Debugging Temporal Specifications with Concept Analysis

Glenn Ammons *
ammons@cs.wisc.edu

David Mandelin †
Rastislav Bodík †‡
{mandelin,bodik}@cs.berkeley.edu

James R. Larus §
larus@microsoft.com

ABSTRACT

Program verification tools (such as model checkers and static analyzers) can find many errors in programs. These tools need formal specifications of correct program behavior, but writing a correct specification is difficult, just as writing a correct program is difficult. Thus, just as we need methods for debugging programs, we need methods for debugging specifications.

This paper describes a novel method for debugging formal, temporal specifications. A straightforward way to debug a specification is based on manually examining the short program execution traces that program verification tools generate from specification violations and that specification miners extract from programs. This method is tedious and error-prone because there may be hundreds or thousands of traces to inspect. Our method uses concept analysis to automatically group traces into highly similar clusters. By examining clusters instead of individual traces, a person can debug a specification with less work.

To test our method, we implemented a tool, Cable, for debugging specifications. We have used Cable to debug specifications produced by Strauss, our specification miner. We found that using Cable to debug these specifications requires, on average, less than one third as many user decisions as debugging by examining all traces requires. In one case, using Cable required only 28 decisions, while debugging by examining all traces required 224.

1. INTRODUCTION

Program verification tools [2–6, 10, 13, 17, 21] can find many errors in programs. These tools need formal specifications of correct program behavior, but writing a correct specification is difficult, just as writing a correct program is difficult. One partial solution is specification mining [1], but specification miners can produce buggy specifications. If program verification tools are to be more effective and widely used, we need methods for debugging specifications, because without these methods, too few specifications will be developed.

Very small specifications can be debugged by inspection. A natural way to debug a more complicated formal specification is by testing it. Conceptually, to test a specification, the specification's 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* can be expressed as a finite automaton (FA) that accepts some program execution traces and rejects others. A tool that verifies temporal specifications generates short program execution traces 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 the trace demonstrates an error or not. If the trace is not erroneous, it should be added to the language of the FA.

A similar method works for debugging temporal specifications found by a specification miner. Given data collected during a few runs of one or more programs, the miner generates a large number of short *scenario traces* and infers a specification FA from them; if some of the runs contain errors (as often happens), some of the scenario traces are also erroneous, and the miner learns an FA that accepts erroneous traces. Worse, this FA is usually more complicated than an FA that accepts only correct traces, so it is hard to debug by inspection. To debug such a specification, a specification expert looks at each scenario trace and decides whether it is erroneous or not. If the scenario trace is erroneous, then the expert tells the miner to ignore it when inferring a correct specification.

These debugging methods are tedious and error-prone because a person must inspect many traces—some program verification tools and miners generate hundreds or thousands of traces [1, 3, 13]. This paper describes a novel method for debugging formal, temporal specifications that allows a person to take all of the traces into consideration without individually inspecting every trace.

In our method, an automatic tool finds similarities within a set of program execution traces and uses *concept analysis* [22] 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 our method to look at a few clus-

*Department of Computer Sciences, University of Wisconsin, Madison, Wisconsin, USA.
†Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, USA.
‡IBM T.J. Watson Research Center, Hawthorne, New York, USA.
§Microsoft Research, Redmond, Washington, USA.

ters 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 to classify the cluster's traces as erroneous or not.

Concept analysis clusters objects hierarchically, producing a *concept lattice* of small clusters and big clusters, with small clusters contained within big clusters. Moreover (and this is a key property), the traces in small clusters are more alike than the traces in big clusters.

Hierarchical clustering is essential. The ideal clustering tool would divide the traces into two clusters: a cluster of traces that the author would classify as erroneous and a cluster of traces that the author would classify as correct. 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 splitting 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 classification 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.

Our method defines the similarity of a set of traces in terms of the transitions of an FA that recognizes traces. We 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 that is itself expressed in terms on an FA [[**Awkward**]]. The definition also enables our use of concept analysis, which clusters objects with discrete attributes. In our case, objects represent traces and attributes represent FA transitions.

To test our method, we implemented a tool, Cable, for debugging specifications. We have used Cable to debug specifications produced by Strauss, our specification-miner [1]. The corrected specifications found 199 bugs in widely distributed X11 programs, including serious race conditions and performance bugs. We found that using Cable to debug these specifications requires less than one-third as many user decisions as debugging by examining all traces requires. In one case, using Cable required only 28 decisions, while debugging by examining all traces required 224. We also found that concept analysis is affordable: it never took longer than about 22 seconds to run.

1.1 Contributions

This paper describes the following contributions:

- A novel method for debugging temporal specifications based on hierarchical clustering. The method applies not only to mined specifications—where it fills a large hole left unexplored by our previous work [1]—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 classifying traces by exploring a cluster hierarchy.

