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

Abstract

Program verification tools (such as model checkers) are powerful tools for finding errors in programs. Unfortunately, these tools need lots of formal specifications of correct program behavior. Can we really expect programmers to write all of these specifications by hand?

This dissertation is about Strauss, a tool I wrote to bring automation to specification-writing. By observing traces of working programs, Strauss infers many temporal specifications, each of which says how correct programs use a small part of an interface.

I used Strauss to derive 17 formal specifications for the X11 windowing system (whose libraries contain over 2000 routines and over 500 data structures), and used the specifications to find bugs in several widely distributed applications.

For Rachel.

List of Figures

1 A simple interface. 4

2 A specification for the simple interface. 4

3 The architecture of Strauss and the three phases of mining with Strauss. 6

4 Interface for the running example. 8

5 Program for the running example. 8

6 Trace for the running example. 9

7 Library state for the running example. 10

8 A state-transition model for the running example. 11

9 The trace from Figure 6 and pseudocode for its semantic-trace interpretation with respect to the STM in Figure 8. 11

10 Two scenarios from the trace in Figure 6, as interpreted with the STM in Figure 8. 14

11 Syntax of trace events. The second production for *value* produces a value with an attached tuple of arguments: the arrow notation is meant to suggest a reference, not a function. 28

12 Abstract syntax for traces. 29

13 A positive specification, which says "locks released with `unlock` must have been acquired with `lock`." 32

14 A general template for positive specifications. 33

15 Concrete syntax of seed patterns. 34

16 Abstract syntax for seed patterns. 34

17 A negative specification, which says "locks must not be released twice in a row with `unlock`." 36

18 A general template for negative specifications. 36

19 A state-transition model. 39

20 Example from Figure 6, written following the abstract syntax of Figure 12. 43

21 Imperative pseudo-code for the example of Figure 20. 44

22 Grammar for STM declarations. 47

23 Conservative completion of the STM in Figure 19. 48

24 Liberal completion of the STM in Figure 19. 48

25 Abstract syntax for STMs. 49

26 The state-transition model of Figure 19, written with the abstract syntax of Figure 25. 49

27 Transition rules for a semantic trace. 52

28 Rewrite rules for the flags transformation. 55

29 Rewrite rules for dropping arguments from a semantic trace. 56

30 Rewrite rules for dropping event pairs from a semantic trace. 59

31 A scenario within its trace. 63

32 A scenario, within its trace, for the extended file interface. 66

33 Scenarios illustrating convexity. 75

34 An algorithm for extracting the largest convex scenario around e_{seed} with at most r_b ancestors and r_f descendants. 76

35 The outer loop of Strauss's algorithm for extracting scenarios. . . . 77

36 The desired set of acceptable scenarios (the output of SA learning). 87

37 Reducing SA learning to NFA learning. 87

38 Two equivalent scenarios, within their enclosing traces. 92

39 Two inequivalent scenarios, within their enclosing traces. 92

40 A state transition model for the socket interface. 95

41 A scenario generated by a program that uses the socket interface. . 96

42 A schedule for the scenario from Figure 41. 97

43 The schedule in Figure 42, with the value 7 replaced by the symbolic name `x0` and the value 8 replaced by the symbolic name `x1`. 98

44 A scenario that is pseudo-equivalent to the scenario in Figure 41. . 99

45 Naive standardization algorithm. 102

46 The named schedule from Figure 42, with a corresponding string for the NFA learner on the far right. 103

47 Two scenarios. 104

48 The algorithm of Figure 45, optimized to consider only one naming per schedule. 106

49 The algorithm of Figure 48, with the schedules considered restricted with $\leq_\sigma$. 108

50 Two nearly pseudo-equivalent scenarios and their scenario strings, with untyped and typed naming. 109

51 A replacement for `AugmentNaming` (Figure 48) that uses a different namespace for each inferred type. 111

52 A program, a library, and a sequence of calls and returns within and between the two. 128

53 High-level pseudocode for a wrapper for `Foo`. 132

54 High-level pseudocode for a wrapper for `Foo`, with the search for callback routines. 134

55 A wrapper for `Foo`, with lookup of `Foo`. 136

56 A wrapper for a callback of type `CBType`. 137

57 Wrapper for `Foo` that prints an event when the interface boundary is crossed. 138

58 An incorrect temporal specification. 153

59 Several violation traces that could be reported by verification of the specification in Figure 58. 154

60 A small FA that recognizes violation traces from verification of the specification in Figure 58. 156

61 A very small FA that recognizes violation traces from verification of the specification in Figure 58. 157

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. 159

63 The result of debugging the specification in Figure 58. 161

64 Several scenarios. 163

65 A context where the objects are animals and the attributes are adjectives that describe animals. 165

66 Concept lattice for Figure 65. The top concept is c0, and the bottom concept is c7. 166

67 Pseudocode for checking that a trace T obeys a positive specification *spec*. 195

68 XSetSelOwner. 196

69 Pseudocode for checking that a trace T obeys a negative specification *spec*. 198

70 A tricky Strauss specification. 200

71 XSetSelOwner. 212

72 XPutImage. 220

73 RegionsAlloc. 232

74 XInternAtom. 233

75 Programs versus violations of the XInternAtom specification. 234

List of Tables

1 Benefit of dropping arguments. 57

2 Benefit of dropping event pairs. 60

3 Cost of inferring flow dependences and extracting scenarios. 80

4 Cost of scheduling and naming. 122

5 Performance of NFA learning. 123

6 Time to read traces (**Construct**) and total end-to-end time to extract and standardize scenarios (**Total**). 124

7 Benefit of standardization. 125

8 The traces that I used in the experiments in this dissertation. . . . 143

9 Running time of concept analysis with respect to 17 mined specifications. 181

10 The cost of manual labeling, with and without Cable. 182

11 The cost of labeling with four automatic Cable strategies. 183

12 Time to debug specifications. 187

13 Useful specifications for the X11 interface, which I found with Strauss. 205

14 For each specification, how I found it (**Method**) and the patterns it follows (**Patterns**). 208

15 The errors that my specifications found. 221

Contents

Abstract **i**

ii

1 My thesis **1**

1.1 The specification problem 1

1.2 Strauss: a specification miner 4

1.2.1 Preprocessing 9

1.2.2 State-transition models 10

1.2.3 Extracting scenarios 13

1.2.4 Learning 14

1.2.5 Debugging 15

1.3 An overview of the dissertation 16

1.4 Related work 17

1.4.1 Other ways to develop specifications 17

1.4.2 Specification mining 20

1.4.3 Specification debugging 22

2 Preliminaries **25**

2.1 Lists, tuples, and records 25

2.2 Finite automata 26

2.3 Traces 27

2.3.1 Concrete trace syntax 27
2.3.2 Abstract Trace Syntax 30
2.3.3 Functions on events 31
2.4 Specifications 32
2.4.1 Positive specifications 32
2.4.2 Negative specifications 35

3 Semantic traces **38**
3.1 An example 43
3.2 Semantics of semantic traces 46
3.2.1 State-transition models 46
3.2.2 Operational semantics of semantic traces 49
3.3 Transformations of semantic traces 53
3.3.1 Flags 53
3.3.2 Dropping arguments 56
3.3.3 Dropping event pairs 58

4 Extracting Scenarios **61**
4.1 Inferring flow dependences 68
4.2 Extracting scenarios 72
4.2.1 Scenarios 72
4.2.2 Extracting bounded scenarios 73
4.3 Experimental results 80

5 Learning specifications **82**

5.1 Semantic equivalence 88
5.1.1 Semantic equivalence for semantic traces 88
5.1.2 Semantic equivalence for scenarios 90
5.1.3 Scalability 93
5.2 Standardization 94
5.2.1 Pseudo-equivalence 95
5.2.2 A naive algorithm for standardization 101
5.2.3 Optimizations of the naive algorithm 105
5.2.4 Extensions 109
5.3 NFA learning 113
5.3.1 NFA learning: Raman and Patrick's formulation 114
5.3.2 The sk-strings learner 117
5.4 Experiments 121

6 Tracing **126**
6.1 What to trace 127
6.1.1 Which calls and returns to trace 127
6.1.2 Which values to trace 129
6.2 Tracing X11 130
6.2.1 Which calls and returns to trace (X11) 132
6.2.2 Printing values 139
6.3 The traces 142

7 Debugging Specifications **148**
7.1 Two examples 153

7.1.1 Debugging by testing 154
7.1.2 Debugging a Strauss specification 162
7.2 Applying concept analysis 164
7.2.1 Concept analysis 165
7.2.2 Clustering traces 168
7.3 Cable 169
7.3.1 The Cable interface 169
7.3.2 Strategies for using Cable 175
7.3.3 Well-formed lattices 177
7.4 Experimental Results 179
7.4.1 Cost of concept analysis 180
7.4.2 Traversal strategies 181
7.4.3 Navigation by transitions 186
7.4.4 Discussion 189
7.5 Conclusion 191

8 Checking Specifications 192

8.1 Dynamic checking algorithms 194
8.1.1 Checking positive specifications 194
8.1.2 Checking negative specifications 197
8.2 Drawbacks of the dynamic checker 199

9 Experience with Strauss 202

9.1 Experimental setup 204
9.2 The specifications 204

9.2.1 Case study: the Book method 208
9.2.2 Case study: the Graph method 212
9.3 The errors 221
9.3.1 Protocol errors: XSetSelOwner 224
9.3.2 Resource leaks: RegionsAlloc and XFreeGC 227
9.3.3 Performance errors: XInternAtom 228

10 Final Thoughts **235**
10.1 Suggestions for future work 236
10.1.1 Increasing automation 236
10.1.2 Mining without traces 237
10.1.3 Static checking 238
10.1.4 Modeling a heap 239
10.2 What I would do differently 239

A Descriptions of specifications **241**

B Specification patterns **244**
B.0.1 The Consumer pattern 244
B.0.2 The Producer pattern 245
B.0.3 The Union pattern 246
B.0.4 The MultiConsumer pattern 247
B.0.5 The MultiConsumerAgrees pattern 248
B.0.6 The TiedConsumer pattern 249
B.0.7 The TiedProducer pattern 251

Bibliography **254**

Chapter 1

My thesis

By observing how programs use an interface, automatic tools can learn formal specifications that all programs that use the interface must obey.

1.1 The specification problem

Although the software industry spends much of its resources on improving the reliability of its software, software is notoriously buggy. At the same time, society depends more than ever on software. While it is difficult to imagine software without bugs, it is not difficult to imagine software that is significantly more reliable than the software of today. To get there, we need better and cheaper ways to avoid, detect, and correct errors in programs.

Testing, the primary current method for detecting errors in programs, is expensive and cannot guarantee that a program is free of errors. By contrast, *program-verification tools* [6, 41, 50, 27, 12, 10, 2, 11, 23] (such as type checkers, model checkers, and theorem provers) are relatively cheap to use, and some of these tools can guarantee that a program is free of certain classes of errors.

Program-verification tools work by comparing a model of a program's possible executions against a formal specification, which defines some property of correct

program executions (for example, "all allocated memory is freed" or "deadlock does not occur"). Many important tools check *temporal specifications*; a temporal specification can be expressed as a finite automaton (FA) that accepts program executions that have the property of interest and reject program executions that do not have the property. Typically, temporal specifications track the state of program values; these specifications can express properties such as "memory is not freed twice" or "if a lock is acquired, it is eventually released."

A major obstacle to the widespread use of program-verification tools is finding specifications that the tools can use to find errors, but finding such specifications is difficult. Most code is written without accompanying specifications beyond comments and other informal documentation, type declarations, and test code. Convincing programmers to write formal specifications for all new code would not solve the problem, because many programs in use today were written years ago or depend on library code that was written years ago. The sheer quantity of this old code makes writing specifications for it prohibitively expensive. Moreover, the programmers who understand the code best are no longer around to help. Finally, each specification defines one property of correct program executions, but it is not often clear (until an expensive system fails) *which* properties are important; manually writing specifications for all properties which may be important is too expensive.

In this dissertation, I show how to automate the development of formal specifications for *interfaces*, which encapsulate communication between two pieces of code, such as a program and a library. My techniques are widely applicable, because the use of interfaces pervades software development. Unless it is very small,

a program is usually a composition of modules, with modules connected by interfaces. Also, most programs (large and small) depend heavily on external libraries, which hide commonly useful code behind an interface.

For example, at the time of this writing, the Computer Systems Lab at the University of Wisconsin has installed 1332 Unix programs for the general use of students and faculty. Of these, 492 are scripts, 613 use shared libraries, and only 227 are statically linked; note that even statically linked programs use libraries. The average program that uses shared libraries uses 5.5 shared libraries. Moreover, larger programs use *more* libraries: the ten largest programs use 8.2 shared libraries, on average.

Ideally, every library developer would distribute formal specifications for his library's interface along with the library itself. This is not the current practice, because writing specifications is perceived as a distraction from library design and implementation, because customers do not demand formal specifications, and because writing formal specifications is still difficult. The techniques described in this dissertation could make distributing formal specifications with libraries more attractive for developers, by making specifications easier to write.

However, my techniques are needed most by library users, who are trying to write formal specifications for a library's interface. Library interfaces are usually documented only informally, in a natural language. Libraries can be large and complicated, and source code is not necessarily available. Even when source code is available, it might be written in an unfamiliar language and the connection between the interface and the code that implements the interface might be tenuous. For example, in the X11 windowing system, many programs are written to the X

```
type file
file new-file()
lock(in file f)
unlock(in file f)
```

Figure 1: A simple interface.

For all trace events matching

```
unlock(f = F)
```

with STM

```
lock(def use f)
unlock(def use f)
```

a scenario matches

```
lock(f = F); unlock(f = F) [seed]
```

Figure 2: A specification for the simple interface.

Toolkit Intrinsics (Xt) interface. Xt, in turn, is implemented on top of the Xlib interface. The Xlib routines communicate with the X server through a network connection, and the X server does most of the actual work of managing windows. Thus, even partially understanding the effect of a call to an Xt routine may require some understanding of the X Toolkit Intrinsics, Xlib, the network, and the X server.

1.2 Strauss: a specification miner

Specification mining is my solution to the specification problem for interfaces. In general, a specification miner takes artifacts of program development, such as the programs themselves or executions of the programs on the developers' test cases,

and searches for patterns that the artifacts have in common.

In the case of my specification-mining techniques, the input artifacts are traces of the run-time interactions of programs with a library that implements an interface. The output patterns are formal specifications that define a property of correct program behavior. In particular, the specifications are *temporal specifications* that track the state of values and specify that programs may call library routines only in certain orders and only with certain values.

To test my techniques, I implemented a specification-mining tool, called Strauss. I have used Strauss to formalize 17 specifications for the Xlib and X Toolkit Intrinsics interfaces to the X11 windowing system and used these specifications to find 61 bugs in widely distributed programs.

Strauss is widely applicable, because it makes few assumptions about libraries and programs. In particular, Strauss does *not* assume that code is available for libraries or programs. Instead of analyzing code, Strauss treats both programs and libraries as black boxes and learns specifications solely by observing how they interact with one another.

I give a formal definition of Strauss specifications in Chapter 2; for now, consider Figure 1, which shows a simple interface, and Figure 2, which shows a Strauss specification.

The interface has a type `file` for files; a constructor `new-file` that creates a file; and two other routines: `lock` acquires a lock on a file and `unlock` releases the lock on a file.

Intuitively, the specification says that "files unlocked with `unlock` must have been locked with `lock`." Note that state, order, and the identity of values are all

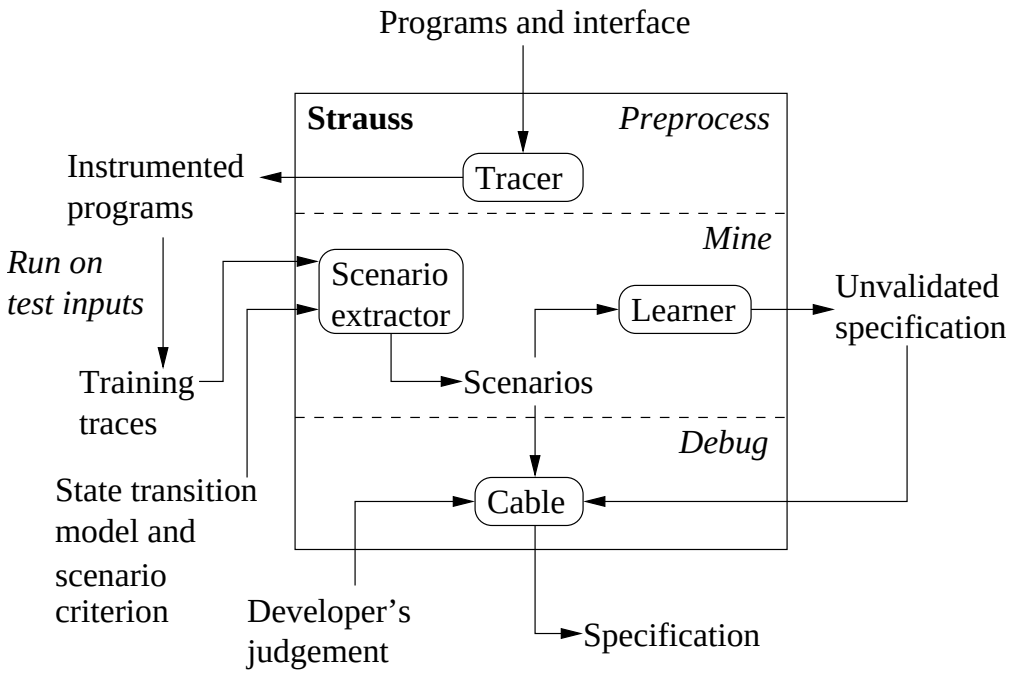

Figure 3: The architecture of Strauss and the three phases of mining with Strauss. Rounded boxes indicate parts of Strauss.

important. The file must be in a certain state when `unlock` is called: specifically, it must be locked. The call of `unlock` must come after the call of `lock`. Finally, the call to `unlock` and the call to `lock` must be for the *same* file.

Like all Strauss specifications, the specification in Figure 2 is *partial*. It does not capture all rules for correctly using `lock` and `unlock`: for instance, it may be illegal to call `lock(F)` twice in a row for the same file `F`.

Figure 3 shows the architecture of Strauss. As the figure shows, specification mining with Strauss has three phases: preprocessing, mining, and debugging. The rest of this section explains Figure 3 twice: first briefly and then in detail with an example.

The preprocessing phase instruments programs so that they record traces of calls to interface routines as they execute. For each call to an interface routine in a program execution, the trace of that execution has an *event* that records the call's argument and return values.

The specification developer gathers a set of training traces by running the instrumented programs on test inputs. This is a time-consuming step, but it is only done once, because the same training set can be used to mine many specifications, by varying other inputs to the miner.

The mining phase learns an initial specification, given the traces and two other inputs: a *state-transition model* (STM) and a *scenario criterion*. Both the STM and the scenario criterion can be written by a specification developer or generated automatically.

The state-transition model focuses Strauss on a particular aspect of the interface and models the manner in which calls to interface routines interact with one another. For example, the C standard library has routines for manipulating files and routines for manipulating strings; one STM would focus Strauss on the file routines, while another would focus Strauss on the string routines.

To process large traces quickly, Strauss does not learn a specification directly from the training traces. Instead, Strauss extracts *scenarios* from the traces, and learns from those. Scenarios are short sequences of interface calls that are related according to the STM. The scenario criterion determines which scenarios Strauss extracts.

Strauss's learner learns a specification from the scenarios: the specification accepts the extracted scenarios and other, similar scenarios.

```
type file
file new-file()
lock(in file f)
unlock(in file f)
string read(in file f)
write(in file f, in string s)
```

Figure 4: Interface for the running example.

```
Routine main()
  file f0 = new-file()
  file f1 = new-file()
  lock(f0)
  write(f0, ‘‘Hello’’)
  write(f1, ‘‘Goodbye’’)
  lock(f1)
  print ‘‘Hello, world!’’
  string s = read(f1)
  unlock(f1)
  unlock(f0)
```

Figure 5: Program for the running example.

If the input traces contain errors, Strauss might learn an erroneous specification. In the debugging phase, the specification developer uses the Cable specification debugger to correct and validate a specification.

The rest of this section explains Figure 3 in detail, using the interface in Figure 4 and the simple program in Figure 5 as a running example.

```
0x10 = new-file()
0x20 = new-file()
lock(f = 0x10)
write(f = 0x10, s = 0x80)
write(f = 0x20, s = 0x90)
lock(f = 0x20)
0x100 = read(f = 0x20)
unlock(f = 0x20)
unlock(f = 0x10)
```

Figure 6: Trace for the running example.

1.2.1 Preprocessing

The preprocessing phase in Figure 3 gathers traces for a particular interface, chosen by the specification developer. The specification developer uses Strauss's tracer to instrument programs so that, as a program runs, it writes a trace of its calls to interface routines to a trace file. For each call to an interface routine, the trace file has an *event* that contains the name of the routine, the values passed to the routine by the program, and the value or values returned to the program by the routine.

The developer gathers traces by running the instrumented programs on test inputs. Trace gathering is probably the most expensive and tedious step in Strauss, because running an instrumented program can take a long time. Fortunately, once traces are gathered, they can be used to mine many specifications.

Figure 6 shows the trace generated by our example program. Note that the trace contains the run-time values of parameters to interface routines.

The example program is straight-line code and takes no input, so it always generates the same trace. Most programs have branches and loops, and will generate

Value	Hidden state
0x10	?
0x20	?
0x80	?
0x90	?
0x100	?

Figure 7: Library state for the running example. The state is a table, indexed by interface values files and strings.

different traces for different inputs.

1.2.2 State-transition models

The specification that Strauss mines depends on the traces and on an STM. Many interfaces are too large to understand all at once: STMs, which can be hand-written or automatically generated, are a mechanism for focusing Strauss on a particular aspect of an interface.

Strauss does not depend on any knowledge of a library's code. Instead, Strauss works under the assumption that the library can be modeled in the following simple way. In the model, the library's state is stored in a table, which is indexed by interface values. In the running example, the table is indexed by the values of files and strings (Figure 7). The state in each row of the table is *hidden*: Strauss has no access to its representation. I call the rows of the table *state variables*.

Strauss interprets each trace as a *semantic trace*, in which trace events are viewed as reading from and writing to the table of hidden states; the STM determines the details of the interpretation. No matter which STM is used, the model imposes this restriction: an event can only read and write the state variables for the values it contains.

```
lock(def use f)
unlock(def use f)
```

Figure 8: A state-transition model for the running example.

`0x10 = new-file()`	() := ()
`0x20 = new-file()`	() := ()
`lock(f = 0x10)`	(State[0x10]) := (⍰(`lock`, { (`f`, State[0x10]) }))
`write(f = 0x10)`	() := ()
`write(f = 0x20)`	() := ()
`lock(f = 0x20)`	(State[0x20]) := (⍰(`lock`, { (`f`, State[0x20]) }))
`read(f = 0x20)`	() := ()
`unlock(f = 0x20)`	(State[0x20]) := (⍰(`unlock`, { (`f`, State[0x20]) }))
`unlock(f = 0x10)`	(State[0x10]) := (⍰(`unlock`, { (`f`, State[0x10]) }))

Figure 9: The trace from Figure 6 and pseudocode for its semantic-trace interpretation with respect to the STM in Figure 8. The function ⍰ models a computation of a new state by the library.

Given an STM, Strauss connects semantic-trace events with flow dependences from state-variable definitions to their uses. When Strauss's scenario extractor extracts scenarios, it uses flow dependences to decide which events and values to consider together, which to consider separately, and which to ignore completely.

An example will make this discussion concrete. Figure 8 shows an STM for the running example, and Figure 9 shows the trace from Figure 6, together with pseudocode for its semantic-trace interpretation with respect to the STM in Figure 8. Note that the pseudocode has an assignment statement for each event, and a state variable for each file value in the trace. The assignments use a special function, ⍰, which models the hidden computations of the library. This function will be discussed further below.

The STM in Figure 8 places `def` and `use` keywords in front of the `f` argument to

both `lock` and `unlock`. Strauss uses the `def` keywords to construct the left-hand side of assignment statements, and the `use` keywords to construct the right-hand side.

A `def` keyword in front of an argument indicates that an event assigns to the state variable associated with the argument's value. Therefore, in Figure 9, calls to `lock` and `unlock` assign a new state to the variable associated with their `f` argument. These assignments represent a change in the library's hidden state. My presentation of this example suggests that this change has to do with whether the file is locked or not, but Strauss does not know (or need to know) any such details.

A `use` keyword in front of an argument indicates that the effect of an event depends on the contents of the state variable associated with the argument's value. In Figure 9, the expressions on the right-hand side of the assignments for calls to `lock` and `unlock` involve the state variable associated with their `f` argument. This models the fact that the effect of a `lock` or `unlock` call varies, depending on whether the program holds the lock for the file or not.

The expressions on the right-hand sides of the assignments in Figure 9 are applications of a function ⍰. This function models the library's hidden computations. Because Strauss assumes that the library's code is unavailable, ⍰ cannot be defined.

However, the ⍰ applications do have meaning: they represent conjectures about the flow of the library's state between events. By treating ⍰ applications as if they were applications of a real function, Strauss can connect events by data dependences. For example, in Figure 9, there would be a flow dependence from each `lock` call to the following `unlock` call on the same file, but no dependences

between calls on different files. As the next section discusses, Strauss uses these dependences to extract two independent scenarios, one for each file.

It is important to understand that the STM in Figure 9 does not define the semantics of the library. Indeed, it is impossible to define a library's semantics without looking at the library's code! An STM simply determines what scenarios Strauss will extract from traces. An STM can be far removed from a true model of the library, and still be "good" if Strauss can use it to find simple and consistent scenarios.

1.2.3 Extracting scenarios

Strauss uses the semantic trace's dependences to extract scenarios from the training traces. A scenario is a short sequence of events from a training trace that, according to the dependences, interact with one another through assignments to state variables.

Strauss starts scenario extraction by asking the user for some parameters: a *seed pattern*, a *forwards-radius*, and a *backwards-radius*. The seed pattern matches events in the training traces, and Strauss extracts one scenario around each event that matches. Strauss grows a scenario by looking forwards and backwards from the seed for related events: the forwards-radius says how far forwards to look, and the backwards-radius says how far backwards to look.

Figure 10 shows the scenarios that Strauss extracts from the trace in Figure 6, as interpreted with the STM in Figure 8. To extract these scenarios, I used a seed

```
lock(r = 0x10); unlock(r = 0x10) [seed]

lock(r = 0x20); unlock(r = 0x20) [seed]
```

Figure 10: Two scenarios from the trace in Figure 6, as interpreted with the STM in Figure 8.

pattern that matches all calls to `unlock`, a forwards-radius of 0, and a backwards-radius of 1. There are two scenarios because there are two events for calls to `unlock` in Figure 6, one for file `0x10` and one for file `0x20`. For each event, Strauss found the corresponding `lock` and added it to the scenario. Note that Strauss keeps track of which event is the seed in a scenario.

1.2.4 Learning

Learning is inductive inference: given some observations, the learner's goal is to find an explanation that accepts those observations and also accepts new observations. The learning step in Strauss finds a specification that accepts the scenarios observed in a set of training traces and accepts scenarios observed in new traces.

Strauss specifications have three parts: the seed pattern, the state transition model, and a scenario acceptor that accepts scenarios. The acceptor contains a nondeterministic finite automaton (NFA), but it is not itself an NFA, because NFAs accept strings, not scenarios. Instead, Strauss defines a standard string form for each scenario, which abstracts away petty differences of event ordering or the precise identity of values; a scenario acceptor accepts a scenario iff the standard string for the scenario is accepted by the acceptor's NFA.

Figure 2 shows the specification that Strauss would mine for the running example. Here is how Strauss interprets this specification:

> Suppose that P is a correct program. Then, in every trace of P (as interpreted with the specification's STM), every event that matches the seed pattern (that is, every `unlock` call) must be part of a scenario that matches the expression:
>
> ```
> lock(f = F); unlock(f = F) [seed]
> ```
>
> That is, the `unlock` call must be preceded by a `lock` call on the same file and with no intervening `unlock` calls on that file.

The idea is that taking the viewpoint of the STM led Strauss to extract certain scenarios from the training traces, so it is reasonable to expect that taking the same viewpoint will allow Strauss to extract similar scenarios from new traces.

1.2.5 Debugging

Strauss's learner assumes that the training scenarios are correct. In fact, the training traces might have errors, so some training scenarios might be erroneous. Strauss includes an interactive specification debugger, called Cable, that makes it easy to find erroneous scenarios and remove them from the specification.

Cable is an essential part of Strauss, but it is not limited to debugging Strauss specifications. In fact, its methods apply to debugging any temporal specification.

The specification for the running example (Figure 2) is correct, so there is no need to debug it.

1.3 An overview of the dissertation

This dissertation describes the design and implementation of Strauss, a system for mining temporal specifications for interfaces from program traces.

Chapter 2 defines some basic notation that is used throughout the dissertation, including notation for traces and specifications.

Chapter 3 gives the formal syntax and semantics of state-transition models and semantic traces.

Chapter 4 describes how Strauss extracts scenarios from semantic traces, and how the choice of state-transition model affects which scenarios are extracted.

Chapter 5 tells how Strauss learns specifications from scenarios. The main challenge is reducing the problem of learning an acceptor of scenarios to the well-known problem of learning an NFA that recognizes strings. This chapter describes this reduction and the off-the-shelf NFA learner that Strauss uses.

Recording the right routines and data in traces is essential for specification mining. Chapter 6 explains how I decided what to record in traces of the X11 libraries. That chapter also explains how I instrumented X11 applications to record those traces; this task was complicated because the applications and libraries are written in C and C++, which are not type-safe languages, and because the X11 interfaces make extensive use of callbacks from library to application code.

Strauss can produce buggy specifications, because traces and hence scenarios can have bugs. Chapter 7 describes a novel specification debugger for temporal specifications, including Strauss's specifications. The debugger finds similarities

within a set of training scenarios and clusters similar scenarios together; by examining summaries of these clusters instead of individual scenarios, the specification developer can take all of the scenarios into consideration without actually inspecting every scenario.

In addition to Strauss, I have implemented a checker for Strauss specifications. Chapter 8 describes this checker, which finds specification violations dynamically, in traces.

Chapter 9 discusses my experiments with Strauss. I mined 17 specifications for programs that use the X11 interface, and found 199 bugs. This chapter presents those results, and also explains how I found these specifications. Picking good state transition models was key, so I explain how to do that by hand and with an automatic tool that analyzes trace statistics.

Finally, in Chapter 10, I reflect on my experience with Strauss, discuss alternative solutions to the specification problem, and propose directions for future work in specification mining.

1.4 Related work

1.4.1 Other ways to develop specifications

Specification miners are tools for developing specifications, for the purpose of finding program errors. This section discusses the existing alternatives.

