.4
RegionsAlloc	18	8	14	53	111	21.8
RegionsBig	267	370	604	678	1385	2670.4
RmvTimeOut	2	3	3	4	4	1.3
XFreeGC	10	10	13	23	38	5.7
XGContextFromGC	25	22	36	73	151	32.0
XGetSelOwner	4	5	6	10	16	2.1
XInternAtom	10	3	11	13	21	4.1
XPutImage	6	3	7	10	14	2.6
XSaveContext	496	989	1519	2162	5017	21402.4
XSetFont	35	20	37	132	306	74.8
XSetSelOwner	7	3	8	10	15	2.8
XtFree	112	95	171	380	869	1506.9
XtOwnSel	24	4	23	36	69	14.0
XtRealizeProc	23	22	36	80	166	43.4

Table 9: Running time of concept analysis with respect to 17 mined specifications. **Scenarios** reports the number of scenarios in the lattice (none of which are identical), **FA** reports the number of states and transitions in the FA that defined similarity, **Lattice** reports the number of concepts and edges in the concept lattice, and **Time (ms)** reports the running time of the analysis, in milliseconds.

method was better than this brute-force method.

Measurements of running time were taken on an Ultra Enterprise 6000 Server; the machine uses 248 Mhz SPARCv9 processors (I used one processor only) and runs Solaris 5.8.

7.4.1 Cost of concept analysis

The statistics in Table 9 demonstrate that concept analysis is affordable. The times in the table do not include reading and parsing the traces, nor do they include writing the final lattice to disk. The reported time is the shortest time from three runs; the time for each run did not vary significantly. This table reports every scenario that Strauss extracted: I removed some scenarios, according to the value of a flag argument (see Section 3.3.1, page 52), before learning three specifications (XGetSelOwner, XSetSelOwner, and XtOwnSelection).

Since Strauss extracted many identical scenarios, I built the lattice from representatives for classes of identical scenarios, rather than from all of the scenarios.

Although concept lattices are potentially exponentially large in the number of objects or attributes (whichever is lower), the size of the lattices generated for my specifications varied roughly linearly with the number of FA transitions. The times seem to vary slightly worse than linearly, but it is hard to tell for sure, since many of the times were so short. These observations agree with the more thorough empirical evaluation that Godin and others did of their algorithm (which I use) in their paper [23].

7.4.2 Benefit of concept analysis

Table 10 compares the cost of labeling by a variety of methods, where cost is defined as in Section 7.3.2. I used Cable to debug each specification and create an accurate labeling. Then, I measured the cost of obtaining the same labeling with each method. Because the Top-down, Bottom-up, and Random strategies have non-deterministic costs, Table 10 reports the lowest cost for Top-down and Bottom-up and the arithmetic mean of the cost of 1024 trials for Random. I was unable to measure the cost of the automatic strategies for the four largest specifications, because the program I wrote to evaluate these strategies took too

Spec.	**Cost of labeling**					
	Opt	**Exp**	**Top-down**	**Btm-up**	**Rand**	**Base**
PrsAccelTbl	4	8	12	16	14.1	18
PrsTransTbl	2	2	2	4	3.4	6
Quarks	2	2	2	16	8.5	16
RegionsAlloc	8	16	15	20	16.0	20
RegionsBig	X	149	X	X	X	540
RmvTimeOut	2	2	2	4	3.4	4
XFreeGC	6	12	10	20	14.5	20
XGContextFromGC	14	24	32	50	38.9	50
XGetSelOwner	4	5	5	4	5.0	4
XInternAtom	4	5	5	20	13.5	20
XPutImage	6	10	17	12	15.6	12
XSaveContext	X	42	X	X	X	184
XSetFont	X	53	X	X	X	56
XSetSelOwner	4	5	5	12	9.4	12
XtFree	X	28	X	X	X	224
XtOwnSel	4	5	5	14	10.6	14
XtRealizeProc	12	13	26	76	51.2	76

Table 10: The cost of labeling with various Cable strategies: Optimal (**Opt**), Expert (**Exp**), Top-down (**Top-down**), Bottom-up (**Btm-up**), and Random (**Rand**). These costs are compared to the Baseline method (**Base**), which does not use Cable. I was unable to measure the costs of the automatic strategies for four specifications; an X stands in for the missing cost.

long to run.

In addition to the strategies listed in Section 7.3.2, Table 10 lists two other methods:

Expert This method measured the actual cost of labeling for the expert user. The expert used a mostly top-down approach, but sometimes directed his search based on transitions he found interesting.

Baseline As mentioned above, many of the traces were identical. Instead of using Cable, this method simply divides the traces into classes of identical events and then counts the cost of inspecting and labeling each class separately. That is, the cost of Baseline is two times the number of classes of identical events.