1.2 Organization of the paper

The rest of the paper is organized as follows. Section 2 presents two examples, which demonstrate how to debug specifications by examining clusters of traces. Section 3 presents concept analysis and shows how to apply it to clustering traces. The Cable tool is described in Section 4, as are strategies for using it effectively. Section 5 evaluates the usefulness of Cable for debugging specifications mined by Strauss. Section 6 discusses related work. Section 7 concludes the paper.

For all calls X = fopen() or X = popen():

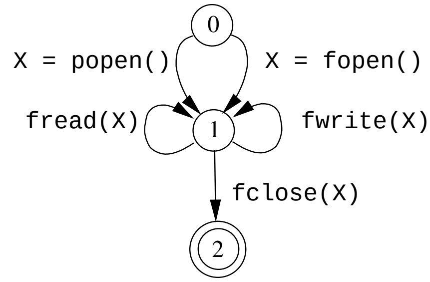

Figure 1: An incorrect temporal specification.

2. TWO EXAMPLES

This section presents two examples, which demonstrate how to debug temporal specifications with concept analysis. The first example demonstrates debugging with the aid of a verification tool by testing a specification against a program, while the second example demonstrates debugging a mined specification by inspecting the traces from which the miner inferred the specifications.

2.1 Debugging by testing

Figure 1 shows a buggy temporal specification. In general, a temporal specification captures the control and data flow of program operations in an FA. 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 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 1 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

```
X = popen(); fread(X); fwrite(X); pclose(X)
X = popen(); fread(X); fread(X); pclose(X)
X = popen(); pclose(X)

X = fopen(); fwrite(X)
X = popen(); fread(X)
X = fopen()

X = fopen(); fread(X); fread(X); pclose(X)
X = fopen(); fwrite(X); fwrite(X); pclose(X)
X = fopen(); pclose(X)
```

Figure 2: Several violation traces that could be reported by verification of the specification in Figure 1.

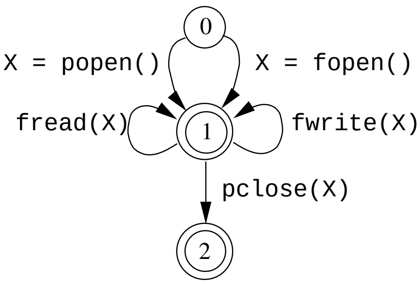

Figure 3: A small FA that recognizes violation traces from verification of the specification in Figure 1.

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 1, the violation traces (see Figure 2 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 we just did. Instead, the tool lists each trace with all of the calls it makes (not just the relevant calls we picked out in the above list), and in no particular order. For a simple example like the one in Figure 1, 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 hundreds or thousands of complicated violations (as some do [3, 13]), the problem is daunting.

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

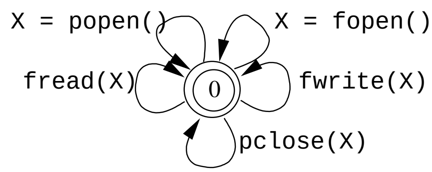

Figure 4: A very small FA that recognizes violation traces from verification of the specification in Figure 1.

Step 1a This step finds a small reference FA that recognizes the violation traces, and will be used to define trace similarity. Algorithms to learn a small FA that recognizes (at least) a set of strings have been studied extensively—see Murphy [15] for a good survey. However, a great FA learning algorithm is not essential. In fact, we have had success with FAs that recognize *all* possible traces. The key property we require is that erroneous and correct traces execute different sets of FA transitions.

Figure 3 shows a small FA that recognizes violation traces from verification of the specification in Figure 1.

Step 1b This step uses a reference FA M to define a measure of similarity for violation traces. M recognizes a trace o iff there is an *accepting sequence of transitions in M for o*, which is a sequence $(a_0, \ldots, a_n)$ such that each transition a_i is labeled by the ith event in o, the head of a_0 is the start state of M, and the tail of a_n is an accepting state of M. If a transition a is on an accepting sequence of transitions in M for o, we say that *o executes a*. Given a set O of violation traces, the *common transitions in M of O* are the transitions in M that are executed by every violation trace in O. The *similarity of O with respect to M* is the number of common transitions in M of O.

Defining similarity with respect to an FA has two benefits. First, by varying 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 3 distinguishes between traces that call `popen` before calling `pclose` and traces that call `pclose` before calling `popen`. If the order did not matter, a very small FA, such as the one given in Figure 4, 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 the top of the concept lattice contain more traces but fewer transitions than concepts at the bottom of the lattice. That is, according to our definition of similarity, the sets of traces in concepts get smaller but more similar as one moves down in the lattice.