Programming by refinement

David Gries argued in an influential textbook that, by writing specifications first and then refining the specifications into executable programs, we can program so well that "finding an error should be the exception rather than the rule" [25]. Similar techniques have been applied to many real systems (for example, an implementation of the I/O subsystem of the GCOS operating system [48]; an implementation of a secure certification authority for smart cards [26]; and a formalization of the IEEE 754 standard for binary floating-point arithmetic [3]), but they are hardly ubiquitous. At least part of the reason is that programmers must develop programs in a completely new way to use these refinement methods. On the other hand, refinement methods are very powerful: if the original specification is correct and the refinement tools are correct, the resulting program is guaranteed to be correct.

Annotating source code

Many tools let developers add specifications to their code, as annotations. Tools that depend on this approach include Extended Static Checking [14], its successor ESC/Java [34], and LCLint [18] (now known as Splint). A disadvantage of these tools is that they require developers to change program source code; on the plus side, developers can help the program-verification tool prove a correctness property by adding annotations for program-specific properties, such as loop invariants.

In particular, the Bandera Specification Language [29, 9] encourages programmers to write temporal specifications and the program annotations necessary to

verify them at the same time. To make this easier, Bandera allows developers to write specifications and annotations in a high-level language that is similar to Java. Also, because Bandera's authors believe that most specifications are instances of a small number of specification patterns [15] (this is true for the specifications I found with Strauss—see Appendix B), Bandera provides special support for tailoring common specification patterns to a particular specification and/or program.

Types

Some languages come with annotation languages "built-in", as a type system. Type systems have been widely adopted, largely because programmers find them convenient for structuring code and data, and partly because they allow compilers to find bugs statically. Although type systems are limited in the kinds of properties they can express, this situation is improving. For example, Vault [12] allows the expression of temporal properties (similar to those mined by Strauss) within a type system that is practical to check.

Separate specifications

The least intrusive verification tools require no changes to program source code. For example, in the `Xgcc` system [27], developers write specifications apart from program code, in the `metal` specification language. Specifications written in `metal` define a state machine and patterns that map program code constructs to state-machine transitions, so no annotations are necessary. However, `Xgcc` is neither sound nor complete: it can report errors falsely and miss real errors. In principle,

these problems could be mitigated with program annotations, but the `Xgcc` philosophy is to sort error reports heuristically so that the user tends to see true reports before false ones.

There are sound tools in this category. SLAM [2] is a model checker for device drivers and other system software; this tool aims to check small programs very thoroughly. Another sound tool, ESP [11], focuses on larger code bases.

1.4.2 Specification mining

Previous work

Ernst and others also proposed automatic deduction of formal specifications [17]. Their Daikon tool works by learning likely invariants involving program variables from dynamic traces. The form of the resulting formal specifications is the key difference between Daikon and Strauss. Daikon's specifications are arithmetic relationships that hold at *specific* program points (e.g., a precondition $x < y$ at entry to a procedure f). By contrast, Strauss specifications express temporal and data-dependence relationships among interface events. Strauss's temporal specifications capture a different aspect of program behavior than Daikon's predicates on values and structures. The two forms of specifications are complementary, but naturally require radically different learning algorithms.

A related tool is Houdini [19], an annotation assistant for ESC/Java. Starting from an initial (guessed) candidate set of annotations, which are similar to those of Daikon, Houdini uses ESC/Java to refute invalid annotations. The focus of Houdini is on annotating points of a single program with true properties, while the

focus of Strauss is on discovering temporal properties that hold across all programs that use an interface.

Several authors describe tools that extract automaton-based models. Cook and Wolf describe a tool for extracting FA models of software-development processes from traces of events [8]. Strauss differs in that it extracts specifications from program traces, which must be reduced to a simpler form before they are palatable for an FA learner.

Ghosh and others describe several techniques for learning the typical behavior of programs that make system calls [22, 35]. Since they intend their models for intrusion detection, the models need only characterize a particular program's behavior, while Strauss finds rules that are generally applicable and understandable by humans.

Wagner and Dean's intrusion-detection system also extracts automaton models, but from source code, not traces [51]. Their system also extracts models that apply only to a single program.

Reiss and Renieris also extract structure from traces [44], but they model the sequence of operations on individual objects, not the data and temporal dependences across several objects.

Finally, Boigelot and Godefroid extract specifications for reactive programs from traces of their behavior [5]. Their specifications model the sequence of all operations at a process that are visible to other processes. By contrast, Strauss assumes that interaction traces contain events that should not be considered together, such as locking operations on different locks. To be effective, Strauss must decide which events and values to consider together, which to consider separately,

and which to ignore completely.

Concurrent and later work

The DIDUCE debugging aid [28] infers an approximation of the set of values taken on by expressions. DIDUCE finds these simple invariants on-line (as a program runs) and reports invariant violations as they occur. The authors argue that this approach helps developers to identify and understand rare "corner cases". By contrast, Strauss mines specifications from a larger space.

Hallem and others propose belief analysis [27], which looks for occurrences of a fixed set of rule templates in the program text, formalizing a rule by instantiating a template. Belief analysis infers temporal rules that instantiate simple schemas, such as "a call to routine `<a>` must be paired with a call to routine `<b>`". Again, Strauss mines more general specifications.

Whaley and others give dynamic and static techniques for extracting simple finite automata that describe Java classes [54]. Their automata are more general than those found by Hallem and others, but still more restrictive than those mined by Strauss. Also, like the methods of Reiss and Renieris, their techniques cannot learn specifications that involve more than one value.

1.4.3 Specification debugging

The debugging method described in this dissertation fills a large hole left unexplored in a previous paper by myself and others 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 in

the scenario acceptor's NFA—for removing errors from those specifications. This dissertation gives a general method for debugging specifications, which applies not only to Strauss 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 [27], 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 [6], uses a number of filtering and ranking heuristics to reduce what they call "noise". By contrast, my specification debugger clusters similar bug reports together but does not rank the clusters. In my 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 [16], a tool for dynamically discovering arithmetic invariants, uses statistical confidence checks to suppress invariants that appear to have occurred by chance. In the case of Strauss, I found that some buggy scenarios occurred so frequently that using a similar approach to suppress them would also suppress valid scenarios.

One way to debug specifications is to assume that any part of a specification that cannot be verified by a program-verification tool is, in fact, wrong. This approach is used by the Houdini tool [19], which guesses many invariants and then uses ESC/Java [34] to prune out those that do not always hold. A similar approach was used to integrate the Daikon and ESC/Java tools [38]. 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.

Chapter 2

Preliminaries

This chapter defines terminology and notation, including the syntax of traces (the input to Strauss) and the syntax and semantics of specifications (the output of Strauss).

2.1 Lists, tuples, and records

A list is a sequence of elements, written $[e_0, e_1, \ldots, e_n]$. If l_0 and l_1 are lists, then $l_0 \cdot l_1$ is their catenation. If e is an element and l is a list, then $e :: l$ is the list with e prepended to l. I use lists where the sequence may be of any length.

A tuple is also a sequence of elements, written $(e_0, e_1, \ldots, e_n)$. I use tuples where the sequence is understood to be of some fixed length.

For any sequence (list or tuple) s and non-negative integer i, $s[i]$ is the ith element of s, if it exists.

A record is a set of named elements, written $\{f_0 : e_0, f_1 : e_1, \ldots, f_n : e_n\}$. The names of a record can be used as functions that access the record's elements: if r is a record that contains a named element $f_i : e_i$, then $f_i(r) = e_i$.

2.2 Finite automata

An *nondeterministic finite automaton (NFA)* is a tuple $(\Sigma, Q, q_s, q_f, \delta)$ where

- Σ is an alphabet.
- Q is a set of states.
- $q_s \in Q$ is the start state of the automaton.
- $q_f \in Q$ is the final state of the automaton.
- The transition function $\delta(q, a)$ is a partial function from $Q \times \Sigma$ to 2^Q. If $q' \in \delta(q, a)$, then we say that the NFA has a transition from q to q' on a.

Following convention, I sometimes use δ to mean a transition function from $Q \times \Sigma^*$ to 2^Q, where Σ^* is the set of finite strings over Σ. For $w = [a_0, \ldots, a_n] \in \Sigma^*$, $q' \in \delta(q, w)$ iff there is a sequence of states $s = [q = q_0, \ldots, q' = q_{n+1}]$ such that $q_{i+1} \in \delta(q_i, a_i)$, for all $0 \leq i \leq n$. In this case, we say that s *takes* q *to* q' *on* w.

I say that the NFA *accepts* w iff $q_f \in \delta(q_s, w)$, and that the NFA *accepts* w *from* q iff $q_f \in \delta(q, w)$. A sequence of states $s = [q_0, \ldots, q_n]$ accepts w iff s takes q_s to q_f on w.

A *deterministic finite automaton (DFA)* is an NFA where $|\delta(q, a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$.

A *probabilistic finite-state automaton* is a tuple $(\Sigma, Q, q_s, q_f, \delta, W)$ where

- Σ, Q, q_s, q_f, and δ have the same meaning as they have for NFAs, except that q_f has no outgoing transitions. That is, $|\delta(q_f, a)| = 0$ for all $a \in \Sigma$.

- The weighting function W is a partial function from $Q \times Q$ to the positive integers.

A PFSA can be interpreted as an acceptor of strings: I say that a PFSA accepts a string w iff the NFA (Σ, Q, q_s, q_f) accepts w. That is, W is ignored.

A PFSA can also be interpreted as a probability distribution over Σ^*, where W determines the probabilities. For $w \in \Sigma^*$, the probability $p(w)$ of w is given by

$$p(w) = \sum_{\substack{s = [q_0, \ldots, q_n] \\ s \text{ accepts } w}} \prod_{i=0}^{n-1} \frac{W(q_i, q_{i+1})}{\sum_{q' \in Q} W(q_i, q')}$$

2.3 Traces

2.3.1 Concrete trace syntax

Chapter 1 said that traces record the calls that programs make to library routines, along with their arguments and return values. In fact, Strauss's traces are slightly more expressive. Many libraries, such as the X11 libraries I used in my experiments, also allow the library to call routines in the program. Therefore, traces record three kinds of *events*: *call events* from the program to library routines; *callback events* from the library to program routines; and *return events*, which return control from a library routine back to the program or from a program routine back to the library.

Figure 11 lists the syntax of these events. The three productions for *event* generate call events, callback events, and return events, respectively. Each call event has a call-site (which identifies the point in the program from which the call

```
event       → call-site : call name ( args )
            |  callee : callback name ( args )
            |  return name ( args )
args        → ϵ
            |  some-args
some-args   → arg
            |  arg , some-args
arg         → name = value
call-site   → Id
callee      → Id
name        → Id
value       → Integer
            |  Integer -> ( args )
```

Figure 11: Syntax of trace events. The second production for *value* produces a value with an attached tuple of arguments: the arrow notation is meant to suggest a reference, not a function.

was made), the name of the called routine, and a tuple of arguments passed to the called routine. Each callback event has a callee (which identifis the program routine that is being called), a name that indicates the type of the callback routine (*not* the name of the routine, as that will vary from program to program), and a tuple of arguments passed to the callback routine. Finally, each return event has a name (which matches the name of the corresponding call or callback event), and a (possibly empty) tuple of arguments returned with the event. In many languages, the return value of a routine is unnamed; by convention, in a trace, the argument for such a value is named "?".

Note that the call-site and callee fields are useful only for relating events back to source code. Consequently, examples in this dissertation often omit those fields.

Argument tuples may be empty. All arguments have a name (an identifier)

```
Trace     = Event sequence
Event     = Call (Name, Arg set)
          | Callback (Name, Arg set)
          | Return (Name, Arg set)
Arg       = (APath, Value)
Name      = Id
APath     = Id
Value     = Integer
```

Figure 12: Abstract syntax for traces.

and a value (an integer). Integral values are adequate for representing the numeric types and pointer types that appear in this dissertation; the syntax could easily be extended to handle other types, if necessary.

In addition to its value, an argument may have an attached tuple of arguments, called its *contents*; the second production for *value* in Figure 11 produces arguments with contents. Argument contents are used to record the fields of structures and the elements of arrays. I assume that all structures and arrays are allocated on the heap or stack and passed by reference: the value of such arguments is the address of the structure or array in memory, and the contents of such arguments is the tuple of field names and their values (or the size of the array and a tuple of its elements, for arrays). If structures or arrays were passed by value, there would be no sensible address to use for the argument value. In that case, a dummy value such as zero could be used.

2.3.2 Abstract Trace Syntax

Traces written in the concrete syntax of Figure 11 are easy to read, but it is easier to express analyses and transformations in terms of an abstract syntax. Figure 12 lists an abstract syntax for traces. It differs from a straightforward transcription of the concrete syntax of Figure 11 in two ways:

- Call events do not include the call-site and callback events do not include the callee. Actually, my implementation retains these parameters for use as documentation, but I omit them here because they are not needed in this dissertation.

- Arguments are flattened and unordered. In the abstract syntax, all arguments, including arguments that are contained by other arguments, appear in a single unordered set. The original nesting structure is indicated by pairing each argument value with an *access path* instead of a name.

  An access path bundles a sequence of identifiers that identify the path to an argument in the concrete syntax into a single identifer of the form Id0 . Id1 . ... . Idn. Id0 identifies the type of the event (`call`, `callback`, or `return`), Id1 is the name of the event, Id2 is the name of the top-level argument, Id3 is the name of an argument contained within the top-level argument, and so on.

  For example, this concrete call:

  ```
  0x0:  call foo(x = 1 -> (y = 2))
  ```

  would be written abstractly as

Call (`foo`, { (`call.foo.x`, 1), (`call.foo.x.y`, 2) })

Notice that the nesting structure of the concrete call, where the argument `x` contains the argument `y`, is indicated via access paths in the abstract call.

2.3.3 Functions on events

I make use of the following functions on events (note that I use the nonterminals in Figure 12 as types here):

- Call: Event → Bool

  True iff an event is a call.

- Callback: Event → Bool

  True iff an event is a callback.

- Return: Event → Bool

  True iff an event is a return.

- Type: Event → {`call`, `callback`, `return`}

  Returns the type of an event.

- Name: Event → Name

  This accesses the event's name.

- Args: Event → Arg set

  This accesses an event's arguments.

- Path: Arg → APath

  This accesses the access-path component of an event argument.

For all trace events matching

```
call unlock(r = R)
```

with STM

```
call lock(def use r)
call unlock(def use r)
```

a scenario matches

```
call lock(r = R); call unlock(r = R) [seed]
```

Figure 13: A positive specification, which says "locks released with `unlock` must have been acquired with `lock`."

- Value: Arg → Value

  This accesses the value component of an event argument.

2.4 Specifications

2.4.1 Positive specifications

A positive specification dictates that programs must do something. Figure 13 shows an example, which says "locks released with `unlock` must have been acquired with `lock`."

Figure 14 is a general template for positive specifications. The template has three parts:

- A seed pattern. Figure 15 lists the concrete syntax of seed patterns, and Figure 16 lists an abstract syntax. The latter is identical to the abstract syntax for events in Figure 12, except that values in seed patterns are identifiers, not integers. Because the two syntaxes are so similar, the functions

For all trace events matching

seed pattern

with STM

state-transition model

a scenario matches

scenario acceptor

Figure 14: A general template for positive specifications.

in Section 2.3.3 can be extended in a natural way to seed patterns.

An event e matches a seed pattern pat iff

- Type(e) = Type(pat); and
- Name(e) = Name(pat); and
- there is a one-to-one map B from identifiers to integers such that for all $a_{pat} \in$ Args(pat), there is an $a_e \in$ Args(e) such that Path(a_{pat}) = Path(a_e) and B(Value(a_{pat})) = Value(a_e).

For example, the event

```
0x0:  call foo(w = 0, x = 1 -> (y = 2, z = 2))
```

matches the seed pattern

```
call foo(x -> (y = X, z = X))
```

because the type of both pattern and event is `call`, the name of both pattern and event is `foo`, and each argument in the pattern matches an argument

seed-pattern → `call` *name* `(` *args* `)`
| `callback` *name* `(` *args* `)`
| `return` *name* `(` *args* `)`
args → ϵ
| *some-args*
some-args → *arg*
| *arg* `,` *some-args*
arg → *name* `=` *value*
| *name* `->` `(` *args* `)`
name → `Id`
value → `Id`
| `Id` `->` `(` *args* `)`

Figure 15: Concrete syntax of seed patterns.

Event = Call (Name, Arg set)
| Callback (Name, Arg set)
| Return (Name, Arg set)
Arg = (APath, Value)
Name = Id
APath = Id
Value = Id

Figure 16: Abstract syntax for seed patterns.

in the event when `X` is mapped to 1. Notice that the event matches the pattern despite the fact that the event has an argument named `w` while the seed pattern does not.

- A state-transition model (STM). I defer a detailed discussion of state-transition models to Chapters 3 and 4. Those chapters define a function, which I need to define the semantics of specifications:

$$\text{Scenarios: Trace} \times \text{STM} \times \text{Event} \rightarrow \text{Scenario set}$$

Given a trace T, a STM M, and a seed event e, Scenarios(T, M, e) is the set of all scenarios seeded by e in T, when T is interpreted with M. A scenario is a small set of trace events; the precise definition is in Chapter 4.

- A scenario acceptor (SA). I defer a detailed discussion of scenario acceptors to Chapter 5. For now, it is enough to know that a SA accepts some scenarios and rejects others. Chapter 5 defines a function

  Language: SA $\rightarrow$ Scenario set

  Given a scenario acceptor A, Language(A) is the set of all scenarios accepted by A.

The semantics of positive specifications is as follows. Let *spec* be a positive specification with seed pattern *pat*, state-transition model M, and scenario acceptor A. Let P be a program. I say that *spec accepts* P (or P *obeys spec*) iff, for any trace T of P and event $e \in T$ such that e matches *pat*,

$$\text{Scenarios}(T, M, e) \cap \text{Language}(A)$$

is not empty.

2.4.2 Negative specifications

A negative specification dictates that programs must *not* do something. Figure 17 shows an example, which says "locks must not be released twice in a row with `unlock`."'

Figure 18 is a general template for negative specifications. The template is very similar to the template for positive specifications (Figure 14). The only difference

For all trace events matching

```
call unlock(r = R)
```

with STM

```
call lock(def use r)
call unlock(def use r)
```

no scenario matches

```
call unlock(r = R); call unlock(r = R) [seed]
```

Figure 17: A negative specification, which says "locks must not be released twice in a row with `unlock`."

For all trace events matching

seed pattern

with STM

state-transition model

no scenario matches

scenario acceptor

Figure 18: A general template for negative specifications.

is that negative specifications replace the phrase "a scenario matches" with the phrase "no scenario matches".

There is a corresponding difference between the semantics of the two kinds of specifications. The semantics of negative specifications is as follows. Let *spec* be a negative specification with seed pattern *pat*, state-transition model M, and scenario automaton A. Let P be a program. I say that *spec accepts* P (or P *obeys spec*) iff, for any trace T of P and event $e \in T$ such that e matches *pat*,

$$\text{Scenarios}(T, M, e) \cap \text{Language}(A)$$

is empty.

That is, the semantics of negative specifications is the same as the semantics of positive specifications, except that the phrase “is not empty” is replaced with the phrase “is empty”.

Chapter 3

Semantic traces

A Strauss specification defines some aspect of correct program behavior. Strauss mines a specification from traces, which record events (calls, callbacks, and returns) that occur at the interface between a library and its client programs. Because traces intermix independent events, Strauss cannot find a specification for related events without knowing how events affect one another. In addition, how events affect one another depends on which aspect of program behavior is under consideration. Put another way, Strauss needs a flexible mechanism for assigning meanings to traces.

Giving meanings to traces is challenging, because I assume that program and library code might be unavailable or difficult to analyze. This is a necessary assumption in practice, for many kinds of software systems:

- distributed systems that are composed from programs that are owned by different people;
- systems that use libraries that are written in multiple languages or distributed without source code;
- systems that use scripts to compose many different programs;
- systems that include other elements that are hard to analyze, like people or devices that interact with the physical world.

```
return new-file (def ?)
call lock (def use f)
call unlock (def use f)
```

Figure 19: A state-transition model. This STM associates a `def` effect with the return argument of `new-file`, and `def` and `use` effects with the `f` argument of `lock` and `unlock`. According to this STM, each return from `new-file` assigns to the state variable for the return value, and each call of `lock` or `unlock` reads from and assigns to the state variable for the value passed as `f`.

Without access to code, Strauss must use external models to define the semantics of traces. This chapter describes these models, which I call *state-transition models (STMs)*. Figure 19 shows an example. Essentially, an STM associates tokens called *effects* with event arguments; the meaning of these tokens will be explained shortly. For example, the STM in Figure 19 associates a `def` effect with the return argument of `new-file`, and `def` and `use` effects with the `f` argument of `lock` and `unlock`.

Given an STM (supplied by the user), Strauss gives meaning to a trace by treating it as a special kind of program, called a *semantic trace*. Intuitively (although not formally), a semantic trace is a sequence of assignment statements, which assign to state variables.

State variables hold abstract states, where each abstract state represents some part of the library's hidden state. There is a one-to-one correspondence between state variables and trace values.

Each trace event corresponds to an assignment statement, which depends on the STM and on the values of the event's arguments. The left side of such a statement is a tuple of state variables, and the right side is a tuple of abstract expressions,

which compute abstract states and represent the library's hidden computations. A state variable appears in the tuple on the left side iff its associated value occurs at an argument that the STM associates with a `def` effect. A state variable appears in the expressions on the right side iff its associated value occurs at an argument that the STM associates with a `use` effect. So, according to the STM in Figure 19, each return from `new-file` assigns to the state variable for the return value, and each call of `lock` or `unlock` reads from and assigns to the state variable for the value passed as `f`.

Semantic traces and STMs have several useful properties:

- Semantic traces are easy to understand because variables and assignments to variables are familiar concepts to the intended users of Strauss.
- STMs are easy to write; useful STMs can be written in a few lines of code, and the specification developer can easily try out many interpretations of semantic traces.
- STMs are general. STMs can be used to model any interface where
  - The library keeps state, and
  - At least some of the library's state is indexed by the values that occur in events.
- As I demonstrate in Chapter 9, STMs are simple enough to construct automatically from statistics about traces.

Of course, semantic traces are not really programs. In a sense, semantic traces are to programs as formal power series are to power series in mathematics. A formal

power series is essentially an infinite sequence written with the notation of power series; power-series notation is convenient because it helps to organize analyses and manipulations of the infinite sequence. Similarly, a semantic trace is essentially an interpretation of a trace written with the notation of programs; program notation is convenient because it helps to describe analyses and transformations of the trace interpretation.

Strauss uses a number of such analyses and transformations. For example, Chapter 4 explains how to extract scenarios from a semantic trace by slicing [53] and chopping [32] in the semantic trace's program dependence graph. Also, Section 3.3 of this chapter presents three useful transformations of semantic traces. Two of these transformations simplify semantic traces while preserving meaning, just as optimizing compilers transform programs to make them run faster while still computing the same results. For example, a common program transformation drops unused program variables; the analogous transformation of semantic traces drops unused arguments. The third transformation (the flags transformation) alters the meaning of semantic traces so that Strauss can mine more expressive specifications. In particular, the flags transformation allows Strauss to learn sensible specifications about routines like the C library routine `fopen`, whose behavior is modified by arguments that act as flags.

Overview of the Chapter

The rest of this chapter is structured as follows. Section 3.1 shows how to use the STM in Figure 19 to interpret a trace as a semantic trace. Section 3.2 gives

the syntax of STMs and the semantics of traces in terms of STMs. Finally, Section 3.3 describes transformations of semantic traces that improve efficiency and allow Strauss to mine more expressive specifications.

```
1   Call (new-file, {})
2   Return (new-file, { (return.new-file.?, 0x10) } )
3   Call (new-file, {})
4   Return (new-file, { (return.new-file.?, 0x20) } )
5   Call (lock, { (call.lock.f, 0x10) } )
6   Return (lock, {})
7   Call (write, { (call.write.f, 0x10) } )
8   Return (write, {})
9   Call (write, { (call.write.f, 0x20) } )
10  Return (write, {})
11  Call (lock, { (call.lock.f, 0x20) } )
12  Return (lock, {})
13  Call (read, { (call.read.f, 0x20) } )
14  Return (read, {})
15  Call (unlock, { (call.unlock.f, 0x20) } )
16  Return (unlock, {})
17  Call (unlock, { (call.unlock.f, 0x10) } )
18  Return (unlock, {})
```

Figure 20: Example from Figure 6, written following the abstract syntax of Figure 12.

3.1 An example

The following example illustrates how STMs give meaning to semantic traces. Figure 20 shows the running example from Figure 6 (Chapter 1), written following the abstract syntax of Figure 12 (Chapter 2). Note that the abstract syntax groups the event arguments in a flat list, which simplifies the notation for semantic-trace semantics, analyses, and transformations.

The trace in Figure 20 records a sequence of events with an intuitive meaning. First, the program creates two files—the values `0x10` and `0x20`—by calling the

```
1     push ⓘ(new-file, top(stack), {})
2     State[0x10] := ⓘ(return.new-file.?, top(stack), {})
      pop
3     push ⓘ(new-file, top(stack), {})
4     State[0x20] := ⓘ(return.new-file.?, top(stack), {})
      pop
5     push ⓘ(lock, top(stack), { (call.lock.f, State[0x10] ) } )
      State[0x10] := ⓘ(call.lock.f, top(stack),
6                         { ( call.lock.f, State[0x10] ) } )
7     pop
8     push ⓘ(write, top(stack), {})
9     pop
10    push ⓘ(write, top(stack), {})
11    pop
      push ⓘ(lock, top(stack), { ( call.lock.f, State[0x20] ) } )
12    State[0x20] := ⓘ(call.lock.f, top(stack),
13                        { ( call.lock.f, State[0x20] ) } )
14    pop
15    push ⓘ(read, top(stack), {})
      pop
16    push ⓘ(unlock, top(stack), { ( call.unlock.f, State[0x20] ) } )
17    State[0x20] := ⓘ(call.unlock.f, top(stack),
                        { ( call.unlock.f, State[0x20] ) } )
18    pop
      push ⓘ(unlock, top(stack), { ( call.unlock.f, State[0x10] ) } )
      State[0x10] := ⓘ(call.unlock.f, top(stack),
                        { ( call.unlock.f, State[0x10] ) } )
      pop
```

Figure 21: Imperative pseudo-code for the example of Figure 20.

library routine `new-file`. Presumably, the library has somehow associated each of these values with a file, which has some state. Next, the program locks the file associated with `0x10` by calling the library routine `lock`. Presumably, the library updates the file's state, perhaps setting a bit to indicate that the file is locked and remembering the ID of the owner. The program performs some operations on both files and eventually locks the file associated with `0x20`. Again, the library sets a bit and records an owner, but this time in the state of the file associated with `0x20`, which is presumably independent of the state of the file associated with `0x10`. The program finishes up by unlocking both files, by calling the library routine `unlock`. Presumably, the library checks the state of each file (perhaps making sure that the locked bit is set and that the owner matches) and then updates the state (perhaps clearing the locked bit).

Now, Figure 21 gives imperative pseudocode for the trace in Figure 20, interpreted with the STM in Figure 19. The idea is that the pseudocode captures just enough of the intuitive meaning above to mine some specifications about locks, without getting bogged down in speculation about the detailed, hidden state that the library keeps for files.

The pseudocode has a state variable for each value in the trace: State[0x10] is the variable for `0x10` and State[0x20] is the variable for `0x20`. These variables hold abstract states, which model the hidden state in the library.

The code also keeps a stack of activation records to model calls and returns. The stack is not strictly necessary for a simple interface like this one, but it is necessary if the interface allows callbacks from the library to the program. In that case, a call may be separated from its return by an arbitrary number of events,

and the stack simply makes the connections between calls and returns explicit.

Events correspond to assignments to the stack and the state variables. For example, the call of `lock` on line 5 pushes a state onto the stack and assigns a new state to State[0x10]. The states are produced by calling ⍰, an unspecified function that models the library's hidden computations.

Note that the pseudocode in Figure 21 specifies inputs for ⍰. Since ⍰ is unspecified, this seems a bit odd: I am not saying how to compute the value of ⍰, so what good are inputs? Intuitively, these inputs express conjectures about which state the library inspects. For example, at the call of `lock` on line 5, I conjecture that the new state of State[0x10] depends on just three things:

- the fact that the value `0x10` occurred at the argument `lock.f`; and
- the states on the stack, which encode the calling context of the `lock` call; and
- the old state of State[0x10].

3.2 Semantics of semantic traces

This section formalizes STMs and semantic traces and presents a small-step operational semantics for semantic traces.

3.2.1 State-transition models

An STM is a list of *STM declarations*, where Figure 22 gives the concrete syntax of STM declarations. An STM associates four kinds of effects with event arguments:

```
stm-decl    → call name ( args )
            | callback name ( args )
            | return name ( args )
args        → ϵ
            | some-args
some-args   → arg
            | arg , some-args
arg         → effects name
            | effects name -> ( args )
effects     → ϵ
            | def effects
            | use effects
            | not-def effects
            | not-use effects
name        → Id
```

Figure 22: Grammar for STM declarations.

a ***def*** *effect* means that the event assigns a new state to the state variable for the argument's value; a ***use*** *effect* means that the event depends on the state variable's state; a ***not-def*** *effect* means that the event definitely does not assign a new state to the state variable; and a ***not-use*** *effect* means that the event definitely does not depend on the state variable's state. Note that `def` and `not-def` effects are incompatible, as are `use` and `not-use` effects.

Given a set of traces T, a *complete STM* with respect to T assigns either a `def` or `not-def` effect and either a `use` or `not-use` effect to every event argument in T. Although the semantics of semantic traces is defined only for complete STMs, Strauss does not expect the user to supply complete STMs. Instead, the user's STMs are completed in one of two ways. The *conservative* completion assigns a `not-def` effect to arguments with neither a `def` nor a `not-def` effect and a `not-use`

```
return new-file (def not-use ?)
call lock (def use f)
call unlock (def use f)
call read (not-def not-use f)
call write (not-def not-use f)
```

Figure 23: Conservative completion of the STM in Figure 19.

```
return new-file (def use ?)
call lock (def use f)
call unlock (def use f)
call read (def use f)
call write (def use f)
```

Figure 24: Liberal completion of the STM in Figure 19.

effect to arguments with neither a `use` nor a `not-use` effect. The *liberal* completion assigns a `def` effect to arguments with neither a `def` nor a `not-def` effect and a `use` effect to arguments with neither a `use` nor a `not-use` effect.[1]

Figure 23 shows the conservative completion of the STM in Figure 19 with respect to the trace in Figure 20, while Figure 24 shows the liberal completion.

Figure 25 gives an abstract syntax for STMs, and Figure 26 shows the STM of Figure 19 written with this syntax. I use the abstract syntax to express semantic-trace semantics and transformations. The main difference between the concrete and abstract syntaxes is that the abstract syntax collects the four kinds of effects into four sets of access paths, rather than peppering effects throughout a list of STM declarations. Note that I require that the intersection of Defs and NotDefs and the intersection of Uses and NotUses be empty.

[1] Usually, the conservative completion is used, but see Section 9.2.1 for a case study that shows how to use the liberal completion to construct an STM incrementally.

STM = (Defs, Uses, NotDefs, NotUses)
Defs = APath set
Uses = APath set
NotDefs = APath set
NotUses = APath set
APath = Id

Figure 25: Abstract syntax for STMs.