Comparing the cost of Expert labeling with the cost of Baseline labeling indicates the value of Cable in practice. By this measure, the advantage of using Cable increases as the number of different scenarios increases.

Cable does not appear to have a large advantage for specifications built from less than 10 unique scenarios. For three of these specifications (XGetSelOwner, PrsTransTbl, and RmvTimeOut), the cost of Baseline was very low. For these specifications the cost for the Expert was also very low. For five other specifications (Quarks, XSetSelOwner, XtOwnSel, XInternAtom, and PrsAccelTbl), the cost of Baseline labeling was a bit higher, while the cost of Expert labeling remained very low. Cable shows an advantage here. Finally, for RegionsAlloc, XFreeGC, and XPutImage, the cost of both kinds of labeling was a bit higher, although the cost of Baseline labeling was still slightly higher than the cost of Expert labeling.

Cable was more useful for debugging specifications built from many tens or hundreds or scenarios. The improvement was sometimes dramatic, as in the case of XtFree. Two specifications were hard to debug, even with Cable: RegionsBig was much easier to debug with Cable than by hand, but still required 149 Cable operations; and XSetFont as just barely easier to debug with Cable than by hand.

Some other observations:

- Expert labeling never did much worse than Baseline labeling, and sometimes did significantly better.

- Because Baseline labeling labels each class of identical traces separately, it does not take into account generalization by the learner. The cost for Expert labeling includes choosing labels to ensure good generalization and verifying that the learner generalized well. The cost for Baseline does not include these actions, so it is an underestimate.

- Top-down labeling and Random labeling beat Baseline labeling on every specification except XGetSelOwner and XPutImage. Only XPutImage is significant, since the cost of labeling XGetSel is very low for all strategies. The fact that these blind strategies do well indicates that the concept lattice clusters traces appropriately.

Finally, note that Bottom-up labeling is equivalent to Baseline labeling on these specifications, but not in general. These specifications have no loops, so each class of identical traces has a characteristic set of FA transitions. These sets appear as concepts near the bottom of the concept lattice.

7.5 Conclusion

This chapter described a method for debugging temporal specifications. The method uses concept analysis to cluster the violation traces from a program verification tool or scenario traces of a specification miner, so that users can label them quickly. I found that this method is efficient, both in terms of machine resources and in terms of human resources.

Bibliography

[1] Ammons, G., Bodík, R., and Larus, J. R. Mining specifications. In *Proceedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)* (Jan. 2002), pp. 4–16.

[2] Ball, T., and Rajamani, S. K. The SLAM project: debugging system software via static analysis. In *Proceedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)* (Jan. 2002), pp. 1–3.

[3] Barrett, G. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering 15*, 5 (may 1989), 611–621.

[4] Biermann, A. W., and Feldman, J. A. On the synthesis of finite-state machines from samples of their behaviour. *IEEE Transactions on Computers 21* (1972), 591–597.

[5] Boigelot, B., and Godefroid, P. Automatic synthesis of specifications from the dynamic observation of reactive programs. In *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)* (Apr. 1997), vol. 1217 of *Lecture Notes in Computer Science*, pp. 321–333.

[6] Bush, W. R., Pincus, J. D., and Sielaff, D. J. A static analyzer for finding dynamic programming errors. *Software Practice and Experience 30* (2000), 775–802.

[7] Chakaravarthy, V. T., and Horwitz, S. On the non-approximability of points-to analysis. *Acta Informatica 38*, 8 (2002), 587–598.

[8] Cook, J. E., and Wolf, A. L. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology 7*, 3 (July 1998), 215–249.

[9] Corbett, J., Dwyer, M., Hatcliff, J., and Robby. Expressing checkable properties of dynamic systems: The bandera specification language. KSU CIS Technical Report 2001-04, Kansas State University, June 2001.

[10] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zheng, H. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering* (June 2000).

[11] Das, M., Lerner, S., and Seigle, M. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI)* (2002).

[12] DeLine, R., and Fähndrich, M. Enforcing high-level protocols in low-level software. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)* (June 2001), pp. 59–69.

[13] Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. Extended static checking. Research Report 159, Compaq Systems Research Center, Dec. 1998.

[14] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering* (May 1999).

[15] Ernst, M. D. *Dynamically Discovering Likely Program Invariants.* PhD thesis, University of Washington, Aug. 2000.

[16] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions in Software Engineering 27*, 2 (Feb. 2001), 1–25.

[17] Evans, D., Guttag, J., Horning, J., and Tan, Y. M. Lclint: A tool for using specifications to check code. In *SIGSOFT Symposium on the Foundations of Software Engineering* (Dec. 1994), pp. 87–96.