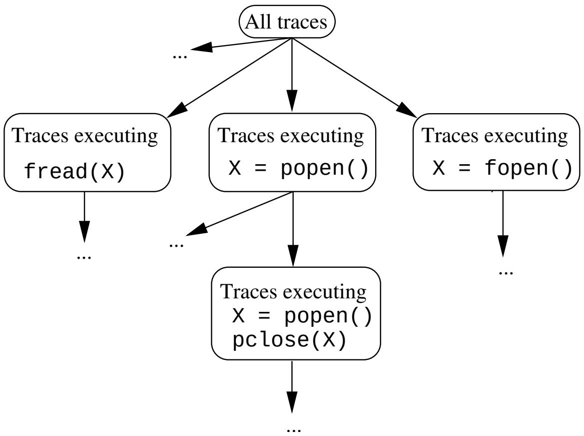

Figure 5: Part of a concept lattice that might be induced by violation traces from verification of the specification in Figure 1, with respect to the FA in Figure 3.

Figure 5 shows part of a concept lattice that might be induced by violation traces from verification of the specification in Figure 1, with respect to the FA in Figure 3.

The concept lattice is a neat summary of the violation traces. In Step 2 of our 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 4 suggest several strategies, which are evaluated in Section 5.

Suppose that the author first looks at the concept that contains traces that execute `X = popen()`. The author asks Cable to display an FA 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. Each of these child concepts contains a proper subset of the traces in the parent concept. Suppose that the first child concept he looks at contains just traces that execute both `X = popen()` and `pclose(X)`. These traces are correct, so the author labels them as "good". Finally, suppose that the author revisits the concept that contains traces that execute `X = popen()`. He asks Cable to display an FA that is inferred from the unlabeled traces in that concept. These traces execute `X = popen()` but not `pclose(X)`, 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 `X = popen()`. The traces that execute `X = 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 an FA that is inferred from all "good" traces. These traces should be accepted by the correct specification. If the author made 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 an FA 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".

For all calls X = fopen() or X = popen():

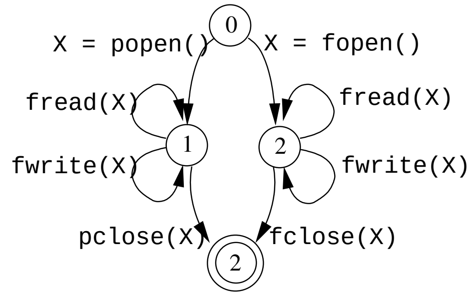

Figure 6: The result of debugging the specification in Figure 1.

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 6 shows the result of fixing the specification in Figure 1.

To summarize, our 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.

2.2 Debugging a mined specification

A specification miner is a tool for learning specifications. Figure 7 shows the architecture of our miner, Strauss. Strauss has a front end and a back end. The front end extracts *scenario traces* from a training set of program execution traces. The details of how this occurs are discussed in a previous paper [1]. The scenario traces may have bugs, because the training set may have bugs. The back end uses machine learning techniques to learn a temporal specification that accepts the scenario traces.

Suppose that Strauss learns the buggy specification in Figure 1 from a set of scenario traces that include

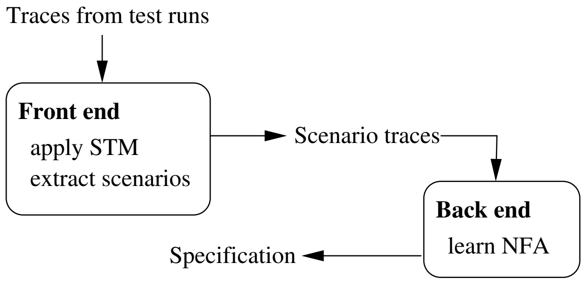

Figure 7: Architecture of Strauss.

```
X = popen(); fread(X); fwrite(X); pclose(X)
X = popen(); fread(X); fread(X); pclose(X)
X = popen(); pclose(X)

X = fopen(); fread(X); fwrite(X); fclose(X)
X = fopen(); fread(X); fread(X); fclose(X)
X = fopen(); fclose(X)
```

Figure 8: Several scenario traces.

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

An expert can produce a correct specification by rerunning the back end of Strauss only on the latter of the two kinds of traces above. Unfortunately, the traces are not summarized so neatly as they are above. In general, it is tedious and error-prone for the expert to inspect every scenario trace.

Our solution is to summarize the scenario traces neatly, before the expert sees them. The method is very similar to the method we discussed in Section 2.1. The differences are in Steps 1a and 3.