({`return.new-file.?`, `call.lock.f`, `call.unlock.f`},
{`call.lock.f`, `call.unlock.f`}, ∅, ∅)

Figure 26: The state-transition model of Figure 19, written with the abstract syntax of Figure 25.

3.2.2 Operational semantics of semantic traces

This section presents a small-step operational semantics for semantic traces. To define the semantics, I need to define semantic traces, semantic-trace state, and the small steps that transition from state to state. First, however, I define some types and functions used in these definitions:

- VState: Integer

  This is the type of states. State variables hold values of this type. The representation of this type is not important for the semantics, so I simply use integers.

- ZeroMap: Value $\mapsto$ VState

  This maps all values to the zero state.

- ⍰: (Name $\cup$ APath) $\times$ VState $\times$ (APath $\times$ VState) set $\rightarrow$ VState

This unspecified function models the hidden computations of the library. The first argument is either the name of an interface routine or an access path to an event argument and indicates the target of an assignment; if the argument is a name, the target is the stack. The second argument is the state at the top of the stack. The third argument is a set of pairs of access paths to arguments and the state of the associated state variable. ⍰ returns a state.

As discussed in Section 3.1, the inputs to ⍰ express conjectures about which state the library inspects when it computes a new state.

The semantics also makes use of the types and functions defined for traces in Chapter 2.

I am now ready to present the semantics. Given a trace and a complete STM, I define a *semantic trace* as a record with these components:

- events: Event list

  This is the sequence of events in the trace.

- defs: APath set

  This is the set of all access paths of arguments that define their values, according to the STM.

- uses: APath set

  This is the set of all access paths of arguments that use their values, according to the STM.

Give a trace T and a STM M, constructing the corresponding semantic trace is easy:

`Routine` `MakeSemanticTrace(`T`,` M`)`

`Return` $\{\text{events} : T, \text{defs} : \text{Defs}(M) \text{uses} : \text{Uses}(M)\}$

A *semantic-trace state* is a record with these components:

- pc: Nat

  This is the offset in the event sequence of the next event to be executed.

- vstates: Value $\mapsto$ VState

  This map implements state variables. It maps each value in the semantic trace to its current state.

- stack: VState list

  This is the activation stack. For each active call or callback, a VState on this stack represents the state of the activation. In the interest of simplicity, the semantics guarantee that the stack is never empty.

The initial state is $\{\text{pc} : 0, \text{vstates} : \text{ZeroMap}, \text{stack} : [0]\}$.

Figure 27 lists the transition rules for a semantic trace $\{\text{events} : \textit{events}, \text{defs} : \textit{defs}, \text{uses} : \textit{uses}\}$. Semantic traces are straight-line code, so the rules are very simple. The rule for a call or callback says that the event pushes an activation record onto the stack and, in parallel, assigns new states to all state variables in the event's `def` effects. Both the activation record and the new states depend on the stack and on the state variables in the event's `use` effects. The rule for a return says that the event pops an activation record off of the stack and, again in parallel,

Call and Callback

$$\frac{\begin{array}{c} events[pc] = e \\ \text{Call}(e) \vee \text{Callback}(e) \\ U = \{a \in \text{Args}(e) | \text{Path}(a) \in uses\} \\ D = \{a \in \text{Args}(e) | \text{Path}(a) \in defs\} \end{array}}{\begin{array}{l} \{\text{pc} : pc, \text{vstates} : V, \text{stack} : s :: S\} \\ \Rightarrow \{\text{pc} : pc + 1, \\ \quad \text{vstates} : V[\{\text{Value}(d) \\ \quad\quad \mapsto ?(\text{Path}(d), s, \\ \quad\quad\quad \{(\text{Path}(u), \text{Value}(u)) | u \in U\}) \\ \quad\quad | d \in D\}], \\ \quad \text{stack} : ?(\text{Name}(e), s, \\ \quad\quad \{(\text{Path}(u), \text{Value}(u)) | u \in U\}) :: s :: S\} \end{array}}$$

Return

$$\frac{\begin{array}{c} events[pc] = e \\ \text{Return}(e) \\ U = \{a \in \text{Args}(e) | \text{Path}(a) \in uses\} \\ D = \{a \in \text{Args}(e) | \text{Path}(a) \in defs\} \end{array}}{\begin{array}{l} \{\text{pc} : pc, \text{vstates} : V, \text{stack} : s :: S\} \\ \Rightarrow \{\text{pc} : pc + 1, \\ \quad \text{vstates} : V[\{\text{Value}(d) \\ \quad\quad \mapsto ?(\text{Path}(d), s, \\ \quad\quad\quad \{(\text{Path}(u), \text{Value}(v)) | u \in U\}) \\ \quad\quad | d \in D\}], \\ \quad \text{stack} : S\} \end{array}}$$

Figure 27: Transition rules for a semantic trace $\{\text{events} : events, \text{defs} : defs, \text{uses} : uses\}$.

assigns new states to all state variables in the event's `def` effects. As with call and callback events, the new states depend on the stack and on the state variables in the event's `use` effects.

3.3 Transformations of semantic traces

This section presents three transformations of semantic traces, which improve efficiency and increase the expressiveness of specifications. The *flags* transformation renames events to include flags that change their function. *Argument dropping* eliminates arguments that have no effect on the semantics defined in Section 3.2. Similarly, *event-pair dropping* drops pairs of events that have no semantic effect.

3.3.1 Flags

This section presents the flags transformation, which improves the expressiveness of specifications by renaming events to include flags that change their function.

Strauss associates a name with every event. For call events, the name identifies the routine that is called; for callback events, the name gives the type of the callback; and for return events, the name is the name of the corresponding call or callback event. It is natural to expect that events with the same name should have similar consequences in the library. However, classifying events by name can be too coarse. For example, the C standard I/O routine `fopen` takes two arguments: a file name and a mode. Calls to `fopen` have different effects for different values of the `mode` argument:

- If the mode is "r", the file is opened for reading.

- If the mode is "w", the file is truncated or created, and then opened for writing.

Some desirable specifications do not depend on the mode: `fclose` must always be called after `fopen`, for example. However, some do: for example, `fread` cannot be called on a file that is not open for reading. To mine the latter kind of specification, Strauss must classify `fopen` calls by their mode.

The *flags transformation* tells Strauss to classify events by the values of certain arguments, called *flags*. Basically, the transformation appends the value of each flag in an event to the name of that event. The type of the flags transformation is

$$\text{APath set} \times \text{Trace} \rightarrow \text{Trace}$$

The first argument is a set of access paths; call this set *flags*. An argument is a flag iff its access path is in *flags*. The second argument is the abstract trace to transform, and the result is the transformed trace.

Figure 28 gives rewrite rules for the flags transformation. The rules use a function, AddFlags, that takes a name and a set of arguments and returns a new name that incorporates the arguments. AddFlags uses a format that puts the arguments in square brackets after the name; for example,

AddFlags(`fopen`, { (`call.fopen.mode`, `"w"`) })

is

```
fopen[call.fopen.mode = "w"].
```

Call

$$\frac{F = \{a \in A | \text{Path}(a) \in \mathit{flags}\}}{\text{Call } (\mathit{name}, A) \Rightarrow \text{Call } (\text{AddFlags}(\mathit{name}, F), A)}$$

Callback

$$\frac{F = \{a \in A | \text{Path}(a) \in \mathit{flags}\}}{\text{Callback } (\mathit{name}, A) \Rightarrow \text{Callback } (\text{AddFlags}(\mathit{name}, F), A)}$$

Return

$$\frac{F = \{a \in A | \text{Path}(a) \in \mathit{flags}\}}{\text{Return } (\mathit{name}, A) \Rightarrow \text{Return } (\text{AddFlags}(\mathit{name}, F), A)}$$

Figure 28: Rewrite rules for the flags transformation.

Call

$$\frac{K = \{a \in A | \text{Path}(a) \in \mathit{uses} \cup \mathit{defs}\}}{\text{Call } (\mathit{name}, A) \Rightarrow \text{Call } (\mathit{name}, K)}$$

Callback

$$\frac{K = \{a \in A | \text{Path}(a) \in \mathit{uses} \cup \mathit{defs}\}}{\text{Callback } (\mathit{name}, A) \Rightarrow \text{Callback } (\mathit{name}, K)}$$

Return

$$\frac{K = \{a \in A | \text{Path}(a) \in \mathit{uses} \cup \mathit{defs}\}}{\text{Return } (\mathit{name}, A) \Rightarrow \text{Return } (\mathit{name}, K)}$$

Figure 29: Rewrite rules for dropping arguments from a semantic trace $\{\text{events} : \mathit{events}, \text{defs} : \mathit{defs}, \text{uses} : \mathit{uses}\}$.

3.3.2 Dropping arguments

Arguments whose access paths are neither in *uses* nor in *defs* play no role in the semantics of Figure 27. Such arguments can be dropped from a semantic trace without changing its meaning. Strauss always performs this transformation before scenario extraction, so scenario extraction does not need to cope with needlessly complicated traces.

Figure 29 gives rewrite rules for dropping arguments from a semantic trace $\{\text{events} : \mathit{events}, \text{defs} : \mathit{defs}, \text{uses} : \mathit{uses}\}$.

Name	Retained	
	Number	**Percent**
PrsAccelTbl	3703	.002
PrsTransTbl	3340	.002
Quarks	22165	.01
RegionsAlloc	33072	.02
RegionsBig	164739	.08
RmvTimeOut	18701	.01
XFreeGC	2817	.001
XGContextFromGC	3079	.002
XGetSelOwner	635	.0003
XInternAtom	243358	.1
XPutImage	13275	.007
XSaveContext	596016	.3
XSetFont	33008	.02
XSetSelOwner	160318	.08
XtFree	275407	.1
XtOwnSel	160548	.08
XtRealizeProc	9616	.005
Average	102576	.05

Table 1: Benefit of dropping arguments. For each specification, **Retained Number** is the number of arguments *not* dropped, and **Retained Percent** is the percentage of arguments not dropped. The last row gives the mean over all specifications (arithmetic mean for **Retained Number** and geometric mean for **Retained Percent**).

Table 1 shows the benefit of dropping arguments. The table lists the number of arguments that were *not* dropped, while mining 17 specifications about the X11 interface. Each specification was mined from the same set of 90 traces, gathered from full runs of 72 programs (see Chapter 6 for more information about these traces). Altogether, the traces contained slightly under 200 million arguments.

On average, less than 0.05 percent of all arguments were retained. The XSaveContext specification retained the most arguments, still barely 0.3 percent of all arguments.

3.3.3 Dropping event pairs

Traces can be very large; for example, Chapter 6 discusses a trace with almost three and a half million events! Processing very large traces takes a long time, even with linear algorithms. Fortunately, given a state-transition model, most events play no useful part in the semantic trace and can safely be ignored.

Consider a call (or callback) that is immediately followed by its return. If neither the call nor the return has an argument in *defs* or in *uses*, then the pair of events can have no effect on the final state of the semantic trace and, moreover, the pair cannot be affected by any other events in the semantic trace. In fact, if the state immediately before the call is $[\text{pc} : pc, \text{vstates} : V, \text{stack} : S]$, then the state immediately after the return must be $[\text{pc} : pc + 2, \text{vstates} : V, \text{stack} : S]$. According to Figure 27, neither the call nor the return changes V and, although the call pushes a value state onto S, the return immediately pops it off. I call such a pair *droppable*.

Droppable pairs can be safely removed from the trace, and removing them is desirable because it gives a smaller trace. Figure 30 lists the rewrite rules for dropping pairs. Note that dropping one pair can expose another droppable pair:

0 Call (foo, {})

1 Callback (foo_callback, {})

2 Return (foo_callback, {})

3 Return (foo, {})

Here, dropping the pair on lines 1 and 2 exposes a droppable pair on lines 0 and 3. As it reads a trace, Strauss conserves memory and speeds scenario extraction

Call

$$\frac{\begin{array}{c} e = \text{Call } (name, A) \\ e' = \text{Return } (name, A') \\ A \cap (defs \cup uses) = \epsilon \\ A' \cap (defs \cup uses) = \epsilon \end{array}}{\begin{array}{l} \{\text{events} : es_0 \cdot [e, e'] \cdot es_1, \text{defs} : defs, \text{uses} : uses\} \\ \Rightarrow \{\text{events} : es_0 \cdot es_1, \text{defs} : defs, \text{uses} : uses\} \end{array}}$$

Callback

$$\frac{\begin{array}{c} e = \text{Callback } (name, A) \\ e' = \text{Return } (name, A') \\ A \cap (defs \cup uses) = \epsilon \\ A' \cap (defs \cup uses) = \epsilon \end{array}}{\begin{array}{l} \{\text{events} : es_0 \cdot [e, e'] \cdot es_1, \text{defs} : defs, \text{uses} : uses] \\ \Rightarrow \{\text{events} : es_0 \cdot es_1, \text{defs} : defs, \text{uses} : uses\} \end{array}}$$

Figure 30: Rewrite rules for dropping event pairs from a semantic trace $\{\text{events} : events, \text{defs} : defs, \text{uses} : uses\}$.

Name	Retained	
	Number	**Percent**
PrsAccelTbl	7406	0.04
PrsTransTbl	6680	0.03
Quarks	34818	0.2
RegionsAlloc	66144	0.3
RegionsBig	238682	1.2
RmvTimeOut	37402	0.2
XFreeGC	5634	0.03
XGContextFromGC	6158	0.03
XGetSelOwner	1270	0.01
XInternAtom	486672	2.4
XPutImage	10454	0.05
XSaveContext	596016	3.0
XSetFont	66016	0.3
XSetSelOwner	320555	1.6
XtFree	550814	2.8
XtOwnSel	321015	1.6
XtRealizeProc	9616	0.05
Average	162668	0.8

Table 2: Benefit of dropping event pairs. For each specification, **Retained Number** is the number of events *not* dropped, and **Retained Percent** is the percentage of events not dropped. The last row gives the mean over all specifications (arithmetic mean for **Retained Number** and geometric mean for **Retained Percent**).

by dropping all droppable pairs.

Table 2 shows the benefit of dropping event pairs. The table lists the number of events that were *not* dropped, while mining 17 specifications about the X11 interface. Each specification was mined from the same set of 90 traces, gathered from full runs of 72 programs (see Chapter 6 for more information about these traces). Altogether, the traces contained slightly under 20 million events.

On average, less than one percent of all events were retained. The XSaveContext specification retained the most events, still barely 3 percent of all events.

Chapter 4

Extracting Scenarios

Strass mines specifications from traces, which record events (calls, callbacks, and returns) that occur at the interface between a library and its client programs. Within practical limits (see Chapter 6), each trace records every interface event in a program execution, along with every value referenced by each event. However, finite-state specifications cannot and should not consider all events and values simultaneously. Thus, a specification miner must decide which events and values to consider together, which to consider separately, and which to ignore completely.

The state-transition models and semantic traces discussed in Chapter 3 provide one piece of the puzzle. The state-transition model determines which aspect of program behavior is under consideration; events that have nothing to do with that aspect become no-ops in the semantic trace and are removed by the transformation described in Section 3.3.3.

However, problems remain. Suppose, for example, that the interesting aspect is the locking and unlocking of files. In a given run, a program might lock and unlock a number of different files. However, because whether file A is locked or not is independent of whether file B is locked or not, it seems strange to consider lock and unlock events for A together with lock and unlock events for B. Also, a miner cannot learn a specification that simultaneously tracks every operation on

every file (while keeping files distinct), because such a specification would not be finite-state.

This example points out two problems. The first problem is identifying events that are independent, so that they can be considered separately. The second problem is keeping the number of values (files in the example) bounded, so that a finite-state specification can track them simultaneously while maintaining the distinction between different values.

My solution to both problems is to extract *scenarios* from the trace. Each scenario consists of a few dependent events that reference a few values, so scenarios are appropriate inputs for a learner of finite-state specifications (see Chapter 5).

Intuitively, a scenario is a set of events in a semantic trace, which are related by the flow of state through state variables. In the following, I explain first how scenarios identify independent events, and then how to bound the size of a scenario, so that the number of values in the scenario is also bounded.

Figure 31 shows an example scenario, within its trace. The trace is of a program that manipulates files, and the trace is interpreted as a semantic trace according to an STM that tracks file locks. The scenario includes all events that manipulate the locking state of file `0x10`, which the semantic trace represents with the state variable State[0x10]. When Strauss builds scenarios, it starts with a seed event and searches for events that are related to the seed by the flow of states. In Figure 31, the seed is the `unlock` call on line 17; the scenario also includes

- The `lock` call on line 5, because the call on line 17 reads the state that the call on line 5 writes to State[0x10]. That is, there is a *flow dependence* from the `f` argument of the `lock` call on line 5 to the `f` argument of the `unlock`

Trace:

```
 1  call new-file()
 2  return new-file(?  = 0x10)
 3  call new-file()
 4  return new-file(?  = 0x20)
 5  call lock(f = 0x10)
 6  return lock()
 7  call write(f = 0x10)
 8  return write()
 9  call write(f = 0x20)
10  return write()
11  call lock(f = 0x20)
12  return lock()
13  call read(f = 0x20)
14  return read()
15  call unlock(f = 0x20)
16  return unlock()
17  call unlock(f = 0x10) [seed]
18  return unlock()
```

State-transition model:

```
return new-file (def ?)
call lock (def use f)
call unlock (def use f)
```

Figure 31: A scenario (*in this typeface*) within its trace. The trace is of a program that manipulates files, and the trace is interpreted as a semantic trace according to an STM that tracks file locks. The scenario includes all events that manipulate the locking state of file `0x10`, and is seeded by the `unlock` call on line 17.

call on line 17, which is *carried* by `State[0x10]`.

- The `new-file` return on line 2, because there is a flow dependence from the `?` argument of the `new-file` return on line 2 to the `f` argument of the `unlock` call on line 5. This dependence and the dependence from line 5 to line 17 form a chain of flow dependences from the return on line 2 to the seed on line 17.

- The `new-file` call on line 1, because the return on line 2 reads the state that the call on line 1 writes to the stack. That is, there is a flow dependence from the `new-file` call on line 1 to the `new-file` return on line 2, which is carried by the stack. This dependence, together with the previous flow dependences, forms a chain of flow dependences from the call on line 1 to the seed on line 17.

- The `unlock` return on line 18, because there is a stack-carried flow dependence from the `unlock` call on line 17 to the `unlock` return on line 18.

Strauss extracts a scenario seeded by each trace event that matches a *seed pattern*, such as `call unlock(f = F)` (see Section 2.4.1). Because scenarios only include events that are related to a seed by state flow, independent events never appear in the same scenario. However, the specification learner can still compare scenarios with one another, because all scenarios focus on the same aspect of an interface. For example, the scenario seeded by the `unlock` call on line 15 on Figure 31 includes all events that manipulate the locking state of file `0x20`; this scenario is very similar to the scenario seeded by the `unlock` call on line 17, even though the two scenarios contain different events.

In the above example, each scenario contained a bounded number of values: in fact, each scenario contained a single value. In general, however, this might not be true if the size of the scenario is unbounded.

To see why, consider adding the following routine to the file interface:

```
assert-and-lock(in file f0, in file f1)
```

When a caller passes a file `f0` and a file `f1` to `assert-and-lock`, `assert-and-lock` verifies that the program holds the lock on file `f0` and then locks file `f1`. It is an error to call `assert-and-lock` if the lock on `f0` is not held or if the lock on `f1` is held.

Figure 32 shows a scenario for this extended file interface, within its trace. The scenario includes most of the trace events, because flow dependences connect most of the trace events to the seed on line 15. For example, the `new-file` call on line 3 connects to the seed by a stack-carried flow dependence from line 3 to line 4, a flow dependence carried by `State[0x20]` from line 4 to line 13, and a flow dependence carried by `0x30` from line 13 to the seed.

The scenario in Figure 32 contains three values, but it is easy to see how to write similar examples, in which a scenario contains four values, five values, or any finite number of values. These examples demonstrate that, even though scenarios contain only related events, they can still contain an arbitrary number of values.

I solve this problem in Strauss by requiring bounds on the size of scenarios. A *scenario criterion* (either inferred automatically or supplied by the specification developer) controls which scenarios will be extracted. A scenario criterion C is a tuple with three elements: the seed pattern mentioned above, plus two elements,

Trace:

```
 1  call new-file()
 2  return new-file(?  = 0x10)
 3  call new-file()
 4  return new-file(?  = 0x20)
 5  call new-file()
 6  return new-file(?  = 0x30)
 7  call lock(f = 0x10)
 8  return lock()
 9  call assert-and-lock(f = 0x10, f = 0x30)
10  return assert-and-lock()
11  call unlock(f = 0x30)
12  return unlock()
13  call assert-and-lock(f = 0x20, f = 0x30)
14  return assert-and-lock()
15  call unlock(f = 0x30) [seed]
16  return unlock()
```

State-transition model:

```
return new-file (def ?)
call lock (def use f)
call unlock (def use f)
call assert-and-lock(use f0, def use f1)
```

Figure 32: A scenario (*in this typeface*), within its trace, for the extended file interface. The trace is of a program that creates, locks, and unlocks files, and is interpreted according to an STM that traces file locks. The scenario is seeded by the unlock call on line 15, and includes all events that affect the locking state of file `0x30` there, directly or indirectly.

the backwards-radius r_b and the forwards-radius r_f, which bound the size of scenarios. The scenario will contain at most r_b events before the seed event and at most r_f events after the seed event.

Extracting short scenarios bounds the number of values in a scenario, but it also has other benefits. First, the time to extract each scenario is also bounded by a constant; since the number of extracted scenarios is at most linear in the length of the trace, Strauss's time and space requirements remain linear in the length of the trace.

Also, because the running time of Strauss's NFA learner increases as the third power of the the length of its input (a common, although not universal, time complexity for NFA learners), short scenarios allow Strauss to learn specifications quickly.

Overview of this chapter

The rest of this chapter explains how Strauss infers flow dependences (Section 4.1) and uses them to extract scenarios (Section 4.2).

4.1 Inferring flow dependences

This section explains how Strauss infers flow dependences, which connect events or arguments that produce states with events or arguments that consume those states. I end with a short argument that proves that flow dependences are adequate for semantic trace analyses like scenario extraction.

According to the semantic rules in Figure 27 (Chapter 3), states flow in two ways: through the stack and through arguments. So, given a semantic trace $\{\text{events} : \mathit{events}, \text{uses} : \mathit{uses}, \text{defs} : \mathit{defs}\}$, I define *stack-flow dependences* and *argument-flow dependences*:

Stack-flow dependences A stack-flow dependence connects a call or callback event with an event that executes in its immediate context. Referring to the semantics in Figure 27, a stack-flow dependence captures two kinds of flow: the flow that occurs when a call or callback pushes a state onto the stack (see the "Call and Callback" rule), and a later event uses the state as a parameter of ⑦; and the flow that occurs when a return event pops a state off of the stack.

Formally, I define a stack-flow-dependence relation D_{stack} as follows:

$$\frac{\begin{array}{c} \mathit{events} = \ldots \cdot [e_c] \cdot \mathit{es} \cdot e \cdot \ldots \\ \text{Call}(e_c) \vee \text{Callback}(e_c) \\ \text{ReturnOf}(e_c) \notin \mathit{es} \\ \forall e \in \mathit{es}.\text{Call(e)} \vee \text{Callback}(e_c) \Rightarrow \text{ReturnOf}(e) \in \mathit{es} \end{array}}{(e_c, e) \in D_{stack}}$$

This definition uses a new function:

> ReturnOf: Event list × Event → Event
>
> Given a sequence of events and a call or callback event in the sequence, returns the corresponding return event in the sequence.

Intutively, the definition says that there is a stack-flow dependence from e_c to e iff e_c is a call or callback, the return from e_c does not occur before e, and all calls and callbacks between e_c and e return before e. Note that, even if e uses the state that e_c pushed onto the stack in more than one instance of ⓘ, there is only one stack-flow dependence from e_c to e. Also, if e is the return from e_c, there is a stack-flow dependence from e_c to e, even if e does not use the state that e_c pushed onto the stack in an instance of ⓘ.

Strauss computes D_{stack} incrementally (in time and space linear in the length of the trace), by maintaining a stack of call and callback events while sequentially traversing the trace. Suppose that e_c is on top of the stack when Strauss begins processing e. Then, Strauss adds $[e_c, e]$ to D_{stack}. Next, if e is a call or callback event, Strauss pushes e onto the stack. Otherwise, e must be the return event for e_c, so Strauss pops e_c off of the stack.

Argument-flow dependences An argument-flow dependence connects an argument that updates the state of a state variable with an argument that reads the updated state. Referring to the semantics in Figure 27, an argument-flow dependence occurs when an event updates the vstates map with a new state, and a later event uses the state as a parameter of ⓘ.

Formally, I define an argument-flow dependence relation D_{arg} between arguments as follows:

$$events = \ldots \cdot [e_d] \cdot es \cdot [e_u] \cdot \ldots$$

$$a_d \in \text{Args}(e_d) \wedge \text{Path}(a_d) \in defs$$

$$a_u \in \text{Args}(e_u) \wedge \text{Path}(a_u) \in uses$$

$$\text{Value}(a_d) = \text{Value}(a_u)$$

$$\frac{\forall e \in es. \forall a \in \text{Args}(e). \text{Path}(a) \notin defs \vee \text{Value}(a) \neq \text{Value}(a_d)}{(a_d, a_u) \in D_{arg}}$$

This definition says that there is an argument-flow dependence from a_d to a_u iff a_d defines a state variable that a_u reads, and the definition at a_d reaches a_u: that is, a_u follows a_d and there are no definitions of the state variable between a_d and a_u.

My reason for defining D_{arg} as a relation on arguments instead of on events will be clear in Section 5.2.3, which explains how to use type inference to improve specification learning.

Strauss computes D_{arg} incrementally (in time and space linear in the length of the trace), by solving a reaching-definitions problem while sequentially traversing the trace. The algorithm maintains a map from each value to the last argument that defined the value. Strauss processes each event e in two steps. The first step adds flow dependences of the form (a_d, a_u) to D_{arg}, where $a_u \in \text{Args}(e)$ uses a value that the map says was last defined by a_d. The second step updates the map: for each $a_d \in \text{Args}(e)$ that defines a value v, Strauss maps v to a_d.

I will have occasion in Chapter 5 to distinguish between stack-flow and argument-flow dependences. However, the distinction is unnecessary for scenario extraction, so I define a general flow-dependence relation D_{flow}, such that $(e_d, e_u) \in D_{flow}$ iff

$$(e_d, e_u) \in D_{stack} \vee \exists (a_d, a_u) \in D_{arg}.(a_d \in \text{Args}(e_d) \wedge a_u \in \text{Args}(e_u))$$

The following function will be useful elsewhere in the dissertation:

> InferFlowDependences: SemanticTrace $\rightarrow$ Event $\times$ Event
>
> Given a semantic trace P, InferFlowDependences(P) is the general flow-dependence relation (D_{flow}) for P.

Flow dependences impose a directed acyclic graph (DAG) structure on events. I will freely use the usual DAG terminology (chain, ancestors, descendants, and so forth).

Adequacy of flow dependences

A *program dependence graph (PDG)* is a graph whose nodes are program statements and whose edges represent dependences between statements. For a rich imperative language with loops and conditionals, Horwitz, Prins and Reps showed that PDGs with control dependences, def-order dependences, and flow dependences characterize program semantics [31], in the sense that if two programs have isomorphic PDGs, then they are semantically equivalent. Also, according to their definitions, control dependences and def-order dependences exist only in programs with loops or conditionals.

Their result implies that flow dependences characterize the semantics of straight-line imperative programs, such as semantic traces. Thus, in principle, flow dependences are adequate for semantic analyses of semantic traces like scenario extraction.

4.2 Extracting scenarios

Using flow dependences, Strauss extracts scenarios (small sets of related events) from traces. This section defines scenarios formally and explains how Strauss extracts bounded scenarios, which consist of a few related events that reference a few values.

4.2.1 Scenarios

Given a semantic trace $\{\text{events} : \mathit{events}, \text{uses} : \mathit{uses}, \text{defs} : \mathit{defs}\}$ with flow dependences D_{flow}, a scenario S is a pair $(e_{\mathit{seed}}, \mathit{es})$, where

- es is a subsequence of events.
- $e_{\mathit{seed}} \in \mathit{es}$.
- If $e_a \in \mathit{es}$, $e_b \in \mathit{es}$ and $[e_a = e_0, \ldots, e_b = e_n]$ is a D_{flow} chain in events, then $e_i \in \mathit{es}$ for any $0 \leq i \leq n$.

  Intuitively, this condition means that, if state flows from e_a to e_b through an event e_c, then e_c is also in the scenario.

- If $e \in \mathit{es}$, then there is a D_{flow} chain through e in es whose endpoints are also on a D_{flow} chain through e_{seed}. That is, there is a chain $C_0 = [e_a =$

$e_{0,0}, \ldots, e_b = e_{0,n}]$ in es and a chain $C_1 = [e_a = e_{1,0}, \ldots, e_b = e_{1,m}]$ such that $e = t_{0,i}$ for some $0 \leq i \leq n$ and $e_{seed} = t_{1,j}$ for some $0 \leq j \leq m$.

Intuitively, this condition means that the scenario contains no events that are unrelated to the seed by state flow.

Section 2.4 defined the semantics of Strauss specifications in terms of a Scenarios function. I can now define that function:

Scenarios: Trace $\times$ STM $\times$ Event $\rightarrow$ Scenario set

Given a trace T, a STM M, and a seed event e, Scenarios(T, M, e) is the set of all scenarios (e, es) in the semantic trace $\{\text{events} : T, \text{defs} : \text{Defs}(M), \text{uses} : \text{Uses}(M)\}$.

4.2.2 Extracting bounded scenarios

For reasons discussed in this chapter's introduction, Strauss extracts bounded scenarios, which satisfy a given scenario criterion. In fact, Strauss extracts exactly one well-defined scenario for each event that matches the criterion's seed pattern. This section defines that scenario and explains how Strauss extracts it.

I assume throughout that I am given a semantic trace $\{\text{events} : \mathit{events}, \text{uses} : \mathit{uses}, \text{defs} : \mathit{defs}\}$ with flow dependences D_{flow}. All scenarios are from this semantic trace.

What Strauss extracts

A scenario criterion C is a tuple with three elements: a seed pattern pat, a non-negative backwards-radius r_b and a non-negative forwards-radius r_f. I say that a

scenario (e_{seed}, es) *satisfies a scenario criterion* $C = (pat, r_b, r_f)$ iff

- e_{seed} matches the seed pattern *pat*. See Section 2.4.1 for the definition of seed patterns and the events that match them; intuitively, a seed pattern is a prototype for events (which includes argument names but omits argument values) and an event matches the seed pattern if it has the same form as the prototype.

- Let $es_{before} \subset es$ be the set of scenario events that strictly precede e_{seed} in *events*. Then, $|es_{before}| \leq r_b$.

- Let $es_{after} \subset es$ be the set of scenario events that strictly follow e_{seed} in *events*. Then, $|es_{after}| \leq r_f$.