[18] Flanagan, C., and Leino, K. R. M. Houdini, an annotation assistant for ESC/Java. In *International Symposium on FME 2001: Formal Methods for Increasing Software Productivity, LNCS* (2001), vol. 1.

[19] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[20] Gansner, E. R., and North, S. C. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience 30*, 11 (Sept. 1999), 1203–1233.

[21] Ghosh, A. K., Michael, C., and Shatz, M. A real-time intrusion detection system based on learning program behavior. In *RAID 2000* (2000), vol. 1907 of *Lecture Notes in Computer Science*, pp. 93–109.

[22] Godefroid, P. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1997), pp. 174–186.

[23] Godin, R., Missaoui, R., and Alaoui, H. Incremental concept formation algorithms based on galois (concept) lattices. *Computational Intelligence 11*, 2 (1995), 246–267.

[24] Gries, D. *The Science of Programming.* Springer-Verlag, New York, New York, USA, 1981.

[25] Hall, A., and Chapman, R. Correctness by construction: Developing a commercial secure system. *IEEE Software 19*, 1 (2002), 18–25.

[26] Hallem, S., Chelf, B., Xie, Y., and Engler, D. A system and language for building system-specific, static analyses. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (May 2002), pp. 69–82.

[27] Hangal, S., and Lam, M. S. Tracking down software bugs using automatic anomaly detection. In *Proceedings of ICSE 2002* (2002).

[28] Hatcliff, J., and Dwyer, M. Using the bandera tool set to model-check properties of concurrent java software. In *Proceedings of CONCUR 2001* (Aug. 2001), vol. 2154 of *LNCS*, pp. 39–58.

[29] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[30] Horwitz, S., Prins, J., and Reps, T. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages* (San Diego, California, Jan. 1988), pp. 146–157.

[31] Jackson, D., and Rollins, E. J. A new model of program dependences for reverse engineering. In *SIGSOFT Symposium on the Foundations of Software Engineering* (Dec. 1994), pp. 2–10.

[32] Johnson, N. L., and Kotz, S. *Urn Models and Their Application: An Approach to Modern Discrete Probability Theory.* John Wiley and Sons, 1977.

[33] Leino, K. R. M., Nelson, G., and Saxe, J. B. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, Oct. 2000.

[34] Michael, C., and Ghosh, A. Using finite automata to mine execution data for intrusion detection: a preliminary report. In *RAID 2000* (2000), vol. 1907 of *Lecture Notes in Computer Science*, pp. 66–79.

[35] Murphy, K. P. Passively learning finite automata. Tech. Rep. 96-04-017, Santa Fe Institute, 1996.

[36] Nimmer, J. W., and Ernst, M. D. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification* (Paris, France, July 2001).

[37] Nye, A. *Xlib Programming Manual*, third ed. O'Reilly and Associates, July 1992.

[38] The Open Group. *Motif 2.1 Programmer's Guide*, 1997.

[39] Park, D. Y. W., Stern, U., Sakkebaek, J. U., and Dill, D. L. Java model checking. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)* (sep 2000), pp. 253–256.

[40] Pennington, H. *GTK+/Gnome Application Development*. New Riders Publishing, 1999.

[41] Raman, A. V., and Patrick, J. D. The sk-strings method for inferring PFSA. In *Proceedings of the workshop on automata induction, grammatical inference and language acquisition at the 14th international conference on machine learning (ICML97)* (1997).

[42] Reiss, S. P., and Renieris, M. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)* (Los Alamitos, California, May 2001), IEEE Computer Society, pp. 221–232.

[43] Rosenthal, D. *Inter-client communication conventions manual (ICCCM), version 2.0.* X Consortium, Inc. and Sun Microsystems, 1994. Part of the X11R6 distribution.

[44] Siff, M. *Techniques for software renovation.* PhD thesis, University of Wisconsin, Madison, 1998.

[45] Staskauskas, M. G. An experience in the formal verification of industrial software. *Communications of the ACM 39*, 12 (Dec. 1996), 256–272.

[46] Tarjan, R. E. Efficiency of a good but not linear set union algorithm. *Journal of the ACM 22*, 2 (1975), 215–225.

[47] Visser, W., Havelund, K., Brat, G., and Park, S. J. Model checking programs. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)* (sep 2000), pp. 3–12.

[48] Wagner, D., and Dean, D. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy* (May 2001).

[49] Wall, L. *Programming Perl*, third ed. O'Reilly and Associates, July 2000.

[50] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering SE-10*, 4 (July 1984), 352–357.

[51] Whaley, J., Martin, M. C., and Lam, M. S. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis* (jul 2002), pp. 218–228.

[52] Wille, R. Restructuring lattice theory: an approach based on lattices of concepts. *Ordered Sets* (1982), 445–470.