In Step 1a, the expert does not need to infer an FA that recognizes the traces. He already has one: namely, the FA from the miner's buggy specification. On the other hand, if the miner infers an FA that makes unnecessarily fine distinctions among scenarios, the expert may choose to use a different FA. However, in our experience, the inferred FA is usually a good starting point.

Steps 1b and 1c are just as in Section 2.1: the expert uses the FA to define a measure of similarity for traces and to build a concept lattice. Step 2 is the same, too: the expert uses Cable to label as "good" the scenario traces that belong in the correct specification and to label as "bad" the scenario traces that don't belong.

The expert fixes the specification in Step 3. In Section 2.1, the specification author did this manually. In mining, the expert just runs the back end of the miner on the traces that have been labeled "good".

There is a further problem, however. A useful miner also *generalizes*: the specification accepts some traces that were not in its training set, but are similar to traces in the training set. For example, a miner given the "good" scenario traces in Figure 8 would ideally produce an FA that accepts any number of calls to

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

`fread` and `fwrite` between calls to `popen` and `pclose` and between calls to `fopen` and `fclose`. Unfortunately, in generalizing, the miner can make mistakes: in this case, a miner might produce an FA that allows a call to `popen` to be followed by a call to `fclose`.

To address this problem, the expert can vary parameters on the miner, but a more frequently fruitful solution is to further subdivide the training set and apply the miner separately to each division. In our example, if the expert observes that the miner 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".

3. APPLYING CONCEPT ANALYSIS

Concept analysis [22] is a hierarchical clustering technique for objects with discrete attributes. This section reviews concept analysis and explains how to use it to cluster program execution traces with respect to a temporal specification. In the process, we 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.

3.1 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 9 shows an example where the objects are animals and the attributes are adjectives that describe animals. [1]

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. To define concepts formally, the standard formulation defines two mappings $\sigma_R : 2^O \rightarrow 2^A$ and $\tau_R : 2^A \rightarrow 2^O$. 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 $\tau_R(Y) = X$. X is called the *extent* of the concept and Y is called the *intent* of the concept.

The choice of O, A, and R uniquely defines a set of concepts. Concepts are partially ordered under the ordering $\leq_R$, defined as follows: $(X_0, Y_0) \leq_R (X_1, Y_1)$ iff $X_0 \subseteq X_1$. This partial order induces a complete lattice on concepts, called the *concept*

[1] We took this example from Michael Siff's thesis [20].

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})

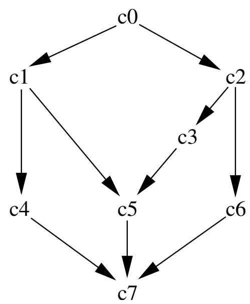

Figure 10: Concept lattice for Figure 9.

lattice. Figure 10 shows the concept lattice for the example in Figure 9.

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 $Y_0 \supseteq Y_1$. This fact allows the definition of 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 $\mathrm{sim}(X_0) \geq \mathrm{sim}(X_1)$.

3.1.1 *Efficiency of concept analysis*

There are several algorithms for building concept lattices. The algorithm we use is due to Godin and others [11] (we 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 our applications, k is typically less than ten, while $|O|$ ranged up to the hundreds. Our measurements (see Section 5) show that the algorithm is practical, terminating in less than 22 seconds on our largest data set, which contained 496 traces.

3.2 Clustering traces

To cluster traces with concept analysis, we need to define O, A, and R:

O The objects are the traces themselves.

A We either have in hand or can infer an FA M that recognizes the traces. The attributes are the transitions of M.

R Let $\mathrm{AS}(o)$ be the set of all accepting sequences of transitions in M for the trace o. We 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 o executes a. R can be computed by simulating each trace on the finite automaton.

This is a natural choice of R, which matches the intuition that traces with many common transitions are more alike than traces with few common transitions. 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, our 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, incorporating transition ordering into the definition of similarity is unnecessary, as any benefit can be had simply by transforming the FA (see the discussion in Section 2.1, Step 1b).

4. CABLE

The section describes Cable, our tool for debugging specifications. Cable displays the concept lattices defined in Section 3.2 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.

4.1 The Cable interface

Cable, which is based on the Dotty [9] 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 miner's specification in a similar fashion. If the expert decides that certain scenario traces are not erroneous, he labels them "good". Scenario traces that are erroneous are labeled "bad". After the expert has labeled every trace, he uses Cable to view the traces labeled "good" and reruns the miner on those traces. Labels are a flexible mechanism: as Section 2.2 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 the empty concept (the 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 *concept views*. Cable supports the following views of a concept:

FA Cable uses an FA learner (Raman and Patrick's sk-strings learner [18]) to construct a summary FA that accepts concept traces and then displays this FA. In our experience with Cable, this was the most frequently used concept view. Cable uses Raman and Patrick's sk-strings learner to construct FAs.