Given a seed event and scenario criterion, there may be many scenarios that satisfy the criterion. However, Strauss extracts a particular scenario—the largest *convex scenario*. I say that a scenario $S = (e_{seed}, es)$ is convex iff all of the following is true:

- If e is a D_{flow} ancestor of e_{seed}, then $e \in es$ or there is no $e' \in es$ such that e' precedes e in *events*.

- If e is a D_{flow} descendant of e_{seed}, then $e \in es$ or there is no $e' \in es$ such that e' follows e in *events*.

Figure 33 illustrates convexity for scenarios. Assuming that all events are either ancestors or descendants of `event1`, the scenario on the left is convex. The scenario on the right is not convex, because it is missing `event2` and `event3`, both of which precede `event4`.

```
event0           | event0
event1 [seed]    | event1 [seed]
event2           |
event3           |
                 | event4
```

Figure 33: Scenarios illustrating convexity. Assuming that all events are either ancestors or descendants of `event1`, the scenario on the left is convex and the scenario on the right is not.

In general, Strauss extracts the largest convex scenario that still satisfies the criterion. If there are two such scenarios, Strauss extracts the scenario with more descendants.

This decision is somewhat arbitrary, although it does have some nice properties. The "largest" requirement means that increasing r_b and r_f will never cause Strauss to extract a smaller scenario. And, the convexity requirement means that Strauss tends to extract scenarios with events that occur near each other. I conjecture that it is easier for a person to understand such scenarios.

Alternatives (which I have not investigated) include extracting some other well-defined scenario that satisfies the criterion, extracting a random scenario, or even extracting multiple scenarios for each seed. Or, the scenario criterion itself could be changed, for example to specify a bound on the number of scenario values or to specify "stop" patterns, which would match events where Strauss should stop growing a scenario.

How Strauss extracts scenarios

Figures 34 and 35 list Strauss's scenario-extraction algorithm. The algorithm extracts the unique scenario around e_{seed}, which satisfies the scenario criterion

```
Routine ExtractOne(e_seed, r_b, r_f, events, D_flow)
  (es_a, es_d) := (Ancestors(e_seed, r_b, events, D_flow),
                   Descendants(e_seed, r_f, events, D_flow))
  es_ad := es_a ∪ es_d
  (e_a, e_d) := (earliest event in es_a, latest event in es_d)
  es_ar := DescendantsInRange(es_a, e_a, e_d, events, D_flow)
  es_dr := AncestorsInRange(es_d, e_a, e_d, events, D_flow)
  es := es_ad ∪ (es_ar ∩ es_dr)
  Return (e_seed, es as an ordered list)

Routine Ancestors(e_seed, r_b, events, D_flow)
  (es_a, W) := (∅, immediate ancestors of e_seed)
  Mark events in W as visited
  While W ≠ ∅ and |es_a| < r_b Do
    e := event in W nearest e_seed in events
    es_ea := unvisited immediate ancestors of e
    Mark events in es_ea as visited
    (es_a, W) := (es_a ∪ {e}, (W ∪ es_ea) − {e})
  Return es_a

Routine Descendants(e_seed, r_f, events, D_flow)
  Return {r_f closest descendants of e_seed}

Routine DescendantsInRange(es_a, e_a, e_d, events, D_flow)
  Return {e ∈ [e_a, ..., e_d] | ∃e' ∈ es_a. e is on a D_flow chain from e'}

Routine AncestorsInRange(es_d, e_a, e_d, events, D_flow)
  Return {e ∈ [e_a, ..., e_d] | ∃e' ∈ es_d. e' is on a D_flow chain from e}
```

Figure 34: An algorithm for extracting the largest convex scenario around e_{seed} with at most r_b ancestors and r_f descendants.

```
Routine ExtractScenario(C = (e_seed, r_b, r_f), events, D_flow)
  S := null
  N := 0
  While N ≤ r_b + r_f
    N_b := Max(N − r_f, 0)
    While N_b ≤ N ∧ N_b ≤ r_b
      N_f := N − N_b
      S' := ExtractOne(e_seed, r_b − N_b, r_f − N_f, events, D_flow)
      If S' satisfies C
        If S = null ∨ (|S'| > |S|)
          S := S'
      N_b := N_b + 1
    If S ≠ null
      Return S
    N := N + 1
  // This line is unreachable.
```

Figure 35: The outer loop of Strauss's algorithm for extracting scenarios.

$C = (pat, r_b, r_f)$ and the conditions described above. The algorithm assumes that e_{seed} matches pat, of course.

I have divided the algorithm into an inner loop (Figure 34) and an outer loop (Figure 35).

In the inner loop, `ExtractOne` extracts a convex scenario with at most r_b ancestors of the seed and r_f descendants of theseed.

Broadly speaking, `ExtractOne` has two steps. The first step finds r_b ancestors and r_f descendants of e_{seed} and places them in the set es_{ad}. The events in es_{ad} might not comprise an entire scenario because es_{ad} may be missing some events on D_{flow} chains from ancestors to descendants. The second step finds these missing events and adds them to es_{ad} to produce a scenario, S. Note that S may or may not satisfy C.

The outer loop (`ExtractScenario`) calls `ExtractOne` repeatedly, decreasing r_b and/or r_f until the largest scenario that satisfies C is found. The logic of `ExtractScenario` is a bit tricky. The variable N holds a total, which should be subtracted from r_b and r_f. Initially, N is 0, which means nothing is subtracted from either bound. For larger values of N, `ExtractScenario` tries all ways of partitioning N into an amount to subtract from r_b (N_b) and an amount to subtract from r_f (N_f), calls `ExtractOne` to extract a scenario for each partition, and returns the largest scenario that satisfies C (if one exists).

N is increased until a scenario that satisfies C is found. This must happen, because when $N = r_b + r_f$, `ExtractOne` is forced to extract the scenario $(e_{seed}, [e_{seed}])$, which certainly satisfies C.

Now let's look more closely at both steps of `ExtractOne`. The first step constructs three sets:

$$\begin{aligned} es_a &= \{r_b \text{ closest ancestors of } e_{seed}\} \\ es_d &= \{r_f \text{ closest descendants of } e_{seed}\} \\ es_{ad} &= \{es_{seed}\} \cup es_a \cup es_d \end{aligned}$$

Strauss uses a simple prioritized worklist algorithm to construct es_a and es_d. I describe the algorithm for ancestors only because the algorithm for descendants differs only in the direction in which the algorithm follows dependences. Initially, es_a is empty, the worklist holds the immediate ancestors of e_{seed}, and the events in the worklist are marked as "visited". Then, the algorithm repeats the following three steps until the worklist is empty or the algorithm finds r_b ancestors. Let e be the worklist event that is nearest e_{seed} in *events*. The first step removes e from

the worklist and adds e to es_a. The second step finds all immediate ancestors of e that are not already marked as "visited". Finally, the third step marks these immediate ancestors as "visited" and adds them to the worklist.

es_{ad} (the union of es_a and es_d) is not necessarily a scenario, because it might be missing some events along D_{flow} chains from events in es_a to events in es_d. In the following, e_a denotes the earliest ancestor in es_a and e_d is the latest descendant in es_d. Any missing events must lie between e_a and e_d in the *events* sequence, and must be on a D_{flow} chain from some member of es_a to some member of es_d. To find such events it suffices to follow dependences forwards from each event in es_a and backwards from each event in es_d, stopping the forwards search at e_d and the backwards search at e_a. The missing events will be found by both searches. Precisely, Strauss constructs two sets:

$$\begin{aligned} es_{ar} &= \{e \in [e_a, e_d] \mid \exists e' \in S_a.\ e \text{ is on a } D_{flow} \text{ chain from } e'\} \\ es_{dr} &= \{e \in [e_a, e_d] \mid \exists e' \in S_d.\ e' \text{ is on a } D_{flow} \text{ chain from } e\} \end{aligned}$$

These sets are easy to construct with a worklist algorithm, similar to the one used to construct es_a; I leave the details of that algorithm to the reader.

The "final" scenario is $es = es_{ad} \cup (es_{ar} \cap es_{dr})$. I put "final" in quotes because es might contain more than backwards-radius events before the seed or more than forwards-radius events after the seed.

Name	Retained		Depend.	Scen.	Time (s)	
	Events	Args			Infer	Extract
PrsAccelTbl	7406	3703	3742	3640	9.7	0.2
PrsTransTbl	6680	3340	5750	2410	1.2	0.5
Quarks	34818	22165	30535	17409	3.5	2.3
RegionsAlloc	66144	33072	49581	33072	2.2	2.6
RegionsBig	238683	164740	277588	110574	29.4	17.0
RmvTimeOut	37402	18701	19504	803	1.5	0.1
XFreeGC	5634	2817	4119	2817	0.6	0.2
XGContextFromGC	6158	3079	4640	3079	0.6	0.3
XGetSelOwner	1270	635	782	146	0.2	0.0
XInternAtom	486672	243358	318257	15678	109.8	4.2
XPutImage	10454	13275	10653	2821	1.3	0.7
XSaveContext	596016	596016	857358	298008	40.8	74.5
XSetFont	66016	33008	64495	33008	2.8	5.7
XSetSelOwner	320555	160318	190508	146	101.9	0.0
XtFree	550814	275407	383038	275407	30.1	20.2
XtOwnSel	321015	160548	191245	376	101.5	0.1
XtRealizeProc	9616	9616	12753	2435	10.3	0.2

Table 3: Cost of inferring flow dependences and extracting scenarios. For each specification, **Retained Events** and **Retained Args** are the number of events and arguments (respectively) that were not dropped by the transformations of Section 3.3, **Depend.** is the number of flow dependences inferred, **Scen.** is the number of scenarios extracted, **Time Infer** is the number of seconds spent inferring dependences, and **Time Extract** is the number of seconds spent extracting scenarios.

4.3 Experimental results

Table 3 shows the cost of flow-dependence inference and scenario extraction, while mining 17 specifications about the X11 interface. Each specification was mined from the same set of 90 traces, gathered from full runs of 72 programs (see Chapter 6 for more information about these traces). For each specification, I report the best times from three runs on an Ultra Enterprise 6000 Server; the machine uses 248 Mhz SPARCv9 processors and runs Solaris 5.8. The difference between the

longest and shortest run was typically less than 5%; the exception is PrsTransTbl, where the longest run took almost six seconds and the shortest run took 1.2 seconds.

In general, the measurements in Table 3 show that flow-dependence inference and scenario extraction is reasonably fast, especially when compared to the times to read traces reported in Chapter 6. The measurements also show that inference time is roughly linear in the number of retained events and extraction time is roughly linear in the number of extracted scenarios. The relationships are not perfect; for example, the traces for XSetSelOwner retain about three-fifths as many events as the traces for XSaveContext, yet flow-dependence inference takes over three times as long for XSetSelOwner. Still, inference and extraction both appear to scale well as traces become longer and more scenarios are extracted.

Chapter 5

Learning specifications

Learning is inductive inference: given some observations, the learner's goal is to find a general rule that explains those observations and also predicts new observations. This chapter describes how Strauss learns a specification (the general rule), given some training scenarios (the observations).

Section 2.4 (page 32) defined a specification as a template with three parts: a seed pattern, a state transition model, and a scenario acceptor (SA). To learn a specification, Strauss only needs to learn the SA, because the seed pattern and state transition model are already given as inputs to Strauss.

The goal is to find the smallest SA that characterizes a set of acceptable scenarios. The difficulty is that the learner is not given a full description of the desired set. Instead, the learner is given a finite sample of training scenarios, possibly labeled by the specification developer as acceptable (that is, in the desired set) or unacceptable (that is, outside of the desired set). The Venn diagram in Figure 36 illustrates the relationships among the desired set and the two sets of training scenarios.

Put another way, despite the severe constraints imposed by limited information about the desired set, the learner aims to find an *accurate* and *understandable* SA:

Accurate An accurate SA accepts exactly the scenarios in the desired set.

Without a full description of the desired set, a learner can never be sure that it has found an accurate SA. However, it is possible to identify *necessary* conditions for accuracy:

- An accurate SA accepts all acceptable training scenarios and rejects all unacceptable training scenarios.
- Section 5.1 of this chapter defines a natural notion of semantic equivalence for scenarios. If an accurate SA accepts (or rejects) a scenario, it must accept (or reject) all semantically equivalent scenarios.
- An accurate SA must accept scenarios that are simple variations of acceptable training scenarios. This condition is fuzzier than the first two conditions; as an example, suppose that every acceptable training scenario opens a file, reads from the file several times, and finally closes the file. A scenario that opens a file, reads from the file six times, and then closes the file should probably be accepted, even if no training scenario reads from the file exactly six times.

Understandable An understandable SA is small. Understandable SAs are important, because, in the end, a person must inspect and validate each mined specification.

For example, suppose that the learner finds two SAs, both of which (accurately) accept all scenarios that open a file, read from the file some variable number of times, and finally close the file:

```
(F = open(); (read(f = F))*; close(f = F))
```

```
(F = open(); (read(f = F))*; (read(f = F))*; close(f = F))
```

The first SA is more understandable than the second.

Finally, in addition to yielding accurate and understandable specifications, learning must be reasonably efficient, and scale well as the number of scenarios increases.

My solution

Strauss reduces the problem of learning an SA from training scenarios to the problem of learning a non-deterministic finite automaton (NFA) from training strings. I chose this tactic because learning SAs is a new and tricky problem, while learning NFAs is a well-studied problem, which is tricky in a similar way. In particular, the same problem of limited information (see Figure 36) arises in both contexts.

Typically, NFA learners deal with the "limited information" problem by defining a new (but related) problem, which can be solved. The approaches are myriad and I will not survey them here—Kevin P. Murphy has written a good survey [37]. My point is that, by reducing SA learning to NFA learning, I exploit this hard work rather than repeating it.

Figure 37 shows my reduction of SA learning to NFA learning. The reduction has two halves. The top of the figure shows the first half, which translates scenarios (the inputs of SA learning) to strings (the inputs of NFA learning). The bottom of the figure shows the second half, which translates an NFA (the output of NFA learning) to an SA (the output of SA learning).

Both halves of my reduction are based on a novel algorithm for converting scenarios to strings, called *standardization*. Standardization is a many-to-one mapping from scenarios to strings, which ideally maps semantically equivalent scenarios to the same "standard" string (as I explain in Sections 5.1 and 5.2, my standardization algorithm achieves scalability by approximating this ideal). The reduction uses standardization as follows:

- The first half of the reduction uses standardization directly to translate scenarios into strings.
- The second half implements an SA by using the NFA (from the NFA learner) as a subroutine. Given a scenario, the SA uses standardization to produce a string. The SA accepts the scenario iff the NFA accepts the string.

Within the constraints imposed by limited information and scalability, my reduction leads to accurate and understandable SAs, for the following reasons:

Accurate I listed three necessary conditions for accurate SAs above. My reduction depends on the NFA learner to ensure the first condition (the training set is accepted properly) and the third condition (the SA accepts simple variations of the acceptable training scenarios). NFA learners are designed to satisfy these conditions. In particular, most learners can learn NFAs with alternations and loops, which is required to handle examples like the one discussed above, in which a program may open a file and read from it an arbitrary number of times before finally closing it.

Standardization tries to ensure the second necessary condition (semantically equivalent scenarios are accepted or rejected together). Unfortunately, as

I demonstrate in Section 5.1, any algorithm that ensures the second condition may have to examine the entire trace to standardize a single scenario. Therefore, Strauss uses a heuristic algorithm, which processes each scenario in isolation yet handles common cases well.

Understandable My reduction takes advantage of the tremendous work that others have done to develop learners that find small NFAs. Also, because standardization is a many-to-one mapping from scenarios to strings, my reduction can supply fewer distinct strings to the NFA learner than there are distinct scenarios. The experiments in Section 5.4 demonstrate that this reduction in the size of the training set leads to smaller NFAs, especially when the original training set is large.

My reduction of SA learning to NFA learning is efficient and scalable, because standardization processes each scenario in isolation. The worst-case running time of standardization is exponential in the number of events within a single scenario, but in practice scenarios are small and standardization is fast. I found (see Section 5.4) that the running time of standardization is an insignificant part of the running time of Strauss.

The rest of this chapter is structured as follows. Section 5.1 defines semantic equivalence for traces and scenarios, and demonstrates that fully accurate standardization cannot scale. Section 5.2 explains my heuristic standardization algorithm. Section 5.3 discusses the problems that arise in NFA learning, describes the off-the-shelf learner I use in Strauss, and explains why I chose it. Finally, Section 5.4 presents experiments that measure the cost and effectiveness of learning.

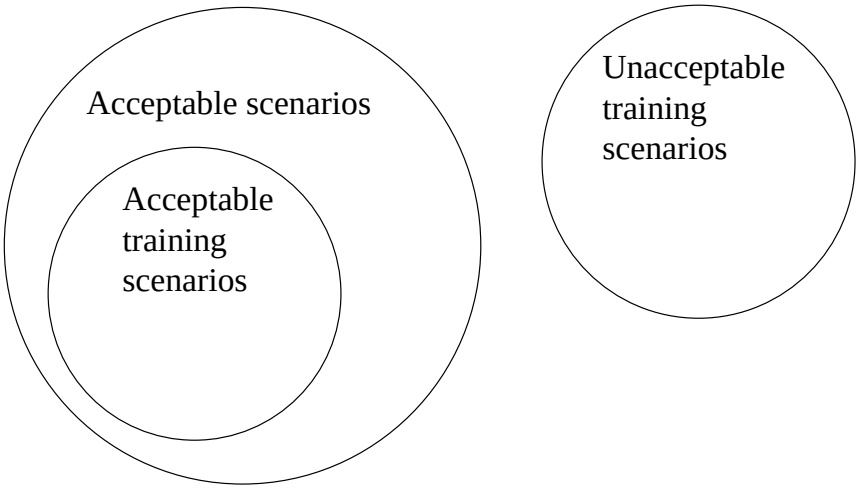

Figure 36: The desired set of acceptable scenarios (the output of SA learning). This set is a superset of the acceptable training scenarios, and is disjoint from the unacceptable training scenarios.

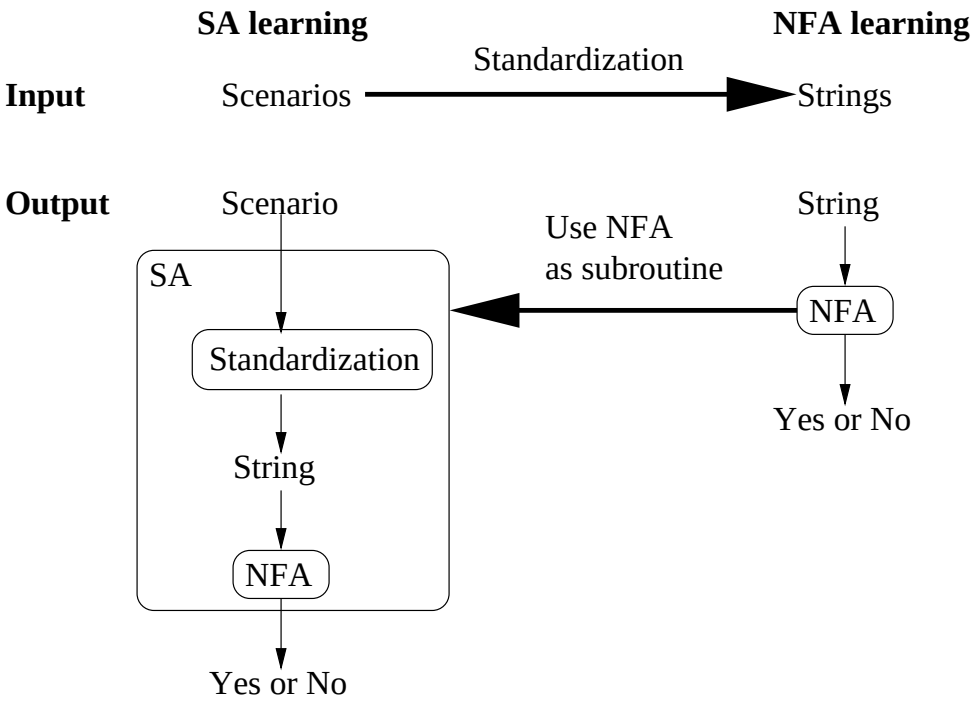

Figure 37: Reducing SA learning to NFA learning.

5.1 Semantic equivalence

An accurate SA accepts a scenario iff it accepts all semantically equivalent scenarios. In this section, I first define two natural notions of semantic equivalence for semantic traces, with respect to the semantics in Figure 27 (page 52). The first notion is more accurate than the second, but less suited to defining semantic equivalence for scenarios. Next, I define semantic equivalence for scenarios in terms of the second definition of semantic equivalence for traces. Finally, I present a simple example, which demonstrates that deciding semantic equivalence for small scenarios can require examining a large number of trace events; the implication is that a scalable standardization algorithm cannot decide semantic equivalence exactly.

5.1.1 Semantic equivalence for semantic traces

I say that two semantic traces are *weakly semantically equivalent* iff, starting from the initial semantic trace state$\{\text{pc} : 0, \text{vstates} : \text{ZeroMap}, \text{stack} : [0]\}$ (see Section 3.2.2, page 51), interpreting both traces produces the same final state, ignoring differences in the pc or in the particular values that index the vstates map. That is, semantic traces P_0 and P_1 are weakly semantically equivalent iff there is a one-to-one and onto map Γ from the values of P_0 to the values of P_1 such that

- Starting from the initial state, P_0 halts with $\{\text{pc} : pc_0, \text{vstates} : V_0, \text{stack} : [0]\}$.
- Starting from the initial state, P_1 halts with $\{\text{pc} : pc_1, \text{vstates} : V_1, \text{stack} : [0]\}$.

- For all values v, $V_0(v) = V_1(\Gamma(v))$. Here, two states are equal iff they are produced by the same tree of ⍰ applications.

For example, with this STM,

```
foo(def z, use x, use y)
bar(def x)
baz(def y)
```

these traces are weakly semantically equivalent:

```
call bar(x = 1); return bar()
call baz(y = 2); return baz()
call foo(z = 3, x = 1, y = 2); return foo()
```

and

```
call baz(y = 2); return baz()
call baz(y = 2); return baz()
call bar(x = 1); return bar()
call foo(z = 3, x = 1, y = 2); return foo()
```

Weak semantic equivalence is an awkward notion, because it asks us to interpret two traces to decide whether they are equivalent or not. It is also not well-suited to defining semantic equivalence for scenarios, because dead code has no effect on weak semantic equivalence: a scenario in dead code is equivalent to any other dead scenario.

A more useful notion is strong semantic equivalence. I say that two semantic traces are *strongly semantically equivalent* (or just *semantically equivalent*) iff they

have isomorphic flow dependence graphs. That is, P_0 and P_1 are strongly semantically equivalent iff there is a one-to-one and onto map Γ from the values of P_0 to the values of P_1 and a one-to-one and onto map σ from the events of P_0 to the events of P_1 such that

- If e_0 is an event in P_0, then replacing each value v in e_0 with $\Gamma(v)$ yields an event that is textually identical to $\sigma(e_0)$.

- $(e_{0,0}, e_{0,1})$ is a flow dependence in P_0 iff $(\sigma(e_{0,0}), \sigma(e_{0,1}))$ is a flow dependence in P_1.

Strong semantic equivalence implies weak semantic equivalence. Horwitz, Prins and Reps proved this fact for a rich imperative language with loops and conditionals [31]; their proof also applies to semantic traces, which are straight-line imperative traces (see page 71 of Section 4.1 for more discussion).

The converse is not true. For example, the weakly semantically equivalent traces above are not strongly semantically equivalent, because they have different numbers of events.

5.1.2 Semantic equivalence for scenarios

A scenario is a sequence of events from a semantic trace. The discussion above shows that two semantic traces can have the same *meaning*, even though they order events differently or refer to different values. The same holds for scenarios. This section defines a natural notion of semantic equivalence for scenarios, in terms of the strong semantic equivalence defined above for traces.

Two scenarios are *semantically equivalent* iff their corresponding traces can be mapped to strongly semantically equivalent traces, in which the two scenarios are textually identical. The intuition is that two scenarios are semantically equivalent iff there is some way to shuffle trace events or permute trace values so that the Scenarios function (see Section 4.2.1, page 73) returns both scenarios in exactly the same form.

Formally, a scenario S_0 (from a semantic trace P_0) is semantically equivalent to a scenario S_1 (from a semantic trace P_1) iff there is a $P_0{}'$ and a $P_1{}'$ such that

- P_0 and $P_0{}'$ are strongly semantically equivalent, with mappings Γ_0 and σ_0.
- P_1 and $P_1{}'$ are strongly semantically equivalent, with mappings Γ_1 and σ_1.
- $\sigma_0(S_0)$ and $\sigma_1(S_1)$ are textually identical. Here I have extended σ_0 and σ_1 to maps on scenarios in the obvious way: for $i = 0$ and $i = 1$, $\sigma_i(S)$ is the scenario S with its seed e_{seed} replaced with $\sigma_i(e_{seed})$ and its sequence of events $[e_0, \ldots, e_n]$ replaced with the set of events $\{\sigma_i(e_0), \ldots, \sigma_i(e_n)\}$ in their proper sequence in $\sigma_i(P_i)$.

For example, the scenarios in Figure 38 are semantically equivalent, while the scenarios in Figure 39 are semantically inequivalent.

The second example illustrates an important point: scenarios with the same flow dependence graph can be semantically inequivalent. In the example, both scenarios have the same flow dependence graph. However, the surrounding traces have different flow dependence graphs. In particular, the flow dependence graph for the trace on the left forbids placing the `G` call before the `F` call, because that would break a flow dependence from the first `A` call to the `F` call. Note that there

call bar(x = 1); `return bar()`	`call baz(y = 2); return baz()`
call baz(y = 2); `return baz()`	**call baz(y = 2);** `return baz()`
call foo(z = 3, x = 1, y = 2) [seed]	**call bar(x = 1);** `return bar()`
return foo()	**call foo(z = 3, x = 1, y = 2) [seed]**
	return foo()

with STM

```
foo(def z, use x, use y)
bar(def x)
baz(def y)
```

Figure 38: Two equivalent scenarios, within their enclosing traces. Scenario events are in **bold**.

`call A(x = 0); return A()`	**call A(x = 1) [seed]; return A()**
call A(x = 1) [seed]; return A()	**call G(y = 0, x = 1); return G()**
call F(y = 0, x = 1); return F()	`call A(x = 0); return A()`
call G(y = 0, x = 1); return G()	**call F(y = 0, x = 1); return F()**

with STM

```
A(def x)
F(use y, use x)
G (def use y, use x)
```

Figure 39: Two inequivalent scenarios, within their enclosing traces. Scenario events are in **bold**.

is also a flow dependence from the first `A` call to the `G` call, so it is also illegal to place the `A` call between the `G` call and the `F` call. Similarly, the flow dependence graph for the trace on the right forbids placing the `F` call before the `G` call.

The upshot is that the embedding of scenarios within traces must be considered when deciding equivalence; the next section shows that this property means that no scalable standardization algorithm can decide semantic equivalence exactly.

5.1.3 Scalability

The example in Figure 39 demonstrates that deciding whether two scenarios are equivalent or not can require inspecting events that are outside of the scenarios. This section argues that, in general, deciding scenario equivalence can require inspecting an arbitrarily large number of events, limited only by the length of the trace. Thus, because the number of extracted scenarios can also grow linearly with trace length, a standardization algorithm that decides scenario equivalence must scale quadratically with trace length. In practice, this means that standardization can only approximate equivalence.

The argument goes as follows. Suppose that the following events are in a scenario:

e_0: `F(y = 10, ...)`

e_1: `G(y = 10, ...)`

and the STM contains

```
F(use y, ...)
G(def y, ...)
```

Suppose that e_1 follows e_0 in the semantic trace and that there is no chain of flow dependences from e_0 to e_1 or (of course) from e_1 to e_0. The question is: is there a semantically equivalent trace in which e_1 appears before e_0 (if so, I say that e_1 can be *scheduled* before e_0)?. This question might have to be answered to decide whether the scenario is equivalent to a scenario that has a `G` call before an `F` call.

In general, answering this question can require inspecting an arbitrarily large number of trace events, limited only by the length of the trace. To schedule e_1 before e_0, the definition of `y` that reaches e_0 (call this $e_{def(y,0)}$) must be schedulable after e_1. However, the definition of `y` at e_1 might reach one event, or ten events, or a million events, or any number of events, up to the number of events following e_1 in the trace. To schedule e_1 before $e_{def(y,0)}$, all of these events must be schedulable between e_1 and $e_{def(y,0)}$. Thus, for each event $e_{use(y,1)}$ that uses the definition of `y` at e_1, one must answer the following question: can $e_{use(y,1)}$ be scheduled before $e_{def(y,0)}$? That is, the number of events that must be examined to schedule the scenario correctly is arbitrarily large.

5.2 Standardization

My reduction from specification learning to NFA learning is implemented via a many-to-one map from scenarios to strings, called *standardization*. This section explains how standardization works.

First, I define an approximation of semantic equivalence called *pseudo-equivalence*.[1] With respect to semantic equivalence, pseudo-equivalence is incomplete (scenarios can be semantically equivalent and yet not pseudo-equivalent) and unsound (scenarios can be pseudo-equivalent and yet not semantically equivalent).

Next, I present a simple algorithm for standardization. The algorithm maps each class of pseudo-equivalent scenarios to a unique, standard string. That is,

[1] In earlier work (with my advisors James R. Larus and Rastislav Bodík), pseudo-equivalence was simply called "equivalence" [1]

```
return socket (def ?)
call bind (def use so)
call listen (def use so)
call accept (use so)
return accept (def ?)
call close (def fd)
call read (use fd)
call write (use fd)
```

Figure 40: A state transition model for the socket interface.

standardization guarantees that my specification learner accepts a scenario whenever it accepts a pseudo-equivalent scenario. After explaining the simple algorithm, I discuss improvements to it that reduce its running time (in practice).

I also discuss two extensions, which annotate standard strings with global information about the trace. The first extension annotates values with type information, and the second extension places a scenario in context by indicating which scenario values have reaching definitions outside of the scenario and which scenario values are live outside of the scenario. This global information helps developers understand scenarios and specifications, and is necessary for some specifications.

5.2.1 Pseudo-equivalence

This section defines *pseudo-equivalence*, which approximates semantic equivalence for scenarios. Informally, two scenarios are pseudo-equivalent if they have the same events and the same dependence graph, even if they reference different values or order events differently. In this section, I define pseudo-equivalence formally, using the state transition model in Figure 40 and the scenario in Figure 41 as a running

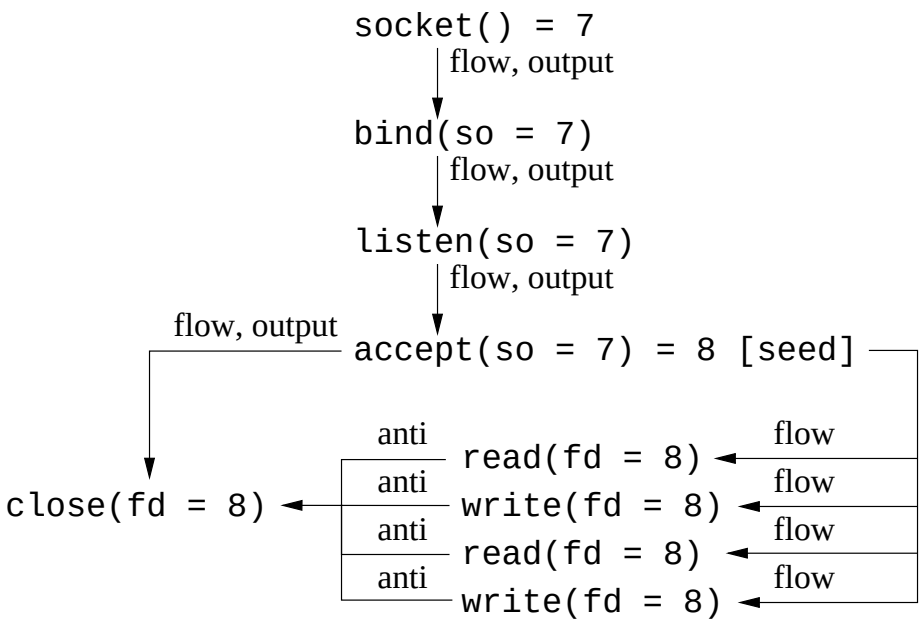

Figure 41: A scenario generated by a program that uses the socket interface.

example. With respect to the semantics in Figure 27 (page 52), my definition of pseudo-equivalence allows anomalies; I discuss one typical anomaly at the end of the section.

Anti- and output dependences

In addition to flow dependences, my definition of pseudo-equivalence uses anti- and output dependences. An *anti-dependence* connects an event that reads the state of a state variable to a later event that writes a new state into the variable. An *output dependence* connects an event that assigns a state to a state variable to a later event that also assigns a state to that variable. In Strauss, output and anti-dependences are carried only by arguments, not by the stack.

Figure 41 shows flow, anti-, and output dependences. For example, there is a

```
1  socket() = 7
2  bind(so = 7)
3  listen(so = 7)
4  accept(so = 7) = 8 [seed]
5  read(fd = 8)
6  read(fd = 8)
7  write(fd = 8)
8  write(fd = 8)
9  close(fd = 8)
```

Figure 42: A schedule for the scenario from Figure 41.

flow dependence and an output dependence from the `socket` call to the `bind` call because the `socket` call assigns a state to the state variable for `7` and the `bind` call reads and updates that variable. There are anti-dependences from the `read` and `write` calls to the `close` call because the `read` and `write` calls read the state of the state variable for `8`, and the `close` call overwrites that state.

I define a binary relation D_{any} on events: $(e, e') \in D_{\text{any}}$ iff there is a flow, anti-, or output dependence from e to e'.

Defining pseudo-equivalence

Let S be a scenario. A *schedule* σ of S is a sequence $s_0, s_1, \ldots, s_n$ that lists each event in S exactly once, such that if $(s_i, s_{i'}) \in D_{any}$, then $i < i'$. That is, a schedule never puts a dependence's sink before its source.

Figure 42 shows a schedule for the scenario in Figure 41. Notice that all dependences are preserved, and that the reads and writes on lines 5–8 appear in different orders in the two figures.

A *naming* Γ is a one-to-one map from values to symbolic names. I extend Γ to map events to events as follows: if e is an event, $\Gamma(e)$ is the event in which every

```
1  socket() = x0
2  bind(so = x0)
3  listen(so = x0)
4  accept(so = x0) = x1 [seed]
5  read(fd = x1)
6  read(fd = x1)
7  write(fd = x1)
8  write(fd = x1)
9  close(fd = x1)
```

Figure 43: The schedule in Figure 42, with the value 7 replaced by the symbolic name `x0` and the value 8 replaced by the symbolic name `x1`.

value v in S is replaced with $\Gamma(v)$. I also extend Γ to map sequences of events to sequences of events, in the natural way. Next, I say that two sequences σ_0 and σ_1 are *pseudo-equivalent up to names* iff (i) for every naming Γ_0 there is a naming Γ_1 such that $\Gamma_0(\sigma_0) = \Gamma_1(\sigma_1)$ and (ii) for every naming Γ_1 there is a naming Γ_0 such that $\Gamma_0(\sigma_0) = \Gamma_1(\sigma_1)$.

The following lemma will be useful for proving the correctness of standardization algorithms:

Lemma 5.1 *The event sequences σ_0 and σ_1 are pseudo-equivalent up to names iff there is a naming Γ_0 and a naming Γ_1 such that $\Gamma_0(\sigma_0) = \Gamma_1(\sigma_1)$.*

Figure 43 shows the schedule in Figure 42, with the value 7 replaced by the symbolic name `x0` and the value 8 replaced by the symbolic name `x1`.

Now let S_0 and S_1 be scenarios. S_0 and S_1 are *pseudo-equivalent* iff (i) for every schedule σ_0 of S_0 there is a schedule σ_1 of S_1 such that σ_0 and σ_1 are pseudo-equivalent up to names and (ii) for every schedule σ_1 of S_1 there is a schedule σ_0 of S_0 such that σ_0 and σ_1 are pseudo-equivalent up to names.

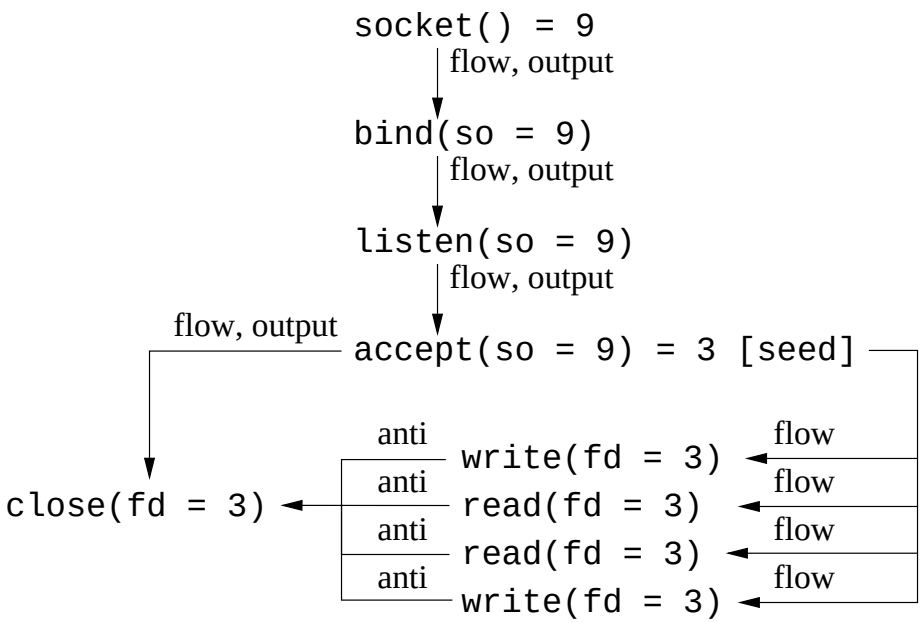

Figure 44: A scenario that is pseudo-equivalent to the scenario in Figure 41.

The following lemma will be useful for proving the correctness of standardization algorithms:

Lemma 5.2 *The scenarios S_0 and S_1 are pseudo-equivalent iff there is a schedule σ_0 of S_0 and a schedule σ_1 of S_1 such that σ_0 and σ_1 are pseudo-equivalent up to names.*

The scenario in Figure 44 is pseudo-equivalent to the scenario in Figure 41.

An anomaly

My definition of pseudo-equivalence allows some semantic anomalies. For example, consider the following scenario:

```
1 call foo()
```

```
2 return foo()
3 call bar()
4 return bar()
```

The scenario has no arguments, so all of its dependences are stack-carried. Pseudo-equivalence does not respect stack-carried anti- and output dependences, so the only dependences are a flow dependence from line 1 to line 2 and a flow dependence from line 3 to line 4. Thus, according to my definitions, the following is an allowable schedule:

```
1 call foo()
3 call bar()
2 return foo()
4 return bar()
```

This schedule does not make semantic sense, because it matches a `bar` call with a `foo` return.

This anomaly could be avoided by adding a stack-carried anti-dependence from the `foo` return to the `bar` call. This makes sense, because the `bar` call updates a stack slot that was read by the `foo` return. However, adding such dependences leaves standardization unable to reorder *any* events: every scenario has just one schedule, because it is never possible to move a call before any preceding return.

The root of the problem is the reliance on output and anti-dependences. These dependences are not semantically necessary. Their only purpose is to simplify the definition of schedules; with these dependences, a schedule is simply a topological

sort of the scenario. The anomaly above demonstrates that without them, the definition is more complicated.

Note that the anomalous schedule above is semantically illegal because it mixes up two *live ranges* of a stack slot. Lines 1 and 2 are one live range, and lines 3 and 4 are another. Placing line 3 before line 4 breaks the live range for line 1.

5.2.2 A naive algorithm for standardization

This section presents a correct but naive algorithm for standardization; I discuss improvements to this algorithm in Section 5.2.3. A weakness of all of my standardization algorithms is discussed at the end of Section 5.2.2.

The goal of the algorithm is to find a unique, standard string for each pseudo-equivalence class of scenarios. The strategy is simple. First, I define a total ordering $\leq_\sigma$ on schedules. Then, given a scenario S and a set X of symbolic names, the algorithm searches for a schedule σ for S and a naming Γ that maps values in S to names in X such that $\Gamma(\sigma)$ is the minimum, according to the ordering. Finally, the algorithm converts $\Gamma(\sigma)$ into a string for the NFA learner.

The total ordering $\leq_\sigma$ is a lexicographic ordering. Any schedule can be converted into a unique string over an alphabet (such as ASCII) by writing down each event, in order, formatted in a standard way (for example, like the example in Figure 41). Then, $\sigma_0 \leq_\sigma \sigma_1$ iff the string for σ_0 is lexicographically smaller than the string for σ_1.

Figure 45 lists pseudocode for the naive algorithm. `Naive` generates all schedules of S and all namings of the values in S with the names in X, finds the schedule

```
MaxSize := maximum number of values in a scenario
X := a set of MaxSize symbolic names

Routine Naive(S)
  Schedules := all schedules of S
  Namings := all namings of the values in S with the names in X
  AllPairs := Schedules × Namings
  Let (σ, Γ) ∈ AllPairs be s.t. for all (σ', Γ') ∈ AllPairs, Γ(σ) ≤_σ Γ'(σ')
    Return ToLearnerString((σ, Γ))

L := an unbounded set of symbolic names
// LMap maps events to names in L
LMap := []

Routine ToLearnerString(s_0, ..., s_n)
  String := ''
  Foreach s_i in order
    If LMap[s_i] has not been set
      A := an unused name in L
      LMap[s_i] = A
    String := append(String, LMap[s_i])
  Return String
```

Figure 45: Naive standardization algorithm.

```
1  socket() = x0                 A
2  bind(so = x0)                 B
3  listen(so = x0)               C
4  accept(so = x0) = x1 [seed]   D
5  read(fd = x1)                 E
6  read(fd = x1)                 E
7  write(fd = x1)                F
8  write(fd = x1)                F
9  close(fd = x1)                G
```

Figure 46: The named schedule from Figure 42, with a corresponding string for the NFA learner on the far right.

σ and naming Γ that minimizes $\Gamma(\sigma)$, and returns a string that corresponds to $\Gamma(\sigma)$.

In Figure 45, `ToLearnerString` converts schedules into strings for the NFA learner. As far as the NFA learner is concerned, each event is simply a letter in the string, with no special meaning. Figure 46 illustrates this point. The far right hand side of the figure shows a letter that `ToLearnerString` could assign to the event on the left hand side of the figure. Notice that events are assigned the same letter iff they are equal. For example, the events on line 5 and 6 both receive the letter E.

`Naive` is correct: it assigns the same string to scenarios S_0 and S_1 iff they are pseudo-equivalent.

Suppose that `Naive` assigns the same string to S_0 and S_1. `ToLearnerString` is a one-to-one function, so `Naive` must have found a schedule σ_0 for S_0, a schedule σ_1 for S_1, and namings Γ_0 and Γ_1 such that $\Gamma_0(\sigma_0) = \Gamma_1(\sigma_1)$. By Lemma 5.1 and Lemma 5.2, S_0 and S_1 are pseudo-equivalent.

For the other direction, suppose that S_0 and S_1 are pseudo-equivalent. Suppose that `Naive` finds that $\Gamma_0(\sigma_0)$ is the minimum named schedule for S_0 and that $\Gamma_1(\sigma_1)$

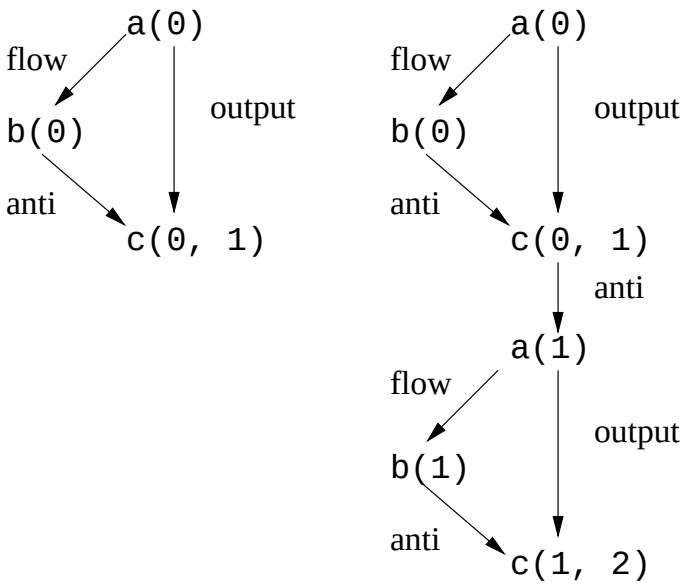

Figure 47: Two scenarios. The scenario on the left is pseudo-equivalent to two subscenarios of the scenario on the right.

is the minimum named schedule for S_1. Suppose that $\Gamma_0(\sigma_0) \not\leq_\sigma \Gamma_1(\sigma_1)$. Because S_0 and S_1 are pseudo-equivalent, there is a schedule $\sigma_0{}'$ for S_0 with a naming $\Gamma_0{}'$ such that $\Gamma_0{}'(\sigma_0{}') = \Gamma_1(\sigma_1)$. But, then $\Gamma_0(\sigma_0) \leq_\sigma \Gamma_0{}'(\sigma_0{}')$, a contradiction. So, $\Gamma_0(\sigma_0) leq_\sigma \Gamma_1(\sigma_1)$ and, similarly, $\Gamma_1(\sigma_1) leq_\sigma \Gamma_0(\sigma_0)$. Thus, $\Gamma_0(\sigma_0) = \Gamma_1(\sigma_1)$. Since `ToLearnerString` is a function, `Naive` assigns the same string to S_0 and S_1.

Unfortunately, the running time of `Naive` is exponential in $|X|$ and $|S|$, because it builds and evaluates all namings and all schedules.

A weakness of standardization

Standardization maps pseudo-equivalent scenarios to the same string, but if two scenarios have pseudo-equivalent subscenarios, the subscenarios are not necessarily mapped to identical subsequences. For example, the scenario on the left side of

Figure 47 is pseudo-equivalent to two subscenarios of the scenario on the right side of the figure. Because standardization assigns a different name to each distinct value that appears in the scenario on the right, it cannot produce strings that expose both pseudo-equivalent subscenarios to the NFA learner.

This is not a serious weakness of standardization per se, because in fact no algorithm that maps scenarios to strings can solve this problem.

5.2.3 Optimizations of the naive algorithm

This section presents two optimizations that improve the efficiency (in practice) of the naive algorithm. The first optimization improves efficiency by reducing the number of namings considered to one naming; this optimization also incorporates a heuristic that increases understandability by biasing standardization towards using a small number of distinct letters to represent events near the seeds of scenarios. The second optimization improves efficiency by reducing the number of schedules considered by taking $\leq_\sigma$ into account during schedule generation.

Considering one naming

The algorithm `Better` in Figure 48 removes the exponential behavior in $|X|$ of `Naive` by considering only one standard naming per schedule. For each schedule, `StdNaming` deterministically constructs the standard naming.

This optimization is safe: S_0 and S_1 are pseudo-equivalent iff `Better` assigns them the same string. Clearly, if `Better` assigns the same string to S_0 and S_1, then S_0 and S_1 are pseudo-equivalent, by an argument like the argument I used for `Naive`.

```
X := an unbounded, ordered set of symbolic names

Routine AugmentNaming(Γ, e)
  Foreach argument a in e
    v := ValueAt(a)
    If Γ(v) is undefined
      Define Γ(v) as the first name in X not already used by Γ
  Return Γ

Routine StdNaming(S = s_0, ..., s_n)
  Γ := ∅
  i_s := index of the seed in S
  Γ := AugmentNaming(Γ, s_{i_s})
  dist := 1
  While i_s − dist ≥ 0 or i_s + dist ≤ n
    If i_s − dist ≥ 0
      Γ := AugmentNaming(Γ, s_{i_s − dist})
    If i_s + dist ≤ n
      Γ := AugmentNaming(Γ, s_{i_s + dist})
    dist := dist + 1

Routine Better(S)
  Schedules := all schedules of S
  StdPairs := {(σ, Γ) | Γ = StdNaming(σ), σ ∈ Schedules}
  Let (σ, Γ) ∈ StdPairs be s.t. for all (σ', Γ') ∈ StdPairs, Γ(σ) ≤_σ Γ'(σ')
    Return ToLearnerString((σ, Γ))
```

Figure 48: The algorithm of Figure 45, optimized to consider only one naming per schedule.

The other direction holds because the names assigned by a schedule's standard naming do not depend on the identities of the argument values, but only on where those values appear in the schedule. That is, if σ_0 and σ_1 are pseudo-equivalent up to names with $\Gamma_0 = \texttt{StdNaming}(\sigma_0)$ and $\Gamma_1 = \texttt{StdNaming}(\sigma_1)$, then $\Gamma_0(\sigma_0) = \Gamma_1(\sigma_1)$. So, if S_0 and S_1 are pseudo-equivalent, the least named schedule that `Better` finds for S_0 must be identical to the least named schedule that `Better` finds for S_1.

`Better` is more efficient than `Naive`, but it also increases the understandibility of specifications. I assume that the user is most interested in the events that are near seeds. `StandardName` assigns names to the values that appear in the seed first, and then works outward. Thus, given two scenarios S_0 and S_1 (which may or may not be pseudo-equivalent), `Better` finds named schedules where the seeds and events near the seeds are likely to be equal. Effectively, `Better` reduces the complexity of the strings it gives to the NFA learners exactly where I assume that the user is most likely to focus his attention.

Considering fewer schedules

The worst-case running time of `Better` is still exponential in $|S|$. Better performance is possible in the common case, since scenarios are not arbitrary DAGs. In particular, events and their arguments have names. The algorithm in Figure 49 uses those names to reduce the number of schedules it considers.

`BetterStill` considers only schedules that put events in the least order under $\leq_\sigma$, with values ignored. Although there can be an exponential number of such orderings, there is often only one. In that case, the set of events `Least` in

```
Routine BlankVals(e)
  Return an event like e, but with values replaced by "_".

Routine RestrictedSchedules(S)
  Schedules := ∅
  Ready := {s ∈ S | ¬∃s' ∈ S s.t. d(s', s)}
  Least := {s ∈ Ready | ¬∃s' ∈ Ready s.t. BlankVals(s') ≤_σ BlankVals(s)}
  Foreach s ∈ Least
    Rest := RestrictedSchedules(S − {s})
    Foreach σ_r ∈ Rest
      σ := concat(s, σ_r)
      Schedules := Schedules ∪ {σ}
  Return Schedules

Routine BetterStill(S)
  Schedules := RestrictedSchedules(S)
  StdPairs := {(σ, Γ) | Γ = StdNaming(σ), σ ∈ Schedules}
  Let (σ, Γ) ∈ StdPairs be s.t. for all (σ', Γ') ∈ StdPairs, Γ(σ) ≤_σ Γ'(σ')
    Return ToLearnerString((σ, Γ))
```

Figure 49: The algorithm of Figure 48, with the schedules considered restricted with $\leq_\sigma$.

Original scenarios

```
1  A(x = 0:T0, y = 1:T1) [seed]    E(x = 0:T0, v = 2:T2) [seed]
2  B(x = 0:T0, y = 1:T1)           B(x = 0:T0, y = 1:T1)
3  C(x = 0:T0, y = 1:T1)           C(x = 0:T0, y = 1:T1)
```

Named with one namespace for all types

```
1  A(x = x0, y = x1) [seed]        E(x = x0, v = x1) [seed]
2  B(x = x0, y = x1)               B(x = x0, y = x2)
3  C(x = x0, y = x1)               C(x = x0, y = x2)
```

Named with a separate namespace for each type

```
1  A(x = x0:T0, y = x0:T1) [seed]  E(x = x0:T0, v = x0:T2) [seed]
2  B(x = x0:T0, y = x0:T1)         B(x = x0:T0, y = x0:T1)
3  C(x = x0:T0, y = x0:T1)         C(x = x0:T0, y = x0:T1)
```

Figure 50: Two nearly pseudo-equivalent scenarios and their scenario strings, with untyped and typed naming.

`RestrictedSchedules` always has one element, the recursion never branches, and `BetterStill` runs in time proportional to $n \log n$, assuming an implementation that keeps the `Ready` and `Least` sets in sorted order and updates them intelligently.

`BetterStill` is correct, because the schedules generated by `RestrictedSchedules` do not depend on the identities of argument values, but only on the scenario's dependence graph and the names of events and arguments.

5.2.4 Extensions

Naming with types

This section presents an algorithm for annotating scenario values with inferred types. Type annotations help the specification developer understand scenarios and traces by succinctly summarizing which event arguments interact (globally, in

the entire trace) with other event arguments.

Type annotations can also improve learning by reducing naming conflicts, thus exposing more scenario similarities to the learner. Figure 50 shows a motivating example. The top of the figure lists two scenarios, with their values annotated with types. Notice that line 1 differs between the two scenarios, while lines 2 and 3 are identical.

The standardization algorithms presented so far use one namespace for all values, and would produce the named scenarios in the middle of the figure. Notice that lines 2 and 3 are not identical in the named scenarios.

By contrast, an algorithm that used a separate namespace for each inferred type would produce the named scenarios at the bottom of the figure. Each scenario refers to exactly one value of each of the three types, so exactly one name is assigned from each namespace. With this optimization, lines 2 and 3 are once again identical.

Type inference analyzes the argument flow dependence graph to find a typing Γ that assigns a type variable to each access path. The typing separates access paths that never interact through flow dependences: if Γ assigns p and p' separate type variables, then no argument accessed by p is connected by argument flow dependences to an argument accessed by p'. Type inference infers the most general typing Γ that satisfies this condition:

$$\text{If } (a, a') \in D_{arg}, \text{ then } \Gamma(\text{Path}(a)) = \Gamma(\text{Path}(a')).$$

where a typing Γ_0 is more general than a typing Γ_1 iff some substitution for the type variables of Γ_0 makes $\Gamma_0 = \Gamma_1$.

```
NameSpaces := a family of disjoint, unbounded, ordered sets of symbol names,
              indexed by inferred types

Routine TypedAugmentNaming(Γ, e)
  Foreach argument a in e
    v := ValueAt(a)
    X := NameSpaces[Type(v)]
    If Γ(v) is undefined
      Define Γ(v) as the first name in X not already used by Γ
  Return Γ
```

Figure 51: A replacement for `AugmentNaming` (Figure 48) that uses a different namespace for each inferred type.

The inference algorithm uses Tarjan's union-find algorithm [49] and requires time nearly linear in the length of the trace. The algorithm starts with an initial typing that gives each access path its own unique type variable. Then, the algorithm visits each dependence $(a, a') \in D_{arg}$ and unifies the types of Path(a) and Path(a'). Type inference is complete when all argument flow dependences have been visited.

Figure 51 lists pseudo-code for a replacement for the `AugmentNaming` routine of Figure 48 that assigns names from separate namespaces for each inferred type.

Reaching definitions and liveness

Scenarios are local and limited in size, but some specifications require knowing if there is an event "over the horizon", which defines a state variable that is read in the scenario or reads a state variable that is defined in the scenario. For example, consider this STM:

with STM

```
A(def x, def y)
B(use x, use y)
```

and this scenario:

```
call A(x = 1, y = 0)
call A(x = 0, y = 3)
call B(x = 0, y = 1) [seed]
```

Suppose that, as a rule, every pair of values passed as `x` and `y` to a `B` call must have been previously passed as `x` and `y` to an `A` call (this is an instance of the TiedConsumer specification pattern; see Section B.0.6, page 249). It is impossible to tell whether the above scenario obeys the rule or not. Moreover, increasing the backwards-radius of scenario extraction would not help because no argument flow dependences target calls to `A`.

I have implemented a partial solution for this problem in Strauss. Optionally, Strauss adds *context* to scenarios as it standardizes them. When this option is turned on, Strauss places pseudo-events of the form `DEFINE(X)` at the beginning of each standardized scenario and pseudo-events of the form `USE(X)` at the end. The former indicates that the value that has been assigned the name X is defined by a preceding event that is not in the scenario; the latter indicates that the value is used by a succeeding event that is not in the scenario.

If the result of standardizing the above scenario with context is

```
DEFINE(X)
DEFINE(Y)
```

```
call A(x = Y, y = X)
call A(x = X, y = Z)
call B(x = X, y = Y) [seed]
```

then the scenario may or may not obey the above rule. However, if the result is (for example)

```
DEFINE(X)
call A(x = Y, y = X)
call A(x = X, y = Z)
call B(x = X, y = Y) [seed]
```

then the scenario definitely does *not* obey the rule.

5.3 NFA learning

Strauss learns accurate and understandable specifications by reducing specification learning to NFA learning (Figure 37). The preceding section described the standardization algorithm that plays the key role in the reduction. This section discusses the target of the reduction, NFA learning.

NFA learning is a large field, and I do not attempt to survey it here. Instead, I concentrate on Raman and Patrick's *sk-strings* learner [43], which is the off-the-shelf learner that I chose to use in Strauss. Strauss could use any NFA learner, but I chose sk-strings for three reasons:

- The introduction to this chapter explained that a fundamental challenge in NFA learning is formulating a workable statement of the problem. Raman

and Patrick formulate the problem as a search for an automaton that minimizes a measure called the *minimum message length* (MML). This is a useful formulation, because MML can be used as a measure of how well a specification fits a set of training scenarios.

- The sk-strings learner is extremely fast. Raman and Patrick report that the learner never took longer than 0.3 seconds to learn NFAs with up to 100 states and 500 transitions, while their implementation of a popular competing learner took longer than an hour for the same data set.[2] Such NFAs are much larger than the typical NFAs generated by Strauss.

- Raman and Patrick have distributed a free implementation of their algorithm.

The rest of this section has two parts. The first part explains Raman and Patrick's formulation of the NFA-learning problem, and the second part explains the sk-strings algorithm.

5.3.1 NFA learning: Raman and Patrick's formulation

Given a set of acceptable strings (and possibly also a set of unacceptable strings), an NFA learner finds an NFA that accepts the acceptable strings and other "similar" strings. This is a lousy problem statement: what does "similar" mean? In fact, as noted in the introduction to this chapter, NFA learners typically define and solve a new, but related, problem.

[2]The competing learner was the k-tails learner. I believe that Raman and Patrick's implementation of that learner was inefficient, since they say that k-tails is exponential in the tail size k; in fact, k-tails is closely related to DFA minimization and can be implemented in time independent of k along the lines of Hopcroft and Ullman's implementation of Moore's algorithm [30].

Raman and Patrick make two changes to the statement. First, they state the problem as a search for a probabilistic finite state automaton (PFSA) instead of as a search for an NFA. I defined PFSAs in Section 2.2 (page 26); essentially, a PFSA is an NFA with integral weights on its transitions. The weights represent execution frequencies, from which probabilities can be computed: if the automaton is in a state q, the probability that it will take transition (q, q') next is given by the weight $W(q, q')$ divided by the sum of the weights of all transitions from q.

Second, the goal of their search is a machine that minimizes an objective measure called the minimum message length (MML), not merely a machine that accepts the training strings and "similar" strings. The MML is the sum of the length of an encoding of the inferred PFSA plus the length of an encoding of the training set, assuming that it was generated by the PFSA (see below for the formula).

The importance of the MML is that it is an objective and easily computed measure of the goodness of a PFSA that captures two intuitive notions:

- A small automaton is better than a large automaton. The MML incorporates this notion via the number of bits needed to encode the PFSA.

- An automaton that encodes common strings very efficiently is better than an automaton that encodes common strings in some complicated manner, even if the latter automaton outperforms the former on some rare strings. The MML incorporates this notion via the number of bits needed to encode the training set.

Because the MML is easily computed, it is practical to use it to compare several PFSAs for a particular training set. However, Raman and Patrick note that the

search for a PFSA that minimizes the MML is exponential. Consequently, their sk-strings algorithm uses the MML to choose among PFSAs generated by several competing heuristics (see Section 5.3.2).

I also use the MML as a measure of a PFSA's goodness. In Section 5.4, I use the MML to measure how much scheduling simplifies PFSAs, compared to listing scenario events in their original trace order.

The MML

Raman and Patrick give this formula for the MML:

$$\sum_{q \in Q} \left\{ m_q + \log \frac{(t_q - 1)!}{(m_q - 1)! \prod\limits_{q' \in \bigcup_{a \in \Sigma} \delta(q,a)} (W(q,q') - 1)!} + m_q \log |\Sigma| + {m_q}' \log |Q| \right\}$$
$$- \log((|Q| - 1)!)$$

where $|Q|$ is the number of states in the PFSA, $t_q = \sum_{q' \in Q} W(q', q)$ is the total weight of the transitions into q, $|\Sigma|$ is the cardinality of the alphabet including a special delimiter symbol (Raman and Patrick use this symbol to indicate the end of a string), $W(q, q')$ is the weight of the transition from q to q', $m_q = |\cup_{a \in \Sigma} \delta(q, a)|$ is the number of transitions from q, and ${m_q}'$ is the number of different transitions on non-delimiter symbols from the q. The logs are base 2 and the MML is in bits.

The interested reader should read Raman and Patrick's paper for a complete discussion of this formula. Briefly, the terms of the sum count the bits required to encode the PFSA and the dataset, under some assumptions:

- The first term under the sum counts the bits required to encode the number of transitions from q, assuming that the probability that a state has n outgoing

transitions is 2^{-n}.

- The second term under the sum counts the number of bits required to specify the particular distribution of t_q transitions into q onto the m_q transitions out of q. This is the term that counts the cost of encoding the training set.

- The third term under the sum counts the number of bits required to specify the symbols on the transitions out of q.

- The fourth term under the sum counts the number of bits required to specify the targets of the transitions out of q. The formula assumes that transitions labeled by the delimiter symbol always target q_f, the final state.

- The subtracted term accounts for the fact that the sum counts the bits required to specify a particular permutation of the PFSA's states, when in fact all permutations are equivalent.