If the concept is PartlyLabeled or FullyLabeled, then the user can choose which concept traces to include in the view: the user can choose to include all traces, only unlabeled traces, or only traces with a given label. Cable constructs the summary FA only from included traces (we say that these traces are *in the view*). This feature is particularly useful once all concepts are FullyLabeled: the user can obtain an FA for all traces with a particular label l by viewing the FA of l-labeled traces in the top concept.

The FA view also provides *selection by transitions*, which enables the user to find traces that execute or do not execute selected transitions in the summary FA. By clicking on transitions in the FA view, the user selects a set of included transitions and a set of excluded transitions. These selections correspond to a selection of traces: a trace is selected iff it is in the view and it executes all included transitions and no excluded transitions.

Selection by transitions enables *navigation by transitions*. After transitions (and hence traces) are selected, Cable will navigate the user to the smallest concept that contains a superset of the selected traces. This concept is smaller and more similar than the current concept (unless the lattice forces Cable to select the current concept). Transition selection thus equips the user with control over which of the smaller, more similar concepts he would like to examine next. As Section 5.4 reports, we found this feature useful for large lattices, where traversal (e.g., Top-down, defined in Section 4.2) was ineffective because many concepts could not be immediately labeled.

Transitions This view displays the transitions in the reference FA that belong to the concept. In our experience, this has been the second most frequently used summary because we often know that the label for a trace depends on whether the trace executes a certain set of transitions in the reference FA or not.

As with the FA view, if the concept is PartlyLabeled or FullyLabeled, the user can choose which concept traces to include in the transitions view.

Traces Cable prints the traces in the concept. We do not use this summary very often because it usually generates more output than we can understand.

As with the FA view, if the concept is PartlyLabeled or FullyLabeled, the user can choose which concept traces to include in the transitions view.

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 an 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 our experiments, we always started Cable with a cluster determined by our miner's inferred FA. If this cluster appeared complicated, we started a focused session. The FAs that we used for focusing followed one of the following three templates:

Unordered FAs that follow this template distinguish among traces based on which events appear in traces, 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 FA. FAs that follow the unordered template work well when correct traces and erroneous traces often contain different events.

Name projection FAs that follow this template distinguish traces based on a single name, say X, that occurs in the inferred FA:

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

where `event0` through `eventN` are events that occur on transitions in the inferred FA, and *wildcard* matches any event. Name projections work well when the inferred FA 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 (that accepts the traces) whose transitions are all labeled by *wildcard* or by an event that refers to X.

Seed order FAs that follow this template distinguish among traces 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 FA. Because FAs that follow the seed order template pay attention to ordering, they can distinguish traces that cannot be distinguished by an unordered FA. However, the ordering is very limited, so the concept lattice stays small.

4.2 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?". 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 5 measures the relative performance of these strategies on several labelings.

We measure the cost of a strategy by counting the number of Cable operations—inspecting concepts and labeling traces—that the strategy performs. We 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 we include it because we want an optimal strategy to generate a shortest sequence of labeling operations. Note that we 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 cannot be labeled because they include traces that should receive different labels.

Bottom-up This strategy loops over the concept lattice until all concepts are FullyLabeled, visiting concepts in a bottom-up order. On 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 cannot 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 a certain concept should be visited first because it has an interesting transition in its attribute set.

4.3 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. We 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. We 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 sequence 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 sequences of `foo` calls. The concept lattice induced by this specification and any set of traces would put all sequences 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 our method again, with a different FA and with the set of traces restricted to the "mixed" traces.

5. EXPERIMENTAL RESULTS

This section evaluates the usefulness of Cable for debugging specifications mined by Strauss [1].

We analyzed traces from full runs of 72 programs that use the Xlib and X Toolkit Intrinsics libraries for the X11 windowing system; in all, we 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.

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

5.1 Specifications

We used Cable to debug seventeen Strauss specifications. For each specification, Table 1 lists the number of states and transitions in the specification's FA (after debugging) and translates the specification into English.

These are important specifications. The debugged specifications found a total of 199 bugs, including resource leaks, potential races, and performance bugs.

All of these specifications are fairly simple, and none of them contain loops. Consequently, the longest scenario through each FA is very short, usually less than ten events long. Debugging specifications that accept such short scenarios is actually a worst case for our method because when Strauss's front end generates short scenario traces, it does not generate very many unique scenario traces. 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 5.3 show that our method was better than this brute-force method.

5.2 Cost of concept analysis

The statistics in Table 2 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 trace that Strauss extracted: we removed some traces before debugging three specifications (XGetSelOwner, XSetSelOwner, and XtOwnSelection). [**[The removed traces had an uninteresting selection value. We should just report the numbers *after* removing those traces.**]] Since Strauss extracted many identical scenario traces, we built the latice 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 our 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 we use) in their paper [11].