5.3.2 The sk-strings learner

The sk-strings learner is actually a family of heuristic learners, all of which are modifications of the seminal k-tails learner [4]. This section first explains how the k-tails learner works and then explains how the sk-strings learners modify it.

The k-tails learner

The k-tails learner learns an NFA from a training set of strings. The learner has an initialization phase and a merging phase. The initialization phase constructs a retrieval tree from the input strings. The retrieval tree is a deterministic finite

automaton (DFA) that recognizes the training set, but is quite large and does not accept any strings that are not in the training set.

The merging phase produces a smaller, possibly nondeterministic, automaton that accepts the training set and possibly other strings. This phase is closely related to Moore's algorithm for minimizing a DFA, in the following sense. Moore's algorithm merges all states that are *equivalent*:

> Let q_0 and q_1 be two states of a DFA. q_0 and q_1 are *equivalent* iff for all strings w, the DFA accepts w starting from q_0 iff it accepts w starting from q_1.

By contrast, the merging phase of the k-tails learner merges all states that are *k-equivalent*.

> Let q_0 and q_1 be two states of a DFA. q_0 and q_1 are *k-equivalent* iff for all strings w such that $|w| \leq k$, the DFA accepts w starting from q_0 iff it accepts w starting from q_1.

A string w is called a *k-tail* of a state q iff $|w| \leq k$ and the DFA accepts w starting from q. Thus, q_0 and q_1 are k-equivalent iff they have the same k-tails.

Hopcroft and Ullman give an implementation of Moore's algorithm that runs in time $O(|\Sigma|n^2)$, where n is the size of the initial DFA and $|\Sigma|$ is the size of the alphabet [30]. Their algorithm can easily be modified to find all k-equivalence classes instead of all equivalence classes, without changing the running time.

From k-tails to sk-strings

Raman and Patrick's sk-strings learners make five changes to k-tails:

1. The sk-strings learners learn PFSAs instead of NFAs.

2. The retrieval tree that is the input of the merging phase is considered to be a PFSA, not a DFA.

3. The sk-string learners consider the distribution of *k-strings* of PFSA states, instead of the k-tails of DFA states. A string w is a k-string of a state q iff

   - $|w| \leq k$ and the PFSA accepts w starting from q (that is, w is a k-tail of q); or
   - $|w| = k$ and the PFSA is defined on w starting from q. That is, $\delta(q, w)$ is defined.

4. The sk-string learners replace the concept of k-equivalence with sk-equivalence. Two states q and q' are *sk-equivalent* iff their k-string distributions are close according to a heuristic, where the heuristic varies from one sk-string learner to another.

   All of the heuristics sort the k-strings of each state in order of descending probability. At each state, the heuristics inspect the smallest set of top k-strings whose probabilities add up to at least $s\%$. The following lists, for each heuristic, the conditions under which the heuristic considers q and q' to be sk-equivalent:

   OR The top $s\%$ of the k-strings of q are k-strings of q' *or* vice-versa.

   AND The top $s\%$ of the k-strings of q are k-strings of q' *and* vice-versa.

LAX The top $s\%$ of the k-strings of q are the same as the top $s\%$ of the k-strings of q', and the top k-strings are in the same order (by probability) at both states.

STRICT The top $s\%$ of the k-strings of q are the same as the top $s\%$ of the k-strings of q', and the top k-strings are in the same order (by probability) at both states. Moreover, the probability distribution of the top k-strings is the same at both states.

5. The sk-string learners replace k-tails's merging phase with a simple greedy algorithm, which merges pairs of sk-equivalent states, in an arbitrary order, until no sk-equivalent pairs remain. The tricky part is recomputing the sk-equivalence relation after a merge; see Raman and Patrick's paper for an explanation of this part of their algorithm.

Raman and Patrick expect that users will run several different sk-string learners, varying the heuristic and the values of s and k, and choosing the PFSA with the lowest MML. Their paper compares the performance of the four heuristics above with s fixed at 50% and with k varying from 1 to 10. On their training data, all of heuristics except STRICT occasionally produced the PFSA with the lowest MML, with the OR heuristic finding the best MML on 85 out of 100 test cases. The best values for k were 1 and 2, although every value up to and including 5 occasionally produced the PFSA with the lowest MML.

Strauss's learner tries all of their heuristics, with s fixed at 50% and k ranging from 1 to 6.

5.4 Experiments

This section presents experimental results for specification learning, which I gathered while mining 17 specifications about the X11 interface. Each specification was mined from the same set of 90 traces, gathered from full runs of 72 programs (see Chapter 6 for more information about these traces). Measurements of 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. Time measurements report the best time for three runs.

All numbers are for initial specifications, before I used Cable (see Chapter 7) to debug them.

Table 4 shows the cost of scheduling and naming. The time measurements did not vary significantly. Ignoring measurements for scheduling and naming where the shortest time was longer than 2 seconds, the difference between the longest and shortest time was typically less than 10%; the exception was naming XSaveContext, where the shortest time was 196.8 seconds and the longest time was 219.4 seconds (a 12% difference).

Table 5 shows some measurements of the performance of NFA learning. The time measurements for NFA learning varied more than did the measurements for scheduling and naming. The worst case was RegionsAlloc, where the longest time was 8.8 seconds and the shortest time was 7.2 seconds (a 23% difference).

The times in Tables 4 and 5 make up a small portion of the total running time of Strauss. For purposes of comparison, Table 6 lists the time Strauss spent reading traces while mining each specification, as well as the total end-to-end time

Name	Scenarios	Schedules	Time (s)	
			Sched	Name
PrsAccelTbl	3640	3640	0.1	2.3
PrsTransTbl	2410	2410	0.1	1.4
Quarks	17409	18129	1.7	11.3
RegionsAlloc	33072	33072	2.6	19.8
RegionsBig	110574	110652	16.8	75.8
RmvTimeOut	803	803	0.0	0.5
XFreeGC	2817	2817	0.2	1.8
XGContextFromGC	3079	3080	0.2	1.9
XGetSelOwner	146	146	0.0	0.1
XInternAtom	15678	15678	0.8	9.4
XPutImage	2821	2821	0.3	1.8
XSaveContext	298008	298128	31.4	196.8
XSetFont	33008	33204	1.9	20.4
XSetSelOwner	146	146	0.0	0.1
XtFree	275407	275407	20.4	163.5
XtOwnSel	376	376	0.0	0.2
XtRealizeProc	2435	2435	0.1	1.6

Table 4: Cost of scheduling and naming. For each specification, **Scenarios** is the number of scenarios extracted from the training traces, **Schedules** is the number of schedules that were considered by standardization, **Time Sched** is the time in seconds spent scheduling scenarios, and **Time Name** is the time in seconds spent naming scenarios.

to extract and standardize scenarios. The time spent reading traces dwarfs the time spent otherwise. The only exceptions were `XSaveContext`and `XSetFont`: Strauss wasted time thrashing the virtual memory system on these specifications.

Table 7 shows the benefit of standardization in reducing the size of inferred NFAs, compared with a simpler standardization algorithm which does no scheduling. Standardization shows no benefit when the original NFAs are small: for these specifications, the scheduled order of most scenarios was the same as the trace order. However, it sometimes shows a significant benefit when NFAs are large: for RegionsBig, it reduced the number of states by more than half, and the number

Name	Trie		FA			Time (s)
	|Q|	**|T|**	**|Q|**	**|T|**	**MML (Kb)**	
PrsAccelTbl	10	19	3	10	12.2	6.7
PrsTransTbl	4	7	3	4	6.2	6.8
Quarks	117	165	63	93	73.2	15.5
RegionsAlloc	35	53	8	14	43.0	7.2
RegionsBig	1220	1487	370	604	506.9	788.2
RmvTimeOut	4	6	3	3	0.3	7.1
XFreeGC	17	27	10	13	12.3	7.4
XGContextFromGC	55	80	22	36	16.2	8.6
XGetSelOwner	10	14	5	6	0.3	6.8
XInternAtom	19	29	3	11	31.9	6.8
XPutImage	11	17	3	7	8.1	7.0
XSaveContext	2570	3066	989	1519	834.6	883.8
XSetFont	66	101	20	37	54.5	9.3
XSetSelOwner	12	19	3	8	0.4	6.8
XtFree	211	323	95	171	1222.7	21.2
XtOwnSel	30	54	4	23	1.2	6.9
XtRealizeProc	37	60	22	36	8.7	7.7

Table 5: Performance of NFA learning. For each specification, **Trie** is the number of states and transitions in the sk-string learner's initial retrieval tree, **FA** is the number of states and transitions and the number of kilobits in the MML of the sk-string learner's final FA, and **Time** is the time in seconds spent running all of the sk-string learners.

of transitions by more than one third. On the other hand, the benefits were minor for the largest specification, XSaveContext. Standardization never significantly reduced the MML.

Name	**Time (s)**	
	Construct	**Total**
PrsAccelTbl	13220.1	13298.6
PrsTransTbl	13280.7	13337.6
Quarks	13258.4	13369.0
RegionsAlloc	13186.3	13379.4
RegionsBig	13223.8	14079.0
RmvTimeOut	13220.2	13271.5
XFreeGC	13278.0	13327.6
XGContextFromGC	13168.1	13220.1
XGetSelOwner	13420.7	13459.1
XInternAtom	12930.1	13235.4
XPutImage	13607.4	13703.6
XSaveContext	13376.2	51716.3
XSetFont	13131.0	16518.8
XSetSelOwner	12993.6	13417.2
XtFree	13216.1	14595.9
XtOwnSel	12995.5	13419.6
XtRealizeProc	13238.1	13307.1

Table 6: Time to read traces (**Construct**) and total end-to-end time to extract and standardize scenarios (**Total**).

Name	FA (no scheduling)			FA (standardization)		
	|Q|	**|T|**	**MML (Kb)**	**|Q|**	**|T|**	**MML (Kb)**
PrsAccelTbl	3	10	12.2	3	10	12.2
PrsTransTbl	3	4	6.2	3	4	6.2
Quarks	63	93	73.2	66	92	71.9
RegionsAlloc	8	14	43.0	8	14	43.0
RegionsBig	370	604	506.9	177	379	524.9
RmvTimeOut	3	3	0.3	3	3	0.3
XFreeGC	10	13	12.3	10	13	12.3
XGContextFromGC	22	36	16.2	22	36	16.2
XGetSelOwner	5	6	0.3	5	5	0.3
XInternAtom	3	11	31.9	3	11	31.9
XPutImage	3	7	8.1	3	8	10.3
XSaveContext	989	1519	834.6	961	1474	831.3
XSetFont	20	37	54.5	20	36	54.5
XSetSelOwner	3	8	0.4	3	8	0.4
XtFree	95	171	1222.7	83	144	1181.8
XtOwnSel	4	23	1.2	4	23	1.2
XtRealizeProc	22	36	8.7	22	36	8.7

Table 7: Benefit of standardization. For each specification, **No scheduling** and **Standardization** list the number of states and transitions and the number of kilobits in the MML of the sk-string learner's final FA; the former lists the results without scheduling, while the latter lists the results with full standardization.

Chapter 6

Tracing

Strauss mines specifications from execution traces, which capture the calls and returns—with the values they convey—through which a program interacts with a library. Section 2.3 (page 27) defined the syntax of traces and trace events, and I have referred to traces throughout this dissertation.

The chapter is organized as follows. First, Section 6.1 explains how I decide which events and values should be included in traces. Enough information must be included to enable Strauss to find useful specifications, but traces that record too much information take too much space and are hard to process. This section lays out simple principles, which I believe apply to many interfaces and specifications.

Next, Section 6.2 explains the implementation of a tracer I built for X11. The tracer instruments X11 applications so that they record traces as they run. The tracer does not depend on the availability of source code for the programs or for the X11 libraries. This decision, together some infelicities of C, raised some interesting engineering challenges.

Finally, Section 6.3 lists the traces that I used in the experiments in this dissertation.

6.1 What to trace

This section discusses what data should appear in the execution traces from which Strauss mines specifications. Gathering traces is expensive and sometimes requires human intervention, so recording the right events and values in traces is very important. Recording too little information greatly limits the specifications that Strauss can mine, but putting too much information into traces slows Strauss down.

6.1.1 Which calls and returns to trace

Strauss's traces capture the calls and returns that cross an interface between a program and a library. A trace has an event for each call from the program to the library, for each callback from the library to the program, and for each return from a call or callback. On the other hand, traces do *not* have events for calls and returns within the library or within the program.

Calls and returns between library routines are not recorded because they should not appear in specifications. A specification dictates what a program should or should not do. Since the author of the program might not have any knowledge of or control over the library's implementation, a specification that included calls and returns between library routines is likely to be incomprehensible and difficult to obey.

Calls and returns between program routines are not recorded because Strauss aims to mine specifications, which apply to *any* program that uses an interface. Mining from a trace that includes calls and returns between a particular program's

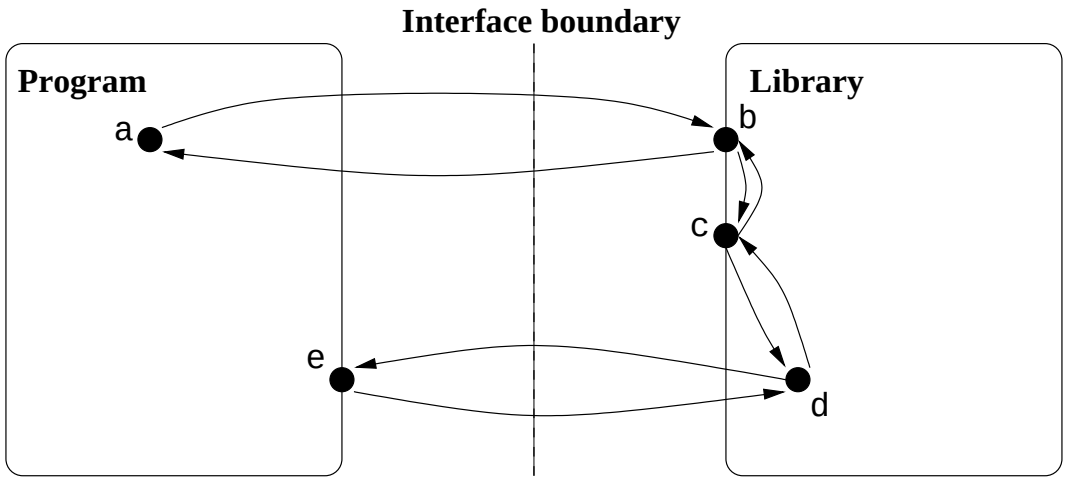

Figure 52: A program, a library, and a sequence of calls and returns within and between the two. Lettered dots indicate routines: `b` and `c` are interface routines, and `e` is a callback routine.

routines risks finding a specification that applies only to that program.

Figure 52 shows an example. The figure depicts a program, a library, and a sequence of calls and returns within and between the two. Lettered dots represent routines: `b` and `c` are interface routines, and `e` is a callback routine. The sequence of calls and returns is

```
call b; call c; call d; call e;

return e; return d; return c; return b
```

The sequence in Figure 52 generates a trace event each time it crosses the dashed line. Note that *no* trace event is recorded for the call from `b` to `c`, because `b` and `c` are both library routines.

Because Strauss traces only record calls and returns that cross an interface, a tracer must know at run-time whether each call or return crosses the interface or

not. Section 6.2 explains how my X11 tracer solves this problem.

6.1.2 Which values to trace

In general, programs and libraries may pass arbitrary amounts of data across an interface. Section 2.3.2 (page 30) gave a flexible syntax for trace events, which can represent arbitrary amounts of structured data. In practice, however, traces must limit the number of values they record. That is, traces must record an abstraction of the values that occur at run-time. This abstraction will necessarily limit which specifications can be mined, because a post-mortem miner like Strauss cannot recreate values that were never recorded.

The following simple abstraction works well for many specifications, although not for all. Suppose that events are represented according to the abstract syntax of Figure 11 (Section 2.3.2, page 29): each event has an attached list of arguments, where each argument has an access path and a value. Recall that the access path bundles a path to the argument into a single identifer of the form

Id0 . Id1 . ... . Idn

Id0 identifies the type of the event (`call`, `callback`, or `return`), Id1 is the name of the event, Id2 is the name of the top-level argument, Id3 is the name of an argument contained within the top-level argument, and so on. The length of an access path is the number of identifiers in it, $n + 1$ in this case.

The *nk-local abstraction* drops all arguments whose access paths are longer than k, and allows no argument list to contain more than n arguments. This abstraction is too coarse to mine *every* specification that one might want to mine

for *every* interface, but it works for many useful specifications that refer only to arguments that are not deeply nested.

The nk-local abstraction does not necessarily rule out mining specifications about unbounded data structures. If the routines that construct, mutate, and visit these structures appear in the trace and do not hide deep structure traversals, it may be possible to recover information about the shape of the data structure. For example, consider the `cons`, `car`, and `cdr` interface in Lisp. In principle, a miner could discover that the `car` of a cons cell equals the first argument of the `cons` that produced the cell, even if the trace event for the `car` does not record every value reachable from the cons cell.

Section 6.2.2 explains how my tracer for the X11 interface applies the nk-local abstraction, with some minor pragmatic modifications.

6.2 Tracing X11

X11 is a portable windowing system for computer networks. In X11, programs run on one machine and communicate over the network with a display server that interacts with the program's user: the server displays windows and graphics, accepts keyboard and mouse input from the user, and relays communication between programs. X11 is a complicated system and the display server only provides basic services, so very few X11 programs are written from scratch. Instead, programs use libraries that hide complexity and build high-level services on top of the server's basic services.

X11 is an excellent testbed for a specification miner, like Strauss, that mines

from traces. First and most importantly, the X11 interface is in wide use: it is not a toy, and it is easy to find programs that use the interface. Also, the X11 interface is complex and poorly documented, so there is a need for specifications. Analyzing the X11 libraries statically would likely be hard, because they use unsafe features of C and because the effects of many library routines are only visible in the display server, not in the state maintained by the libraries themselves. Finally, analyzing X11 programs statically may also be hard, because many of them use third-party libraries like Motif and Gtk, which hide calls to the underlying X11 interface behind a layer of abstraction.

I have written a tracer that records a program's interaction with the standard X11 libraries (Xlib, the X Toolkit Intrinsics, and several standard extensions). The tracer is a Solaris shared library; to trace a program, you tell the dynamic linker to load the tracer before loading the program by setting the `LD_PRELOAD` environment variable. For example, this command would run `xterm` and record a trace of its interaction with the X11 libraries:

```
LD_PRELOAD=X11_tracer.so xterm
```

The tracer does not analyze the program or the X11 libraries. Programs must be linked dynamically, but no source code is required.

The tracer works by redirecting all calls to traced routines to *wrappers*. Each wrapper prints a call or callback event (if appropriate), calls the traced routine, and prints a return event after the traced routine returns (if a call or callback event was printed). Figure 53 shows high-level pseudocode for a wrapper for the (mythical) routine `Foo`.

```
Routine Foo(args)
  If should print a call (or callback) event Then
    Print a call (or callback) event
  rc := call the real Foo with args
  If printed a call (or callback) event Then
    Print a return event
  Return rc
```

Figure 53: High-level pseudocode for a wrapper for `Foo`.

Wrappers print arguments by calling *printers*. I wrote or generated a printer for each data type used by the X11 interface. That was easy; the bigger challenge was determining which arguments should be printed, because routine prototypes in C do not declare input and output parameters.

Because the X11 interface is so large, I wrote Perl [52] scripts to generate most of the code for the tracer. Some code must be written by hand, and parts of my solutions only apply to X11.

The rest of this section describes how my tracer implements the principles presented in Sections 6.1 and 6.1.2. Section 6.2.1 explains how the tracer uses wrappers to determine which events to print, and Section 6.2.2 explains how printers print arguments.

6.2.1 Which calls and returns to trace (X11)

My tracer breaks down the problem of determining which calls and returns to trace into three steps:

- Identifying the routines that should be wrapped. This step also identifies the type of the routine, because the routine's type determines which arguments

should be printed.

- Redirecting calls to interface and callback routines to wrapper routines.
- Writing wrappers that determine if a particular call or return crosses the interface and so should generate an event.

Which routines should be wrapped

The tracer generates a wrapper for each routine in the X11 libraries that could be called by the program and for each routine in the program that could be called by the libraries.

Identifying such routines in the libraries was easy, because these routines are well-defined by header files. I collected the routine prototypes in a single file and wrote a Perl script to parse them and generate a wrapper for each routine.

Identifying routines in the program that could be called by the libraries was harder. The tracer needs to trace each callback routine that can be called from the libraries, and this set of routines varies from program to program. Also, to print arguments, the tracer must know the type of each callback routine. Finally, I did not want the tracer to require source code for the program.

My solution depends on three properties of the X11 interface:

- The X11 header files specify all allowable types of callback routines.
- The X11 libraries always call callback routines through pointers, which are passed to the libraries by the program.
- The program implicitly identifies the type of the routine when it passes the

```
Routine Foo(args)
  If should print a call event Then
    Print a call event
  Search args for pointers to callbacks
  rc := call the real Foo with args
  If printed a call event Then
    Print a return event
  Return rc
```

Figure 54: High-level pseudocode for a wrapper for `Foo`, with the search for callback routines.

pointer to the libraries. After all, if it did not, then the X11 libraries would not know which arguments are expected by the callback routine.

The first observation means that I could create a list of callback types by scanning the X11 header files. Given this list, a Perl script generates, for each type of callback, a number of generic wrappers that are allocated on demand to callback routines.

The last two observations allow the tracer to identify callback routines without analyzing the program's code. Figure 54 lists an extended wrapper for library routines. The extended wrapper, in addition to printing events and calling the library routine, also searches its input parameters (including structure fields) for pointers to callback routines.

Searching a data structure for function pointers is very similar to printing the contents of the structure. To create the searches, I wrote a Perl script, which generates a search for each data type used by the X11 interface.

The type of a callback usually follows from the type of the parameter or fields where it was found, although sometimes it is necessary to inspect values.

As implemented in the tracer, the search is specific to X11. The problem is that X11 often casts pointers to C structures into pointers to smaller C structures. To search through such pointers, the tracer must determine the "real" type of the pointer. The rules for doing this are specific to X11, so the search through such pointers is necessarily also specific to X11.

Identifying callback routines would be easier if program source were available. A conservative solution would take the list of callback types and simply wrap every program routine with a matching type. A better solution would wrap only routines whose address may escape to the libraries. Conservative solutions are acceptable because callback wrappers only generate events when they are called by the libraries.

Redirecting calls

This section explains how the tracer intercepts calls to traced routines and redirects them to wrappers, and how wrappers call traced routines. The tracer has one pair of solutions for these problems for library routines, and another for program routines.

For library routines, the tracer relies on the dynamic linker to redirect calls to wrappers. Wrappers for library routines are given the same name as the routine that they wrap. Because the dynamic linker loads the tracer first, it resolves dynamic calls intended for library routines to wrappers. Thus, dynamic calls to library routines are intercepted. Static calls are not intercepted, but intercepting them is unnecessary, because they can only come from within the X11 libraries.

Wrappers call traced routines as follows. The tracer loads the code for the X11

```
Routine Foo(args)
  real_Foo := find_symbol(``Foo'')
  If should print a call event Then
    Print a call event
  Search args for pointers to callbacks
  rc := (*real_Foo)(args)
  If printed a call event Then
    Print a return event
  Return rc
```

Figure 55: A wrapper for `Foo`, with lookup of `Foo`.

libraries at startup, using the Solaris system call `dlopen`. During execution, to find a library routine, a wrapper calls `find_symbol` with the name of the routine. `find_symbol` uses the Solaris system call `dlsym` to search the loaded libraries for the named routine and caches the result for subsequent lookups. Figure 55 lists pseudocode for the wrapper for a library routine named `Foo`.

Redirecting calls to program routines to their wrappers is harder because program routines are not identified until run-time. Say that `CB` is a program routine, of type `CBType`. The tracer has a number of wrappers for callbacks of type `CBType`, so that each callback (up to a compile-time limit) can be assigned its own wrapper. When the program first passes `CB`'s address to the library, the tracer performs four steps:

1. Replace the instruction at `CB`'s entry point with a trap instruction.

2. Create a trampoline `tramp_CB` that jumps to the instruction just after the trap instruction, with `CB`'s original entry-point instruction in the delay slot of the jump.

```
Routine wrap_CBType_I(args)
  Return wrap_CBType_generic(I, args)

Routine wrap_CBType_generic(I, args)
  tramp_CB := lookup trampoline for wrap_CBType_I
  If should print a callback event Then
    Print a callback event
  rc := (*tramp_CB)(args)
  If printed a callback event Then
    Print a return event
  Return rc
```

Figure 56: A wrapper for a callback of type `CBType`.

3. Select an unused wrapper `wrap_CBType_I` from the wrappers for callbacks of type `CBType`.

4. Associate `CB` with `wrap_CBType_I` and `tramp_CB` by adding an entry to a table.

Whenever the trap instruction is executed, the operating system sends the process a signal. The tracer catches the signal and arranges to have execution resume at the entry point of `wrap_CBType_I`. Figure 56 lists pseudocode for `wrap_CBType_I`. To save space in the text segment, most of the wrapper's work is done by a generic helper routine, `wrap_CBType_generic`.

To call the traced routine, the wrapper finds `tramp_CB` in the table and executes the original code through the trampoline.

```
Routine Foo(args)
  real_Foo := find_symbol(``Foo'')
  caller := get_caller()
  If not is_in_library(caller) Then
    Print a call event
  Search args for pointers to callbacks
  rc := (*real_Foo)(args)
  If not is_in_library(caller) Then
    Print a return event
  Return rc
```

Figure 57: Wrapper for `Foo` that prints an event when the interface boundary is crossed.

When to print events

Section 6.1 explained that tracers should only print events when the boundary between library code and program code is crossed. There are two ways to cross the boundary: a call to a library routine crosses the boundary iff the caller is in the program, and a call to a callback routine crosses the boundary iff the caller is in the library. So, to determine which calls and returns cross the boundary, the X11 tracer just needs a way to tell whether or not a wrapper's caller is in the library.

The Solaris system call `dladdr` looks up the shared library that owns a given address. With `dladdr` and the names of the X11 libraries, it is easy to write a routine `is_in_library` that detects whether an address is in the library or not. It is also easy to write a macro that gets the caller of a routine:

```
#define get_caller(v) asm("mov %%i7, %0" : "=r" (v));
```

Here `asm` is a `gcc` directive. This macro works on SPARC machines running Solaris.

Figure 57 lists pseudocode for a wrapper for a library routine, which prints events only when the interface boundary is crossed; the code for callback wrappers is similar.

6.2.2 Printing values

This section explains how the tracer prints arguments, applying the *nk*-local abstraction, with some minor pragmatic modifications.

There are two subproblems: deciding which values to print, and figuring out how to print each value. The former was by far the larger challenge.

Call and callback events should print the input values that the caller passes to the callee; return events should print the output values that the callee returns back to the caller.

Unfortunately, in C, parameters are passed by value, and other calling conventions are simulated with pointers. Because pointers are used for so many reasons in C, there is no general way to tell from looking at a routine's prototype which values are inputs and which are outputs.

I solved this problem by adding my own annotations to the prototypes of all library routines. Often, the proper annotations were indicated by the name of the argument, because C programmers realize that C prototypes are incomplete. In difficult cases, I consulted the X11 manual pages.

When my Perl scripts generate the code to print an event, they use the annotations to print the proper values for call, callback, and return events.

An example[1] is

[1]In my implementation, annotations are encoded in argument names, but for clarity I write

```
extern int XGetDeviceFocus(
  Display* display,
  XDevice* device ,
  Window* focus /*@ return @*/,
  int* revert_to /*@ return @*/,
  Time* time /*@ return @*/
);
```

The wrapper for `XGetDeviceFocus` prints the values of the `display` and `device` arguments in the call event and the values of `*focus`, `*revert_to`, and `*time` in the return event.

Some return values are only conditionally valid:

```
extern Bool XmbufQueryExtension(
  Display* dpy,
  int* event_base /*@ checked_return @*/,
  int* error_base /*@ checked_return @*/
);
```

Here, the return event includes the values of `*event_base` and `*error_base` iff the return value of `XmbufQueryExtension` matches some condition, in this case iff the return value is `true`. My implementation of the `checked_return` keyword hard-codes the condition based on the routine's return type; I have not felt the need for more general conditions.

With the help of these annotations, the tracer can decide which arguments

them as comments here.

to print. To tell the tracer how to print each argument, I wrote or generated a printer for each type in the X11 interface. Wrappers print arguments by calling these printers.

For the basic types used by the X11 interface, I wrote printers by hand.

For structures, I wrote a Perl script to generate the printers. To bound the number of values that are printed, these printers enforce a modified form of the nk-abstraction. Instead of a single parameter k, the printers use two parameters k_s and k_p. The tracer will not print values that nest more deeply than k_s within a C structure or are reached by more than k_p pointers.

The default value of k_s is 10 and the default value of k_p is 1, but the user can change these value by setting environment variables. I determined these numbers empirically, and found that they strike a good balance between keeping trace sizes manageable and maintaining the precision I needed to mine good specifications.

Finally, I should mention how I dealt with two minor annoyances:

- The X11 interface uses pointer casting to simulate polymorphism; the true type of a pointer can be discovered, but only in an ad hoc fashion. I generated special-purpose code to print the contents of such pointers, just as I did to search the contents for pointers to callback routines.

- Printing arrays properly required more annotations. In C, the size of an array is not kept with the array. Luckily, for every array used by the X11 interface, there is some way to tell how big it is: usually, there is another parameter that holds the size. An example is

```
extern XFeedbackState* XGetFeedbackControl(
```

```
  Display* display,
  XDevice* device,
  int* num_feedbacks /* return, counts(return) */
);
```

Here, the return event includes the array returned by `XGetFeedbackControl` and `*num_feedbacks` is assumed to hold the size of the array. Note that the tracer will not print more than n elements of an array, as dictated by the nk-abstration. The user can change this value by setting an environment variable.

6.3 The traces

Table 8 lists the traces that I used in the experiments in this dissertation.

Table 8: The traces that I used in the experiments in this dissertation. **Program** lists the name of the executable, **Source** lists the distribution from which the executable came, **Events** lists the number of events in the trace, **Size Gzip** lists the size of the trace (in kilobytes) when compressed with Gzip, and **Size Full** lists the uncompressed size of the trace (in kilobytes).

Program	**Source**	**Events**	**Size (KB)**	
			Gzip	**Full**
airport	motif-2.1.10	75424	438	7366
animate	ImageMagick-5.1.1	108534	490	14200
animate	motif-2.1.10	220635	582	14819
autopopups	motif-2.1.10	77176	420	5943
bitmap	XFree86-4.0.3	13996	151	2723
bugsx	bugsx108	57498	181	4811
calculator	fox-0.99.189	367192	839	29599
clipboard	Clipboard-2.4	474	26	273
coolinput	cooledit-3.17.5	10222	46	1115
coollistbox	cooledit-3.17.5	55836	123	4549
coolman	cooledit-3.17.5	48224	152	4231
coolmessage	cooledit-3.17.5	4296	27	565
coolquery	cooledit-3.17.5	2710	23	460
dainput	motif-2.1.10	46548	263	4775
display	ImageMagick-5.1.1	6408	64	947

(continued)

Program	Source	Events	Size (KB)	
			Gzip	Full
display	ImageMagick-5.1.1	520066	2959	67402
DNDDemo	motif-2.1.10	38162	229	3083
dogs	motif-2.1.10	90285	437	7148
draw	motif-2.1.10	186940	975	15758
e93	e93-1.3r2X	1330984	3442	102513
earth	motif-2.1.10	24880	120	2124
editres	XFree86-4.0.3	24326	282	4314
emacs-20.7	emacs-20.7	270662	1052	31085
emu	emu-1.31	22934	118	2263
exm_in_c	motif-2.1.10	192256	1042	14855
exm_in_uil	motif-2.1.10	161009	1054	16174
filemanager	motif-2.1.10	655160	2317	40696
getsubres	motif-2.1.10	48026	283	4039
gnuplot_x11	gnuplot-3.7.1	2310	21	293
hellomotif	motif-2.1.10	9298	72	891
i18ninput	motif-2.1.10	376448	1865	30769
import	ImageMagick-5.1.1	544	12	124
kterm	kterm-6.2.0	26079	194	3347
mMosaic	mMosaic-3.0.11	1797566	8740	148259
motifshell	motif-2.1.10	257788	913	16877

(continued)

Program	Source	Events	Size (KB)	
			Gzip	Full
nedit	nedit-5.0.2	3436480	15761	272682
panner	motif-2.1.10	128806	780	11295
piano	motif-2.1.10	292386	1264	21305
pixmap	pixmap2.6	289827	912	24420
rtc	rtc-2.0a	481066	2522	34725
rxvt	rxvt-2.16	10850	42	1310
sampler2_0	motif-2.1.10	611666	3188	49519
setDate	motif-2.1.10	69396	423	6019
setDate	motif-2.1.10	92	8	73
simpledrop	motif-2.1.10	81102	397	5986
smalledit	cooledit-3.17.5	447340	1432	39041
todo	motif-2.1.10	448226	2480	34505
ups	ups-3.36	74544	221	7437
winterp	winterp-2.10	53391	332	4413
winterp	winterp-2.10	104719	555	8015
winterp	winterp-2.10	20041	144	1798
winterp	winterp-2.10	37057	232	3112
winterp	winterp-2.10	48723	286	4225
winterp	winterp-2.10	269503	875	17277
winterp	winterp-2.10	36825	202	3201

(continued)

Program	Source	Events	Size (KB)	
			Gzip	Full
winterp	winterp-2.10	41635	213	3955
winterp	winterp-2.10	48531	225	4046
winterp	winterp-2.10	583003	2730	45392
winterp	winterp-2.10	21285	168	2119
winterp	winterp-2.10	25431	177	2338
winterp	winterp-2.10	109263	651	9147
winterp	winterp-2.10	265771	1271	20424
wish8.3	tk-8.3.4	46678	186	4302
wish8.3	tk-8.3.4	42532	149	3910
wmagnify	wmaker-0.70.0	40644	523	14241
xart	xart-19980417	219722	1592	37051
xcalc	XFree86-4.0.3	722	28	344
xcb	xcb-2.3	1344	21	314
xclipboard	XFree86-4.0.3	278	24	239
xconsole	XFree86-4.0.3	188	13	123
xcoral	xcoral-3.2	246588	1028	27185
xcpustate	xcpustate-2.5	2080	24	357
xcutsel	XFree86-4.0.3	378	15	160
xdtm	xdtm-2.5.8	12110	209	2805
xfontsel	XFree86-4.0.3	30174	315	4370

(continued)

Program	Source	Events	Size (KB)	
			Gzip	Full
xhtml	xhtml-1.3	1505364	7402	117202
xinout	xinout-1.1	160081	804	12747
xip	xip-1.1b	628	28	265
xmag	XFree86-4.0.3	163760	214	12637
xmforc	motif-2.1.10	308956	1467	23624
xmform	motif-2.1.10	53534	252	4131
xmotd	xmotd-1.16	156	17	145
xpaint	xpaint-2.4.7	321832	1642	34657
xpaste	xpaste-1.1	135	14	109
xpdf	xpdf-0.91	135776	444	12792
xpilot	xpilot-4.3.2	1093354	10946	154978
xp-mapedit	xpilot-4.3.2	263402	860	27187
xsession	xsession-1.1	1390	36	471
xterm	xterm-R6-sb_right-ansi-3d	7180	48	974
xterm	color_xterm	61798	211	4897

Chapter 7

Debugging Specifications

This dissertation is about methods for addressing the specification problem—that is, 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).

Not surprisingly, 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 an SA that accepts erroneous interaction traces. Clearly, a solution to the specification problem also requires methods for debugging specifications.

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 (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 the trace 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.

These 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, 27, 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 individually inspecting every trace. I have used the method to debug Strauss specifications, but in fact the method applies to any temporal specification.

[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.

In my method, an automatic tool finds similarities within a set of traces and uses *concept analysis* [55] to cluster similar traces together. The user inspects clusters of traces—summarized in various ways—instead of individual traces. Ideally, 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 to classify the cluster's traces as 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 (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.

My method defines the similarity of a set of traces in terms of the transitions of an 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; it is also intuitive, because the user is debugging a specification that is itself expressed in terms of an FA. The definition also enables my use of 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 61 bugs in widely distributed X11 programs, including serious race conditions and performance bugs. I 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. I also found that concept analysis is affordable: it never took longer than about 22 seconds to compute the concept lattice.

Contributions

This chapter describes the following 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 classifying traces by exploring a cluster hierarchy.

Acknowledgements

Some of the work described in this chapter was done by another student, David Mandelin. Specifically, David invented and implemented navigation by transitions (see Section 7.3) and performed the experiments described in Section 7.4.3.

Organization of this chapter

The rest of the chapter is organized as follows. Section 7.1 presents 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 7.5 concludes the chapter.

For all trace events matching

```
X = popen() | X = 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

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

Figure 58: An incorrect temporal specification.

7.1 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 Strauss specification by inspecting the scenarios from which Strauss inferred the specification.

I will refer to several FAs in this section and in the rest of this chapter. Note that, in this chapter, the start state of an FA is always state 0, and double lines indicate an accepting state.

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

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

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

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

7.1.1 Debugging by testing

Figure 58 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 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 hundreds or thousands of complicated violations (as some do [27, 6]), 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

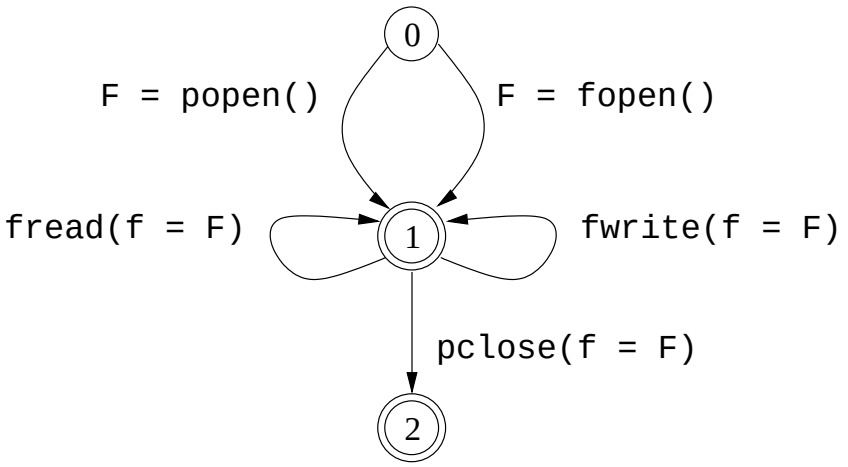

Figure 60: A small FA that recognizes violation traces from verification of the specification in Figure 58.

summarizes the violation traces, before the specification author sees them. This step has three substeps:

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 [37] for a good survey. However, an FA learning algorithm that performs well on traditional measures, such as training set accuracy, is not needed for concept analysis. I only require that dissimilar traces execute different transitions in the automaton (see Step 1b). For example, I have had success with FAs that recognize *all* possible traces over the API.

Figure 60 shows a small FA that recognizes violation traces from verification of the specification in Figure 58. Figure 61 shows an automaton that recognizes all traces over the API.

Step 1b This step uses a reference FA M to define a measure of similarity for

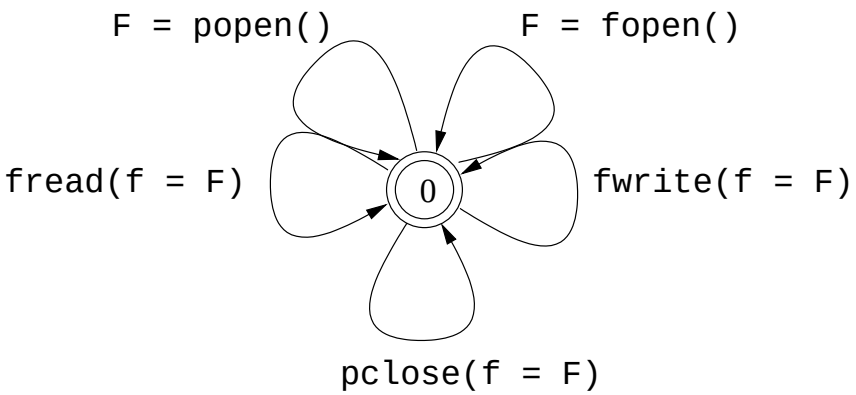

Figure 61: A very small FA that recognizes violation traces from verification of the specification in Figure 58.

violation traces. M recognizes a trace o iff there is an *accepting sequence of M-transitions 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 an M-transition a is on an accepting sequence of M-transitions for o, we say that o *executes* a. Given a set O of violation traces, the *common M-transitions of O* are the M-transitions that are executed by every violation trace in O. The *similarity of O with respect to M* is the number of common M-transitions of O.

Note that we want a reference FA that is useful for classification. In particular, erroneous traces and correct traces should execute different transitions, so that they are not considered highly similar. It is also helpful, but not necessary, if correct (erroneous) traces execute many of the same transitions as other correct (erroneous) traces, so that they are considered highly similar.

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 60 distinguishes between traces that call `popen` before calling `pclose` and traces that call `pclose` before calling `popen`, since the latter execute no transitions in the FA. If the order did not matter, a very small FA, such as the one given 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 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

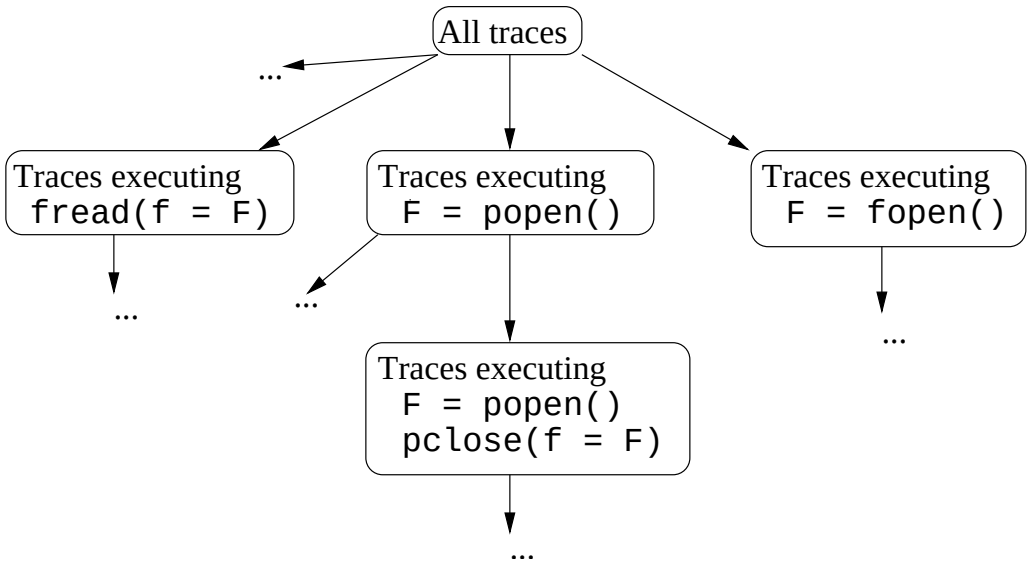

Figure 62: Part of a concept lattice that might be induced by violation traces from verification of the specification in Figure 58, with respect to the FA in Figure 60.

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 suggest several strategies, which are evaluated in Section 7.4.

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".

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:

For all trace events matching

```
X = popen() | X = 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

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

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

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 find an FA for the scenarios. He already has one: namely, the FA in the scenario acceptor from Strauss's buggy

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

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

Figure 64: Several scenarios.

specification. On the other hand, if Strauss infers an FA that makes unnecessarily fine distinctions among scenarios, the expert may choose to use a different FA. In my experience, however, the inferred FA is usually a good starting point.

Steps 1b and 1c are just as in Section 7.1.1: the expert supplies a reference FA, which defines a measure of similarity for scenarios and 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 in the training set. For example, Strauss, given the "good" scenarios in Figure 64, would ideally produce an FA 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 FA 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. In the example, if the expert observes that Strauss overgeneralizes, he would redo Step 2 and assign several different kinds of "good" labels. Here, the expert would assign a label "good_fopen" and another label "good_popen". Next, the expert would run the Strauss's learner twice, once on the "good_fopen" scenarios and once on the "good_popen" scenarios. Because the learner sees each class of scenarios separately, it can not confuse them. The final specification would be the union of the specification for "good_fopen" scenarios with the specification for "good_popen" scenarios.

7.2 Applying concept analysis

Concept analysis [55] 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. 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.

	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.

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. 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

[2] I took this example from Michael Siff's thesis [46].

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})

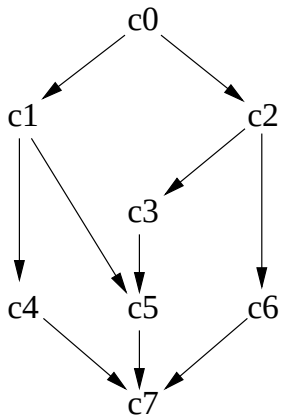

Figure 66: Concept lattice for Figure 65. The top concept is c0, and the bottom concept is c7.

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 lattice*. Figure 66 shows the concept lattice for the example in Figure 65. In general, the *top concept* of a concept lattice is the concept with all objects and the *bottom concept* is the concept with all attributes. In the example, the top concept is c0, and the bottom concept is c7.

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 $\text{sim}(X) = |\sigma_R(X)|$. 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 $\text{sim}(X_0) \geq \text{sim}(X_1)$.

Efficiency of concept analysis

There are several algorithms for building concept lattices. The algorithm I use is due to Godin and others [24] (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, which contained 496 traces.

7.2.2 Clustering traces

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 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 M-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 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?".

My definition of similarity with respect to an FA ignores the order in which transitions are executed. 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, by changing the FA, my definition of similarity can simulate definitions that track the order of transitions. The FA already constrains the order in which transitions may execute. Thus, by distinguishing traces that execute different sets of transitions, the FA also makes some distinctions among traces that execute transitions in different orders. If more ordering information is desired, the FA can be modified to make finer distinctions (see the discussion in Section 7.1.1, Step 1b).

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 [21] and Grappa [36] graph visualization tools, 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 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 [43]) to construct a summary FA that accepts concept traces and then displays this FA. In my experience with Cable (and David Mandelin's 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.

Transitions This view displays the transitions in the reference FA that belong to the concept. In my experience, this has been the second most frequently used view 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 view.

Traces This view displays the traces in the concept. We do not use this view 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 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 a reference 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 FA. If this cluster appeared complicated, I sometimes started a focused session. The reference FAs that I used for focusing followed one of the following three templates:

Unordered FAs 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 FA. FAs that follow the unordered template work well when correct scenarios and erroneous scenarios often contain different events.

Name projection FAs that follow this template distinguish scenarios 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 scenarios) whose transitions are all labeled by *wildcard* or by an event that refers to X.

Seed order FAs 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 FA. Because FAs that follow the seed order template pay attention to ordering, they can distinguish scenarios that cannot be distinguished by an unordered FA. 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?". 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 otherwise an optimal strategy could include redundant 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 visits Unlabeled and PartiallyLabeled concepts in a random order, subject to the constraints that no concept may be visited unless one its parents has already been visited and that no concept may be visited if carries an "inspected" mark. At each visit, the strategy marks the concept as "inspected" and inspects the concept's unlabeled traces. If all unlabeled traces should receive the same label, then the strategy labels them. Labeling a concept can make it possible to label the concept's ancestors, because labeling the concept changes the sets of unlabeled traces in the ancestors. For this reason, when a concept is labeled, the strategy clears the marks on the concept's ancestors..

The advantage of this strategy is that, because it is biased toward visiting concepts near the top of the lattice, it is likely to exploit opportunities to label many traces at once. 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.

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 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 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 61 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 method was better than this brute-force method. Nonetheless, the results in Section 7.4.2 show that my 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. Since Strauss extracted many identical scenario traces, I built the lattice from representatives for classes of identical traces, rather than from all of the traces.

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

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	8	10	13	21	37	5.5
RegionsAlloc	10	14	15	18	24	4.2
RegionsBig	270	375	611	680	1377	2.67×10^3
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	2	5	5	4	4	1.2
XInternAtom	10	3	11	13	21	4.1
XPutImage	6	3	7	10	14	2.6
XSaveContext	92	150	224	302	639	477
XSetFont	28	30	40	66	129	25.0
XSetSelOwner	6	3	7	9	13	2.3
XtFree	112	95	171	380	869	1.51×10^3
XtOwnSel	7	3	8	10	15	2.7
XtRealizeProc	38	57	64	104	207	37.4

Table 9: 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 reference 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.

empirical evaluation that Godin and others did of their algorithm (which I use) in their paper [24].

7.4.2 Traversal strategies

Tables 10 and 11 compare 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 11 reports the lowest cost for Bottom-up and

Spec.	Cost of labeling	
	Exp	Base
PrsAccelTbl	8	18
PrsTransTbl	2	6
Quarks	2	16
RegionsAlloc	16	20
RegionsBig	149	540
RmvTimeOut	2	4
XFreeGC	12	20
XGContextFrom	24	50
XGetSelOwner	5	4
XInternAtom	5	20
XPutImage	10	12
XSaveContext	42	184
XSetFont	53	56
XSetSelOwner	5	12
XtFree	28	224
XtOwnSel	5	14
XtRealizeProc	13	76

Table 10: The cost of manual labeling, with and without Cable. The cost of Expert (**Exp**) is compared to the cost of Baseline (**Base**), which does not use Cable.

the arithmetic mean and standard deviation of the cost of 1024 trials for Top-down and Random. I was unable to measure the cost of the Optimal strategy for RegionsBig and XSaveContext, because the program I wrote to evaluate the strategies on these specifications took too long to run. For those specifications, I report a lower bound on the cost of Optimal.

Table 10 lists two manual labeling 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. On 5 specifications (RegionsBig, XSaveContext, XGContextFromGC, XtFree, and XtRealizeProc), the expert

Spec.	Cost of labeling					
	Opt	Top-down		Btm-up	Rand	
		Mean	Std dev		Mean	Std dev
PrsAccelTbl	4	14.0	1.9	16	14.1	2.8
PrsTransTbl	2	2.0	0.0	4	3.4	0.9
Quarks	2	2.0	0.0	16	8.6	3.2
RegionsAlloc	8	15.7	1.4	20	16.2	2.8
RegionsBig	≥ 7	121	9.2	540	231	29.2
RmvTimeOut	2	2.0	0.0	4	3.4	0.9
XFreeGC	6	12.9	2.3	20	14.4	3.4
XGContextFromGC	14	39.0	4.3	50	39.5	5.9
XGetSelOwner	4	5.0	0.0	4	4.5	0.9
XInternAtom	4	5.0	0.0	20	12.5	5.6
XPutImage	6	21.0	3.9	12	15.6	3.9
XSaveContext	≥ 5	267	21.8	184	188	18.9
XSetFont	16	52.0	5.4	56	50.3	4.8
XSetSelOwner	4	5.0	0.0	12	8.7	3.4
XtFree	24	124	10.9	224	149	13.4
XtOwnSel	4	5.0	0.0	14	9.8	3.9
XtRealizeProc	12	29.1	2.1	76	51.0	7.4

Table 11: The cost of labeling with four automatic Cable strategies: Optimal (**Optimal**), Top-down (**Top-down**), Bottom-up (**Btm-up**), and Random (**Rand**).

also used Cable's "Focus" command, using reference FAs that matched the templates described in Section 7.3.1. This was an advantage for the expert, since the automatic strategies did not use this feature of Cable.

Baseline As mentioned above, many of the traces were identical. Instead of using Cable, this method simply divides the traces into classes of identical traces 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 traces.

Comparing the cost of Expert with the cost of Baseline indicates the value of

Cable in practice. By this measure, the advantage of using Cable increases as the number of different scenario 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 was a bit higher, while the cost of Expert remained very low. Cable shows an advantage here. Finally, for RegionsAlloc, XFreeGC, and XPutImage, the cost of both methods was a bit higher, although the cost of Baseline was still slightly higher than the cost of Expert.

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

  XSaveContext is a special case. This specification's FA was large: 150 states and 224 transitions. In fact, I discovered while debugging this specification that most of these states and transitions are irrelevant. These irrelevant transitions hurt the performance of Top-down and Random, because concept analysis found many spurious concepts. On the other hand, these transitions did not hurt Expert very much, because the expert used the "Focus" command to choose a better reference FA.

- Top-down outperforms Random, which implies that Top-down was often able to label concepts near the top of the lattice. Top-down beat Random on all but four specifications (XGetSelOwner, XPutImage, XSaveContext, and XSetFont). Top-down beats Random handily when no traces are erroneous (PrsTransTbl, Quarks, and RmvTimeOut). This is not surprising, since the top concept can be labeled immediately in this case. Top-down also beats Random significantly on more interesting specifications, particularly XtRealizeProc and RegionsBig. By contrast, Random beat Top-down significantly on just one specification: XSaveContext, which contained many irrelevant

transitions.

- 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.

7.4.3 Navigation by transitions

In another set of experiments, David Mandelin and I (together with our advisors, of course) studied how long it took him to debug specifications using Cable. For 11 of the 17 specifications, David measured the actual time elapsed during a Cable debugging session. He then compared the session's time to the time that he, as an expert, needed to debug the specification without Cable, by examining and classifying individual scenario traces.

This experiment used Cable's *navigation by transitions* feature, which David invented and implemented (see Section 7.3.1). The idea was to replace a traversal strategy (see Section 7.3.2) with navigation to concepts that contain only correct or incorrect traces. The expert's strategy was as follows. Starting at the top concept, the expert examined the summary FA. If the FA described only correct behavior or only erroneous behavior, the expert labeled the concept accordingly. Otherwise, the expert selected transitions that were either clearly executed by all correct traces, or clearly executed only by erroneous traces. Then, the expert asked Cable to find the smallest concept that contained all traces that execute all of the selected transitions and applied this procedure recursively at the new

Name	Time		Cost of labeling	
	Cable	Baseline	Cable	Baseline
PrsAccelTbl	136	42	5	18
PrsTransTbl	9	11	2	6
RmvTimeOut	9	6	2	4
XGContextFromGC	158	107	12	50
XGetSelOwner	32	9	5	4
XInternAtom	100	37	8	20
XPutImage	148	31	14	12
XSetFont	117	133	12	56
XSetSelOwner	27	33	5	12
XtFree	251	254	42	224
XtOwnSel	98	21	10	14

Table 12: Time to debug specifications. **Time** reports the time to debug a specification using either Cable or the baseline method of classifying individual traces. **Cost of labeling** reports the cost of labeling as defined in Section 7.3.2.

concept. After labeling a concept, the expert returned to the concept above it. If this concept still contained unlabeled traces, the expert started over at this concept, but only with the unlabeled traces.

For this experiment, we used a modified set of attributes for concept analysis. As in previous experiments, the attributes of a trace included the transitions executed by the trace in the reference FA. Our modification added "negative" attributes: each trace had a negative attribute for each transition *not* executed by the trace. That is, a trace that executed transition i but did not execute transition j would have attributes a_i and $\bar{a_j}$. The rationale for creating more attributes was to create a lattice with more concepts. Specifically, we wanted to ensure that Cable was always able to navigate to a concept closely matching the transitions selected by the expert. See Section 7.4.4 (below) for a discussion of how to define suitable attributes.

Table 12 shows the time to debug each specification, together with the cost of labeling, as defined in Section 7.3.2. For four specifications (PrsTransTbl, XSetFont, XSetSelOwner, and XtFree), using Cable with the navigation strategy was faster than the baseline method of classifying individual traces, but in general Cable was slower. However, Cable was often significantly better in terms of labeling cost, which reflects the number of concepts that the expert examined.

Comparison with Table 10 reveals three cases where one expert beat the other by a large margin: on XtFree, the expert who used the traversal strategy beat the expert who used navigation (28 to 42); on XGContextFromGC and XSetFont, the expert who used navigation won (12 to 24 and 12 to 53). On XGContextFromGC, the navigation expert even beat Optimal, which was possible because the navigation expert was working with a larger concept lattice. The fact that neither expert dominated the other indicates that both traversal and navigation are valuable, although more study is needed to settle this question.

A natural question is why the sometimes dramatic reductions in labeling cost from Baseline to Cable did not always translate into shorter classification times. In our experience, using Cable with navigation was slower than Baseline in those cases when the summary FA for the top concept was hard to understand. If the summary FA lacked salient features, such as an obviously erroneous loop, the expert had to study all paths through the FA. The understanding of the top summary FA thus became even harder than the baseline method because the FA presented potentially confusing overlapping paths. A possible solution is for the expert to start by traversing the lattice, using navigation by transitions only when he finds a summary FA with a salient feature.

7.4.4 Discussion

I presented and evaluated two approaches to debugging specifications using concept analysis. In *traversal* strategies, the user visits concepts in some order and attempts to label them (Section 7.4.2). In *navigation* strategies, the user selects features of a concept in order to navigate to a concept with those features (Section 7.4.3). These two strategies have complementary benefits but also conflicting requirements.

In a traversal strategy, the user searches the lattice for classifiable concepts. The search is undirected in that the user cannot influence the traversal order by indicating which traces from the current concept he would prefer to visit next. For example, in a top-down strategy, when the user encounters a concept that cannot be classified, he moves to a subconcept that represents a smaller, simpler subset of the traces. The traversal is thus based only on the structure of the lattice, not on user-selected features of concepts.

As a result, the traversal strategy works best on a small lattice because, otherwise, its undirected search often visits many ancestors (or descendants) of the classifiable concepts.

In a navigation strategy, the user skips over many concepts by navigating directly to a subconcept that focuses on an interesting feature of a larger concept. For example, in our experiments, the user selected salient transitions in a summary FA and navigated to a concept containing matching traces.

The navigation strategy works best when concepts have salient features and there is a closely matching concept for any user selection of salient features. Call

such a lattice *navigable*. A simple way to get a navigable lattice is to use a complete subset lattice.

Traversal and navigation have complementary benefits. Navigation is very effective when concepts have salient features. However, in practice, concepts do not always have salient features, in which case traversal is needed. Thus, one would like to combine traversal and navigation into a single strategy with the benefits of both.

Combining traversal and navigation is, however, difficult because they have conflicting requirements. While traversal requires a small lattice, navigation requires a navigable lattice. Therefore, a combined strategy would work best on a small, navigable lattice. As noted, a complete lattice is navigable, but not small. Conversely, while designing our experiments, we observed that small lattices are often not navigable.

One solution to the problems with combining navigation and traversal is to use a single lattice, but with a different view for each approach. The lattice would be complete, or nearly so, to support navigation. However, the user would only see a coarse subset of the lattice, which could be traversed effectively. The research question is how to select the subset of attributes that will define the coarser lattice used in the traversal.

Another solution is to select attributes judiciously to produce a lattice that is both small and navigable. If there were an attribute for each salient feature, the lattice would be navigable, and not too large. Since the salient features are not known when the lattice is computed, one would have to alter the lattice in response to user input. An interesting question is how to infer good attributes from user

input. For example, with navigation by transitions, a tool could add attributes as necessary to distinguish selected transitions. As another example, a tool could infer useful attributes according to labels previously assigned by the user.

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. Also, there appears to be a trade-off between small concept lattices, which are easier to traverse, and large concept lattices, which are more likely to contain a desired concept. Future work should explore this trade-off, probably by changing the lattice interactively.

Chapter 8

Checking Specifications

Specification mining would be much less attractive if program-verification tools could not use the specifications to find errors. In this chapter, I describe a simple verification tool, which uses Strauss specifications to find errors. The tool found 199 serious errors in widely distributed X11 programs, and I discuss those errors in detail in Chapter 9. This chapter focuses on the verification tool itself.

The tool is a *dynamic checker*, which means that it finds errors by analyzing program traces, not by analyzing source code. A dynamic checker cannot find errors that never execute. On the other hand, implementing a dynamic checker was very easy, because Strauss also analyzes program traces. In fact, the dynamic checker reuses the Strauss code that constructs semantic traces from traces (Chapter 3); extracts scenarios from semantic traces (Chapter 4); and reduces a scenario acceptor, which accepts scenarios, to an NFA, which accepts strings (Chapter 5).

The dynamic checker is fast: like specification mining, its time complexity is linear in the size of the trace.

The dynamic checker is also sound. For positive specifications, which dictate that programs must do something, "sound" means that if the checker reports no errors, then the input trace has no errors. For negative specifications, which dictate that programs must not do something, "sound" means that if the checker reports

errors, then the input trace definitely has errors. The checker is not complete: for positive specifications, it may report errors that do not exist; for negative specifications, it may miss errors.

The checker loses completeness in a trade for speed: if completeness were restored, the checker's time complexity would be exponential in the size of the trace.

In the rest of this chapter, I explain how the dynamic checker checks positive and negative specifications (Section 8.1) and the tradeoffs and drawbacks of dynamic checking in general—and this dynamic checker in particular (Section 8.2).

8.1 Dynamic checking algorithms

8.1.1 Checking positive specifications

A positive specification dictates that programs must do something. As Chapter 9 reports, 16 of the 17 specifications I found with Strauss are positive specifications.

In Section 2.4.1 (page 32), I gave a semantics for positive specifications. In essence, the semantics says that a program obeys a positive specification iff, in every trace of the program, every event that matches the specification's seed pattern has a scenario that is accepted by the specification's scenario acceptor.

Strauss checks that a trace T obeys a positive specification *spec* by searching for such scenarios. To run in linear time (in the size of the trace), the algorithm restricts the search in two ways. First, the search only considers convex scenarios; that is, it only considers scenarios that can be found by `ExtractScenario` (Section 4.2.2, page 77). Second, the search ignores scenarios whose size exceeds a constant, $RMAX$. This constant is empirically chosen; for my experiments, I used 10.

Restricting the search in these ways can cause the checker to generate spurious errors. This happens when a seed event has scenarios that satisfy the specification's scenario acceptor, but all of these scenarios are not convex or too large. Section 8.2 discusses an example where the checker generates a spurious error.