5.3 Benefit of concept analysis

Table 3 compares the cost of labeling by a variety of methods, where cost is defined as in Section 4.2. One of the authors (an expert user and developer of the tool) used Cable to debug each specification and create an accurate labeling. Then, we 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 3 reports the lowest cost for Bottom-up and the arithmetic mean and standard deviation of the cost of 1024 trials for Top-down and Random. We were unable to measure the cost of the automatic strategies for the four largest specifications, because the program we wrote to evaluate these strategies took too long to run.

In addition to the strategies listed in Section 4.2, Table 3 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 scenario traces increases.

Cable does not appear to have a large advantage for specifications built from less than 10 unique scenario traces. 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 of scenario traces. 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.

[[**DPM-ADDITION**]]

5.4 Time studies

In another set of experiments, we studied how long it took a user to debug a specification using Cable. For 11 of the 17 specifications, we measured the wall clock time elapsed during a Cable debugging session.

The reference FA used was the summary FA for the top concept of the lattice.

The user strategy was as follows. Starting at the top concept of the lattice, the user examined the summary FA. If the FA described only correct behavior, the user labeled the concept "good". If the FA described only erroneous behavior, the user labeled the concept "bad". Otherwise, the user looked for edges that were either clearly required for all correct traces, or clearly followed only by erroneous traces. Next, the user highlighted the edge green if required, or red if erroneous. Then, the user selected the matching concept using edge navigation and applied this procedure recursively. The idea was to use edge navigation to home in on concepts that contain all correct traces or all incorrect traces. After labeling a concept, the user returned to the concept above it. If this concept still contained unlabeled traces, the user started over but with only the unlabeled traces.

[[**Question: Is the following para even remotely comprehensible without an example?**]]

We modified the definition of attributes used in the concept lattice for these experiments. As before, there was an attribute for each edge taken in the reference FA by a trace. We added an attribute for each edge not taken in the reference FA. We did this to ensure that Cable was always able to find an exactly matching concept when the user navigated by an edge in the FA.

To provide a baseline for comparison, the user also debugged each specification by classifying the individual traces, without any use of concept analysis or clustering. For this, we built a simple tool that shows traces one by one. The user examined each trace and clicked a button to classify it as "good" or "bad".

Table 5.4 summarizes the results of this experiment. In most cases, the user was able to classify all traces by examining and classifying a small number of concepts. However, in 6 of the 11 trials this took more time than the baseline method of classifying individual scenarios. The main reason was that the user typically took a long time to examine and understand the summary FA for the top concept, because it tends to be fairly large and complex. This has the benefit of helping the user better understand the specification and the trace set, but it can be time-consuming. Debugging with Cable outperformed the baseline method only in those trials where it was easy for the user to spot transitions in the summary FA that distinguished correct behavior from erroneous behavior.

5.5 Discussion

As noted in 5.4, navigation by transitions requires a large lattice to be effective. If the lattice is small, a concept that exactly matches a selected set of transitions is unlikely to exist. Conversely, if the lattice is a complete subset lattice, a concept that exactly matches a selected set will always exist. This allows the user to very effectively separate correct traces from erroneous ones by applying a small amount of domain knowledge. Unfortunately, a complete lattice provides no clustering. If the user cannot find transitions in the summary FA to use for navigation, he will be lost.

A small lattice, on the other hand, provides a great deal of clustering and is easy to explore by traversal. The problem is that the clustering is not guaranteed to cleanly separate correct traces from erroneous ones, or to correspond with the user's intuition.

This suggests that a better method would build small lattices using domain knowledge from the user. The resulting lattices would be small and easily traversable, and they would also separate traces according to the user's desired classification. This might be done by refining the measure of similarity in response to user input. For example, a tool could allow the user to select transitions like Cable does, but use them to change the lattice to form concepts according to the selections. Another possibility is a tool that infers measures of similarity from labels applied by the user.

The questions to be investigated in future work are: What measures of similarity yield the best clustering for trace classification? How can measures of similarity be refined in response to user input? What is the best user interface for suggesting measures of similarity?

6. RELATED WORK

This paper fills a large hole left unexplored by our previous work on specification mining [1]. That paper explained how to extract specifications from program execution traces, but only offered a naive mechanism—coring, or dropping low frequency transitions—for removing errors from those specifications. This paper gives a general method for debugging specifications, which applies not only to our miner's specifications but also to temporal specifications from any source.

Previous work on helping users work through the large numbers of bug reports that can be produced by verification tools has focused on ranking the bug reports, so that the user sees likely bugs before likely false positives, and severe bugs before minor bugs. An example is the Xgcc system [13], which uses statistical and other heuristics to rank likely bugs and severe, hard-to-find bugs above other bug reports. Xgcc also does some simple clustering based on which functions appear in bug reports. Another tool, PREfix [3], uses a number of filtering and ranking heuristics to reduce what they call "noise". In our opinion, ranking and clustering are complementary: ranking tells the user what reports to inspect first, while clustering helps the user avoid inspecting redundant reports.

Daikon [7], a tool for dynamically discovering arithmetic invariants, uses statistical confidence checks to suppress invariants that appear to have occurred by chance. In our case, we found that some buggy traces occurred so frequently that suppressing them similarly would also suppress valid traces.

One way to debug specifications is to assume that any part of a specification that can not be verified by a program verification tool is, in fact, wrong. This approach is used by the Houdini tool [8], which guesses many invariants and then uses ESC/Java [14] to prune out those that do not always hold. A similar approach was used to integrate the Daikon and ESC/Java tools [16]. Both tools still rely on a user to help debug specifications, because programs are buggy: an invariant that should be true (and so should be checked) may be unverifiable because of an error in the program.

This paper concentrates on debugging temporal specifications. Many program verification tools rely on Hoare-style invariants, preconditions, and postconditions [12]. It would be interesting to see if clustering techniques such as ours apply to such specifications.

The concept lattice that we build is sensitive to the particular FA used to recognize traces. Strauss uses Raman and Patrick's sk-string algorithm [18] to infer the FAs in its specifications. There are many other algorithms in the literature; Murphy has written a good survey [15]. For Cable, it would be particularly interesting to explore interactive algorithms, which would allow the user to fine-tune the concept lattice as he uses it for labeling.

Concept analysis is not the only hierarchical technique for clustering data with discrete attributes. Other techniques cluster spatially by defining a distance metric; still other techniques build graphs and find cliques or connected components. It would be worthwhile to investigate these alternative approaches; interestingly, however, concept analysis is closely related both to distance-based clustering [19] and to finding maximal cliques in a bipartite graph [23].

7. CONCLUSION

This paper described a method for debugging temporal specifications. Our 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. We found that our method is efficient, both in terms of machine resources and in terms of human resources.

8. REFERENCES

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, pages 4–16, January 2002.

[2] Thomas Ball and Sriram K. Rajamani. 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)*, pages 1–3, January 2002.

[3] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30:775–802, 2000.

[4] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

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

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

[7] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, August 2000.

[8] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium on FME 2001: Formal Methods for Increasing Software Productivity, LNCS*, volume 1, 2001.

[9] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1233, September 1999.

[10] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, January 1997.

[11] Robert Godin, Rokia Missaoui, and Hassan Alaoui. Incremental concept formation algorithms based on galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.

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

[13] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. 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)*, pages 69–82, May 2002.

[14] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[15] Kevin P. Murphy. Passively learning finite automata. Technical Report 96-04-017, Santa Fe Institute, 1996.

[16] Jeremy W. Nimmer and Michael D. Ernst. 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.

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

[18] Anand V. Raman and Jon D. Patrick. 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.

[19] Michael D. Rice and Michael Siff. Clusters, concepts, and pseudometrics. In *The First Irish Conference on the Mathematical Foundations of Computer Science and Information Technology*, volume 40 of *Electronic Notes in Theoretical Computer Science*, July 2000.

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

[21] Willem Visser, Klaus Havelund, Guillaume Brat, and Seung Joon Park. Model checking programs. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 3–12, sep 2000.

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

[23] Mohammed Javeed Zaki and Mitsunori Ogihara. Theoretical foundations of association rules. In

Proceedings of the 3rd SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'98), pages 71–78, Seattle, Washington, USA, June 1998.

Name	FA $\lvert Q \rvert$	FA $\lvert T \rvert$	Bugs True	Bugs False	English description
PrsAccelTbl	13	22	0	1	Accelerator tables must be parsed with `XtParseAcceleratorTable` before they are used.
PrsTransTbl	4	5	0	0	Translation tables must be parsed with `XtParseTranslationTable` before they are used.
Quarks	10	13	0	0	The `name` and `the_class` arguments of `XrmQGetResource` must come from calls to `XrmPermStringToQuark`.
RegionsAlloc	30	35	16	0	Every `Region` that is created by the program must eventually be destroyed by the program; every `Region` that is destroyed by the program must have been created by the program.
RegionsBig	352	623	12	0	Every `Region` that is created by the program must eventually be destroyed by the program; calls that accept `Region`s must be passed `Region`s that were either created by the program or supplied by the library.
RmvTimeOut	5	6	0	0	`XtRemoveTimeOut` can only remove time-outs added with `XtAddTimeOut`.
XFreeGC	10	13	44	0	Every `GC` that is created by the program must eventually be destroyed by the program; every `GC` that is destroyed by the program must have been created by the program.
XGContextFromGC	38	48	44	1	`XGContextFromGC` must be passed a valid `GC`; every `GC` that is created by the program must eventually be destroyed by the program; every `GC` that is destroyed by the program must have been created by the program.
XGetSelOwner	5	5	9	0	After calling `XSetSelectionOwner`, selection ownership must be verified by calling `XGetSelectionOwner`.
XInternAtom	7	15	42	0	For good performance, `XInternAtom` should not be called in the event loop.
XPutImage	7	9	2	2	The image and graphics context passed to `XPutImage` must have been created on the same display.
XSaveContext	66	86	33	0	An association installed with `XSaveContext` must eventually be deleted with `XDeleteContext`; also, the association must be used by a call to `XFindContext` at some point.
XSetFont	30	40	44	1	`XSetFont` must be passed a valid `GC`; every `GC` that is created by the program must eventually be destroyed by the program; every `GC` that is destroyed by the program must have been created by the program.
XSetSelOwner	5	9	7	0	The timestamp passed to `XSetSelectionOwner` must come from an event received from the X server.
XtFree	29	35	45	0	Memory allocated with `XtCalloc` or `XtMalloc` must be deallocated with `XtFree`; memory deallocated with `XtFree` must have been allocated with `XtCalloc` or `XtMalloc`; memory must not be deallocated twice.
XtOwnSel	5	10	1	0	The timestamp passed to `XtOwnSelection` must come from an event received from the X server.
XtRealizeProc	57	64	0	0	If a `XtRealizeProc` callback calls `XtCreateWindow`, the call must pass the callback's `widget` and `attributes` arguments to `XtCreateWindow`. Also, `XtCreateWindow` must not be called except by a `XtRealizeProc` callback.

Table 1: Seventeen Strauss specifications, which we debugged with Cable. FA reports the number of states and transitions in each specification's debugged FA and Bugs reports the number of true bugs and false positives that each specification found.

Spec.	Traces	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 2: Running time of concept analysis with respect to 17 mined specifications. Traces reports the number of scenario traces 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. [[Do not report too much precision here.]]

Spec.	Cost of labeling							
	Opt	Exp	Top-down		Btm-up	Rand		Base
			Mean	s				
PrsAccelTbl	4	8	14.0	1.9	16	14.1	2.8	18
PrsTransTbl	2	2	2.0	0.0	4	3.4	0.9	6
Quarks	2	2	2.0	0.0	16	8.6	3.2	16
RegionsAlloc	8	16	15.7	1.4	20	16.2	2.8	20
RegionsBig	X	149	X		X	X		540
RmvTimeOut	2	2	2.0	0.0	4	3.4	0.9	4
XFreeGC	6	12	12.9	2.3	20	14.4	3.4	20
XGContextFromGC	14	24	39.0	4.3	50	39.5	5.9	50
XGetSelOwner	4	5	5.0	0.0	4	4.5	0.9	4
XInternAtom	4	5	5.0	0.0	20	12.5	5.6	20
XPutImage	6	10	21.0	3.9	12	15.6	3.9	12
XSaveContext	X	42	X		X	X		184
XSetFont	16	53	52.0	5.4	56	50.3	4.8	56
XSetSelOwner	4	5	5.0	0.0	12	8.7	3.4	12
XtFree	X	28	X		X	X		224
XtOwnSel	4	5	5.0	0.0	14	9.8	3.9	14
XtRealizeProc	12	13	29.1	2.1	76	51.0	7.4	76

Table 3: 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. We were unable to measure the costs of the automatic strategies for four specifications; an X stands in for the missing cost.

Name	Wall clock time		Items visited	
	Cable	Baseline	Cable	Baseline
PrsAccelTbl	136	42	3	8
PrsTransTbl	9	11	1	2
RmvTimeOut	9	6	1	2
XFreeGC	158	107	7	25
XGetSelOwner	32	9	3	2
XInternAtom	100	37	5	10
XPutImage	148	31	9	6
XSetFont	117	133	7	30
XSetSelOwner	27	33	3	6
XtFree	251	254	25	112
XtOwnSel	98	21	5	7

Table 4: Wall clock time taken to debug specifications. Wall clock time reports the time taken to debug a specification using either Cable or the baseline method. Items visited reports the number of items visited while debugging a specification. For Cable, this is the number of concepts for which the user viewed a summary FA. For the baseline, this is the number of individual traces classified, which is the same as the number of traces in the entire set.