Figure 67 lists pseudocode for checking that a trace T obeys a positive specification *spec*. The algorithm reuses algorithms that I have discussed before, as parts of mining:

```
RMAX is a small, nonnegative integer

Routine CheckPositive(T, spec)
  P := MakeSemanticTrace(T, stm(spec))
  D_flow := InferFlowDependences(P)
  Foreach e ∈ events(P)
    If e matches pattern(spec)
      If NoMatchingScenario(e, P, D_flow, spec)
        Report an error at e

Routine NoMatchingScenario(e, P, D_flow, spec)
  r_total := 0
  While r_total <= RMAX
    r_f := 0
    While r_f <= r_total
      S := ExtractScenario((e, r_total − r_f, r_f), P, D_flow)
      If acceptor(spec) accepts S
        Return false
      r_f := r_f + 1
    r_total := r_total + 1
  Return true
```

Figure 67: Pseudocode for checking that a trace T obeys a positive specification *spec*.

1. `CheckPositive` constructs a semantic trace by calling `MakeSemanticTrace` (Chapter 3, page 51).

2. `CheckPositive` finds flow dependences by calling `InferFlowDependences` (Chapter 4, page 71).

3. For each seed event e, `NoMatchingScenario` searches for a scenario that is seeded by e and accepted by acceptor(*spec*). Each scenario is extracted by calling `ExtractScenario` (Chapter 4, page 77).

The running time of each of these steps is linear in the size of T, so the time

For all trace events matching

```
XSetSelectionOwner(time = X)
```

with STM

```
XSetSelectionOwner(use time, add_to_name selection)
XFilterEvent(event -> (def time))
XtDispatchEvent(event -> (def time))
return XNextEvent(event -> (def time))
return XWindowEvent(event -> (def time))
callback_XtActionProc(widget -> (event -> (def time)))
```

a scenario matches

```
(XFilterEvent(event -> (time = X))
 | XtDispatchEvent(event -> (time = X))
 | XNextEvent returns (event -> (time = X))
 | XWindowEvent returns (event -> (time = X))
 | callback_XtActionProc(widget -> (event -> (time = X))));
XSetSelectionOwner[selection = 1](time = X) [seed]
```

Figure 68: XSetSelOwner. A positive Strauss specification for the informal rule, "The timestamp passed to `XSetSelectionOwner` must come from an event received from the X server."

complexity of `CheckPositive` is also linear in the size of T.

Essentially, the algorithm in Figure 67 tries to construct a proof that T obeys *spec*. If `CheckPositive` generates no errors, then it has constructed a scenario for each seed event that satisfies acceptor(*spec*), and these scenarios constitute a proof that T obeys *spec*. As a theorem prover, `CheckPositive` is sound but not complete, because it may generate errors for traces that do satisfy the specification.

As a simple example of `CheckPositive` in action, consider checking that the following (greatly simplified) trace obeys the Strauss specification in Figure 68:

```
XFilterEvent(event = 0x10 -> (time = 123))
```

```
XSetSelectionOwner(selection = 1, time = 123)
XSetSelectionOwner(selection = 1, time = 0)
```

Both calls of `XSetSelectionOwner` match the seed pattern, so `CheckPositive` tries to find a scenario for each one that satisfies the specification's scenario acceptor. The first call has such a scenario:

```
XFilterEvent(event = 0x10 -> (time = 123))
XSetSelectionOwner[selection = 1](time = 123) [seed]
```

`CheckPositive` correctly reports an error for the second call, which has no such scenario. `NoMatchingScenario` wastes some time on this call. Although the call has just one scenario (that is, the call itself), `NoMatchingScenario` extracts this scenario several times, each time with different values for the forwards and backwards radii. As an optimization, `NoMatchingScenario` could detect when increasing the forwards or backwards radius does not increase the size of the extracted scenario and cut off the search earlier. I have not felt the need to implement this optimization.

8.1.2 Checking negative specifications

A negative specification dictates that programs must not do something. As Chapter 9 reports, just 1 of the 17 specifications I found with Strauss is a negative specification. However, that specification found 9 bugs in widely distributed X11 programs.

In Section 2.4.2 (page 35), I gave a semantics for negative specifications. In

```
  RMAX is a small, nonnegative integer

* Routine CheckNegative(T, spec)
    P := MakeSemanticTrace(T, stm(spec))
    D_flow := InferFlowDependences(P)
    Foreach e ∈ events(P)
      If e matches pattern(spec)
*       If HasMatchingScenario(e, P, D_flow, spec)
          Report an error at e

* Routine HasMatchingScenario(e, P, D_flow, spec)
    r_total := 0
    While r_total <= RMAX
      r_f := 0
      While r_f <= r_total
      S := ExtractScenario((e, r_total − r_f, r_f), P, D_flow)
        If acceptor(spec) accepts S
*         Return true
        r_f := r_f + 1
      r_total := r_total + 1
*   Return false
```

Figure 69: Pseudocode for checking that a trace T obeys a negative specification *spec*.

essence, the semantics says that a program obeys a negative specification iff, in every trace of the program, every event that matches the specification's seed pattern does not have a scenario that is accepted by the specification's scenario acceptor.

A few modifications turn the checker for positive specifications (Figure 67) into a checker for negative specifications. Figure 69 lists pseudocode for the negative specification checker. Most of the code is identical to the code for the positive checker; an asterisk indicates lines that have changed.

The point of the changes is to reverse the sense of error reports. A positive specification dictates what programs must do, so `CheckPositive` reports an error when it finds a seed event with no scenario that matches acceptor(*spec*). By contrast, because a negative specification dictates what programs must *not* do, `CheckNegative` generates an error when it finds a scenario that matches acceptor(*spec*).

Essentially, the negative checker tries to construct a proof that a trace T disobeys a negative specification *spec*. If `CheckNegative` generates an error, then it has constructed a scenario for some seed event that satisfies acceptor(*spec*), and this scenario constitutes a proof that T disobeys *spec*. Like `CheckPositive`, `CheckNegative` is sound but not complete: `CheckNegative` may miss errors in traces that contain them.

8.2 Drawbacks of the dynamic checker

A major drawback of dynamic checkers (in general) is that they only find errors that are exercised by an execution. Thus, dynamic checkers cannot certify that programs are correct. Also, some of the most interesting program errors are hard to execute—for example, errors in exception-handling code—and therefore hard to detect with a dynamic checker.

Besides that basic drawback, which applies to all dynamic checkers, the dynamic checkers described in this chapter are also incomplete, in that they can miss scenarios that would satisfy the specification's scenario acceptor.

The following example demonstrates the checker's incompleteness. Consider checking the specification in Figure 70 on the following trace:

For all trace events matching

```
F(a = X, b = Y)
```

with STM

```
F(use a, use b)
G(def a)
H(def b)
```

a scenario matches

```
G(a = X); F(a = X, b = Y) [seed]
```

Figure 70: A tricky Strauss specification.

```
G(a = 10)

H(b = 20)

F(a = 10, b = 20)
```

The trace satisfies the specification, with the scenario

```
G(a = 10); F(a = 10, b = 20)
```

But this scenario will never be produced by `ExtractScenario`, and the checker will never find it. `ExtractScenario` adds events near the seed to the scenario first. In this case, `ExtractScenario` always adds the call to `H` before it adds the call to `G`. That is, `ExtractScenario` extracts only the following three scenarios:

```
(1) F(a = 10, b = 20)

(2) H(b = 20); F(a = 10, b = 20)

(3) G(a = 10); H(b = 20); F(a = 10, b = 20)
```

This problem could be solved by implementing the Scenarios function (Section 4.2.1, page 73) and using Scenarios to extract *all* scenarios around a given

seed. After all, the semantics of specifications is defined in terms of Scenarios (Section 2.4.1, page 32). I have not done this, for two reasons. First, I have not seen this sort of error in practice. Second, a seed event may have an exponential number of scenarios, so the modified checker would run in exponential time (in the size of the trace).

Note that I am not arguing that all complete dynamic checkers require exponential time. The discussion in the preceding paragraph only applies to checkers that verify traces directly, by extracting scenarios.

Chapter 9

Experience with Strauss

This chapter presents my experience as a user of Strauss. Previous chapters have explained how, given a state transition model and a seed pattern, Strauss extracts scenarios and infers a specification. In addition, Chapter 7 presented a method for debugging finite-state specifications that are overly general or based, in part, on erroneous traces. By contrast, this chapter explores how to use those mechanisms to find useful specifications, which find errors in programs.

My approach here is pragmatic instead of theoretical. I demonstrate that Strauss helps find useful specifications by presenting the specifications that I have found. The main challenge in using Strauss effectively is finding good choices for the state transition model and seed pattern, so I present detailed case studies, which explain how I met that challenge. Finally, to show that my specifications are not trivial, I discuss the program errors that they found.

This chapter is organized as follows. First, Section 9.1 briefly reviews my experimental setup.

Next, Section 9.2 covers the specifications I found and how I found them. In this section, I summarize all of my useful specifications; for two of them, I give a detailed account of the methods I used to find them. The first of these specifications formalizes an existing, informal rule, while the second comes from an automatic

analysis of the traces.

Finally, Section 9.3 covers the errors that my specifications found. I found 199 serious errors in widely distributed X11 programs. These errors were not all alike: I divide them into three different classes and detail the most interesting or severe errors in each class.

9.1 Experimental setup

The specifications in this chapter were mined from traces collected 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.

More information about these traces is not necessary for understanding this chapter. The curious reader will find a thorough description of the traces in Chapter 6.

9.2 The specifications

This section summarizes the useful specifications I found with Strauss and explains, with case studies, the methods I used to find them.

Table 13 lists the specifications and some general facts about them (see Appendix A for an informal description of each specification). For each specification, **STM Size** is the number of annotations in the specification's STM, **Seeds** is the number of event names in the specification's seed pattern, **FA** is the number of states and transitions in the specification's FA, and **Bugs** is the number of true errors and false positives that the specification found.

The true number of program errors is difficult to count, because one program error can cause a number of specification violations. In Table 13, I err on the side of caution by reporting one error and/or one false positive for each program

Name	STM Size	Seeds	FA		Bugs	
			|Q|	|T|	True	False
PrsAccelTbl	9	4	13	22	0	1
PrsTransTbl	5	2	4	5	0	0
Quarks	4	1	10	13	0	0
RegionsAlloc	4	3	30	35	16	0
RegionsBig	33	13	352	623	12	0
RmvTimeOut	3	1	5	6	0	0
XFreeGC	3	2	10	13	44	0
XGContextFromGC	4	3	38	48	44	1
XGetSelOwner	3	1	5	5	9	0
XInternAtom	17	2	7	15	42	0
XPutImage	9	1	7	9	2	2
XSaveContext	8	1	66	86	33	0
XSetFont	4	3	30	40	44	1
XSetSelOwner	16	1	5	9	7	0
XtFree	4	3	29	35	45	0
XtOwnSel	16	1	5	10	1	0
XtRealizeProc	4	2	57	64	0	0

Table 13: Useful specifications for the X11 interface, which I found with Strauss. For each specification, **STM Size** is the number of annotations in the specification's STM, **Seeds** is the number of event names in the specification's seed pattern, **FA** is the number of states and transitions in the specification's FA, and **Bugs** is the number of true errors and false positives that the specification found.

that violates a specification, no matter how many times the program violates the specification.

Also, because a call may violate several specifications, there is overlap among the errors reported in Table 13. Specifically, XFreeGC, XGContextFromGC, and XSetFont all find the same errors, and RegionsAlloc finds every error found by RegionsBig. In total, these specifications found 61 distinct errors. The specifications were remarkably productive: over half of them found program errors, even though my dynamic checker only finds errors that are exercised.

None of the specifications in Table 13 have large STMs, which is good because

large STMs should be harder to find than small STMs. On the other hand, one of them (`RegionsBig`) has an FA with hundreds of states and edges, while several have FAs with scores of states and edges. As Chapter 7 noted, large specifications are harder to debug than smaller specifications; debugging the larger specifications would have been very tedious without the aid of a specification debugger like Cable.

There are several simple *specification patterns* that recur in my specifications. Specification patterns are similar to traditional design patterns [20]: a specification pattern describes a general template for a class of formal specifications, just as a design pattern describes a general solution for a class of programming problems. My specification patterns are also similar to the specification patterns described by Dwyer, Avrunin, and Corbett [15]; however, their patterns are for temporal logic, not Strauss, and do not mention data values. Appendix B documents the specification patterns I found in detail; briefly, the patterns are

Union A specification that combines two simpler specifications, and accepts a trace iff it is accepted by either specification.

Consumer A specification where the state of `X`, which is consumed by a seed event `F(a = X)`, must be produced by an event `G(a = X)`.

Producer A specification where the state of `X`, which is produced by a seed event `F(a = X)`, must be consumed by an event `G(a = X)`.

MultiConsumer A specification where, given a seed event `F(a = X, b = Y)`,

- the state of `X`, which is consumed by `F`, must be produced by an event `G(a = X)`, and

- the state of `Y`, which is consumed by `F`, must be produced by an event `H(a = Y)`.

TiedConsumer A specification where `X` and `Y`, which are consumed by a seed event `F(a = X, b = Y)`, must be produced simultaneously by some preceding event `G(a = X, b = Y)`.

TiedProducer A specification where `X` and `Y`, which are produced by a seed event `F(a = X, b = Y)`, must be consumed simultaneously by some following event `G(a = X, b = Y)`.

MultiConsumerAgrees A specification where, given a seed event `F(a = X, b = Y, c = Z)`,

- the state of `Y`, which is consumed by `F`, must be produced by an event `G(a = X, b = Y)`, and
- the state of `Z`, which is consumed by `F`, must be produced by an event `H(a = X, b = Z)`, and
- all three events use the same `X`.

Table 14 lists the patterns followed by each of specification. Note that none of the specification patterns (and hence none of the specifications) have FAs with loops or even chains of data dependences. However, most of the specifications are Unions, and so do not follow extremely simple patterns like "A follows B". Thus, my specifications are intermediate between the complicated specifications that people write, and the very simple specifications inferred by systems like Xgcc. [27]

Name	Method	Patterns
PrsAccelTbl	Book	Union: Consumer
PrsTransTbl	Book	Union: Consumer
Quarks	Graph	MultiConsumer
RegionsAlloc	Graph	Union: Consumer, Producer
RegionsBig	Graph	Union: Consumer, Producer
RmvTimeOut	Book	Consumer
XFreeGC	Graph	Union: Consumer, Producer
XGContextFromGC	Graph	Union: Consumer, Producer
XGetSelOwner	Book	Producer
XInternAtom	Book	Union: Consumer (negative)
XPutImage	Graph	Union: MultiConsumerAgrees
XSaveContext	Graph	TiedConsumer
XSetFont	Graph	Union: Consumer, Producer
XSetSelOwner	Book	Union: Consumer
XtFree	Graph	Union: Consumer, Producer
XtOwnSel	Book	Union: Consumer
XtRealizeProc	Graph	Union: TiedConsumer, TiedProducer

Table 14: For each specification, how I found it (**Method**) and the patterns it follows (**Patterns**).

Table 14 also lists how I found each specification. I used two methods. The Book method starts with an existing rule (usually from a programming manual) and derives inputs that cause Strauss to mine an equivalent but formal specification. Section 9.2.1 describes how I used the Book method to mine the XSetSelOwner specification.

The Graph method, by contrast, finds the inputs for Strauss automatically, by finding connected components in an affinity graph. Section 9.2.2 details this method and shows how I used it to mine the XPutImage specification.

9.2.1 Case study: the Book method

Consider the following informal rule:

> The timestamp passed to `XSetSelectionOwner` must come from an event received from the X server.

This case study explains how I derived a Strauss specification (XSetSelOwner) from this rule. The central problem was choosing the right seed and state transition model to use as inputs to the miner.

Choosing the seed was easy. `XSetSelectionOwner` is the only X11 routine mentioned in the rule, so I chose it as the seed.

I was also able to make some quick initial progress on the state transition model. `XSetSelectionOwner` has four arguments. One of them is named `time` and is of type `Time`; this must be the timestamp referred to in the rule. The rule says that `XSetSelectionOwner` receives this timestamp, so the state transition model must include the declaration:

```
XSetSelectionOwner(use time)
```

To complete this state transition model, I had to know how events are "received from the X server"; the routines that receive events will become definers in the model. One way to find these routines is to consult the documentation. A section titled "Routines that Get Events" of the Xlib Programming Manual [39] lists 16 routines, 12 of which receive events. But, this section says nothing about where to find an event's timestamp. Also, Xlib is just one part of the X11 interface. The interface has routines that receive events and are not listed in this manual. In short, finding all routines that receive events by hand is difficult and error-prone.

A better way is to use Strauss to find all trace calls that produce a timestamp used by `XSetSelectionOwner`. Chapter 3 defined the liberal completion of a partial

state transition model: the liberal completion assigns a def effect to arguments with no explicit def or not-def effect and a use effect to arguments with no explicit use or not-use effect. That is, if a routine is not listed in the partial state transition model, then the liberal completion treats all of its arguments as both definers and users. This partial state transition model explicitly says that `time` is the only argument of `XSetSelectionOwner` that uses the state of its value:

```
XSetSelectionOwner(use time,
                   not-use display,
                   not-use selection,
                   not-use owner)
```

To find the calls that produce the timestamps used at `time`, I used Strauss to extract scenarios with backwards-radius 1, `XSetSelectionOwner` as the seed, and dependences determined by the liberal completion of this partial model. On the X11 traces, Strauss found scenarios that mentioned 44 defining arguments. Unfortunately, many of these arguments are from routines that appear to have nothing to do with receiving events:

```
XCopyArea(def src_x, def src_y, def dest_x, def dest_y)
XDrawLine(def x2)
return XDrawString(def ?)
```

What went wrong? It turned out that the above routines appeared in the scenarios because of spurious dependences carried by the value 0. 0 is a very common value, and the fact that it appears at many arguments in a trace does not

indicate that those arguments are related. After eliminating scenarios where `time` was 0, 7 defining arguments remained:

```
XFilterEvent(event -> def time)
XtDispatchEvent(event -> def time)
XGrabPointer(def time)
XUngrabPointer(def time)
return XNextEvent(event def time)
return XWindowEvent(event -> def time)
callback_XtActionProc(widget -> event -> def time)
```

Note that `XGrabPointer` and `XUngrabPointer` do not have an argument named `event`. The informal rule is about receiving events, so I did not add those routines to the state transition model. After adding the other routines to the original state transition model, Strauss found the specification in Figure 71.[1]

Note that `XtDispatchEvent` and `callback_XtActionProc` are not mentioned as event receiving routines in the Xlib manual: both belong to the X Toolkit Intrinsics part of the interface (not to Xlib), and `callback_XtActionProc` is a callback, not a library routine. I do not write X11 programs, and without the aid of Strauss, I might have thought to add `XtDispatchEvent` to the model, but I doubt that I would have realized that `callback_XtActionProc` also belongs there.

[1] As presented, this specification includes return events without their corresponding call events. The original Strauss specification does include those call events, and as a result is slightly larger than the specification here.

For all trace events matching

```
XSetSelectionOwner(time = X)
```

with STM

```
XSetSelectionOwner(use time, add_to_name selection)
XFilterEvent(event -> def time)
XtDispatchEvent(event -> def time)
return XNextEvent(event def time)
return XWindowEvent(event -> def time)
callback_XtActionProc(widget -> event -> def time)
```

a scenario matches

```
(XFilterEvent(event -> time = X)
 | XtDispatchEvent(event -> time = X)
 | XNextEvent returns (event -> time = X)
 | XWindowEvent returns (event -> time = X)
 | callback_XtActionProc(widget -> event -> time = X));
XSetSelectionOwner[selection = 1](time = X) [seed]
```

Figure 71: XSetSelOwner. A Strauss specification for the informal rule, "The timestamp passed to `XSetSelectionOwner` must come from an event received from the X server."

9.2.2 Case study: the Graph method

In the preceding case study, I found a Strauss specification with the help of a large hint: an informal rule with the same intent as the specification. This case study shows how to develop a Strauss specification without such a hint. The key step is finding an appropriate state transition model; before I can explain how to do that, I must explain what I mean by "appropriate".

Appropriate state transition models

An *appropriate state transition model* lists interface arguments such that each pair of arguments is potentially dependent, either directly or indirectly. Suppose that a_0 and a_1 are interface arguments. I say that a_0 and a_1 are *potentially dependent* iff

- a_0 and a_1 are arguments of the same routine, or
- there is a trace T and a value v such that
  - T has a call that binds v to a_0, and
  - T has a call that binds v to a_1, and
  - at both calls, v indexes the same chunk of implementation state.

Also, I say that a_0 and a_1 are *indirectly potentially dependent* iff

1. a_0 and a_1 are potentially dependent, or
2. there is an a_2 such that a_0 and a_2 are indirectly potentially dependent, and a_1 and a_2 are indirectly potentially dependent.

To understand potential dependence, it helps to think about what happens when the above conditions do not hold. Suppose that there is no value v in *any* trace T such that a_0 and a_1 are both bound to v somewhere in T. Then, no matter what state transition model is chosen, Strauss never infers a dependence between an instance of a_0 and an instance of a_1. On the other hand, if there is such a v and T, then there is a model that causes Strauss to infer a dependence between

an instance of a_0 and an instance of a_1. However, if v indexes a different chunk of implementation state at each instance, then the dependence is surely false.

The state transition model in Figure 71 is appropriate. The event-receiving arguments are each directly potentially dependent with the `time` argument of `XSetSelectionOwner`, and indirectly potentially dependent with each other.

Finding appropriate models automatically

I am now ready to explain the Graph method for finding appropriate state transitions models automatically. To motivate the method, I will start by presenting an inferior method and analyzing its shortcomings.

The inferior method and the Graph method both use an *argument affinity graph (AAG)* as the basic data structure. The nodes of the AAG are routine arguments. Edges have positive weights: higher weights are supposed to indicate pairs of arguments that are more likely to be potentially dependent. Both methods generate state transition models directly from the AAG by removing edges whose weight falls below some threshold and listing the connected components in the reduced graph. The two methods differ only in how they assign the weights.

The inferior method's weight function is based on the tactic I used to find the state transition model for the XSetSelOwner specification. In that case study, I found the state transition model by searching (with Strauss) for event arguments that shared a dependence with the `time` argument of a call of `XSetSelectionOwner`, under a very liberal state transition model. Similarly, with the inferior method, if a_0 and a_1 are routine arguments, the weight of the AAG edge between a_0 and a_1 equals the number of times that instances of a_0 and a_1 share a dependence, again

under a very liberal state transition model.

Following dependences worked adequately for XSetSelOwner, but there were two problems along the way:

- The common value 0 caused many spurious dependences. The underlying problem is that the value 0 does not satisfy the third condition for potential dependence: when the value 0 occurs at two call arguments, it is unlikely that it indexes the same chunk of implementation state at both calls.

- The routine `XCheckWindowEvent` was missing in the initial specification. Calls of `XCheckWindowEvent` never appeared outside of sequences like

  ```
  XCheckWindowEvent; XGrabPointer; XSetSelectionOwner
  ```

  That is, an irrelevant call (`XGrabPointer`) shadowed the significant call (`XCheckWindowEvent`).

The same problems plague the inferior method. AAG edges with a high weight tend to come from arguments that share common values like 0; such edges tend not to connect potentially dependent arguments. Also, arguments that are potentially dependent often have no AAG edge connecting them, because of shadowing. The result is that connected components in the AAG tend not to be appropriate state transition models.

The Graph method avoids both problems. The method assigns low weights to AAG edges that are connected only by common values, and avoids shadowing by ignoring the order of trace events. For each routine argument a, the method records the values that occur at instances of a in a number of traces. Call this set

of values V_a. Intuitively, if $V_{a_0} \cap V_{a_1}$ contains many uncommon values, then a_0 and a_1 are likely to be potentially dependent.

More specifically, the weight $w(a_0, a_1)$ of the AAG edge between a_0 and a_1 should vary inversely with the probability that a_0 and a_1 would share the values $V_{a_0} \cap V_{a_1}$ purely by chance. That is, weights should be defined as follows:

> Let $f : V \to N$ map each trace value v to the frequency with which v occurs in a trace (or set of traces). Place $f(v)$ copies of each value v into an urn. Consider the following procedure for drawing a set of m values from the urn: draw a value v from the urn, add v to the set, and remove all copies of v from the urn; repeat until m values are drawn. Now, let $P(E, m_0, m_1)$ be the probability that, after drawing a set V_0 (with $m_0 = |V_0|$) and a set V_1 (with $m_1 = |V_1|$) independently in this fashion, $E = V_0 \cap V_1$. Finally, let $w_{\text{hard}}(a_0, a_1) = 1/P(V_{a_0} \cap V_{a_1}, |V_{a_0}|, |V_{a_1}|)$.

Eric Bach pointed out to me that this is an example of a Polya-Eggenberger urn model [33]. Unfortunately, I cannot think of any way to compute $w_{\text{hard}}(a_0, a_1)$ in less than exponential time. Therefore, the Graph method makes some simplifying assumptions:

- Draws from the urn are independent. After drawing v from the urn, instead of removing all copies of v from the urn, v is returned to the urn. With this assumption, the probability of drawing v in m draws is

$$1 - (1 - f(v)/f(V))^m$$

- The probability of drawing a value v is independent of the probability of drawing any other value. This is a simplification, because in fact the number

of draws is fixed. With this assumption, the probability of drawing all of the values in a set E in m draws is

$$\prod_{v \in E} 1 - (1 - f(v)/f(V))^m$$

With these assumptions, computing weights is easy:

$$w(a_0, a_1) = (\prod_{v \in E} 1 - (1 - f(v)/f(V))^{m_0})^{-1} (\prod_{v \in E} 1 - (1 - f(v)/f(V))^{m_1})^{-1}$$

where $E = V_{a_0} \cap V_{a_1}$.

This weighting function performs well in practice. I built the AAG[2] for the first 100000 events in 8 traces and used it to infer 25 state transition models. Of these, 10 models led to useful specifications (the Graph specifications in Table 14).

There are some practical problems with building and using the AAG, which I have dealt with only partially:

- The AAG can be unmanageably large. For my training set, the AAG had about 14000 nodes and at least 5 million edges: I had to stop adding edges to the AAG because the machine I used ran out of random access memory and started thrashing.

  I dealt with this problem by building approximations of the AAG. First, I built a routine affinity graph (RAG). The difference between the RAG and the AAG is that the nodes of the RAG represent routines instead of routine arguments. Next, I found RAG components that were connected by edges whose weights exceeded an empirically determined cutoff. For each such

[2] I actually built approximations of the AAG; see the list of practical issues below.

component, I built a small AAG with a node for each argument of each routine in the component.

Most of the edges in the unmanageable large AAG were of low weight. It should be possible to use this fact to find a small subgraph of the full AAG without losing accuracy, but I have not done this.

- Structured data add noise to the AAG. That is, if $w(a_0, a_1)$ is high and $a_0.x$ and $a_1.x$ are fields of a_0 and a_1, then $w(a_0.x, a_1.x)$ is also likely to be high. Two weighty connected components result, one that contains a_0 and a_1 and another that contains $a_0.x$ and $a_1.x$. But, the second component is uninteresting, because it is implied by the first.

  I dealt with this problem by ignoring connected components for fields when a connected component also existed for the containing arguments.

  I believe that this problem could be addressed by adding nodes for fields only when their edge weights differ from those of their containing structures. This solution would also reduce the size of the AAG, without losing accuracy.

- A connected component in the AAG dictates which routine arguments appear in a state transition model. However, it cannot be used to decide if an argument should have a def effect, a use effect, or both.

  This is not a major problem. I have dealt with it by exhaustively searching for a good set of effects, by using the names of routines and arguments as hints (see the example below), and by inspecting traces to determine if any arguments always precede or follow other arguments. The first and the last

of these solutions could be automated.

Example: XPutImage

Here is a connected component, which I found with the AAG and used to mine the XPutImage specification:

```
XCreateGC(display)
return XCreateGC(?)
XCreateImage(display)
return XCreateImage(?)
XShmCreateImage(display)
return XShmCreateImage(?)
XPutImage(display, gc, image)
```

Based on their names, I decided to annotate the return arguments of `XCreateGC`, `XCreateImage`, and `XShmCreateImage` with def effects. `XPutImage` was the only routine left, so I annotated its arguments with use effects. Initially, I also annotated the `display` arguments of `XCreateGC`, `XCreateImage`, and `XShmCreateImage` with def effects. With these effects, the complete state transition model was

```
XCreateGC(def display)
return XCreateGC(def ?)
XCreateImage(def display)
return XCreateImage(def ?)
XShmCreateImage(def display)
return XShmCreateImage(def ?)
```

For all trace events matching

```
XPutImage(display = X, gc = Y, image = Z)
```

with STM

```
XCreateGC(use display)
return XCreateGC(def ?)
XCreateImage(use display)
return XCreateImage(def ?)
XShmCreateImage(use display)
return XShmCreateImage(def ?)
XPutImage(use display, use gc, use image)
```

a scenario matches

```
XCreateGC(display = X);
XCreateGC returns (?  = Y);
((XCreateImage(display = X);
  XCreateImage returns Z)
 | (XShmCreateImage(display = X);
    XShmCreateImage returns Z));
XPutImage(display = X, gc = Y, image = Z) [seed]
```

Figure 72: XPutImage. A Strauss specification for the informal rule, "The image and graphics context passed to `XPutImage` must have been created on the same display."

```
XPutImage(use display, use gc, use image)
```

This model did not work well. Using `XPutImage` as the seed because it was the only consuming routine, I used Strauss to mine a specification. The specification was complex, because in many scenarios the producer of the `gc` or `image` argument of `XPutImage` was shadowed by an intervening call, which defined the `display` argument.

Name	Kinds	Violations			Motif bugs
		False	Bugs	Minor	
PrsAccelTbl	hard	1	0	0	0
PrsTransTbl	hard	0	0	0	0
Quarks	hard	0	0	0	0
RegionsAlloc	leak,hard	6	0	0	0
RegionsBig	leak,hard	5	0	0	0
RmvTimeOut	hard	0	0	0	0
XFreeGC	leak,hard	3	6	76	0
XGContextFromGC	leak,hard	3	6	76	0
XGetSelOwner	protocol	0	11	0	0
XInternAtom	performance	9	32	281	32
XPutImage	hard	2	2	0	1
XSaveContext	leak	4	2	31	2
XSetFont	leak,hard	3	6	76	0
XSetSelOwner	protocol	0	7	0	0
XtFree	leak,hard	11	0	1355	0
XtOwnSel	protocol	0	1	0	0
XtRealizeProc	hard	0	0	0	0

Table 15: The errors that my specifications found. For each specification, **Error kinds** lists the kinds of errors that the specification finds; **Violations** is the number of false errors, significant bugs, and minor violations that the specification found; and **Motif bugs** is the number of those significant bugs that were in the Motif widget library [40].

Next, I tried annotating *all* of the `display` arguments with use effects. The resulting specification, which I accepted, is in Figure 72. [[**Explain this specification, in English.**]]

9.3 The errors

The specifications that I found with Strauss found a variety of serious program errors in widely distributed X11 programs. Table 15 is an overview of the violations reported by the specifications. The first column of the table lists the specifications.

The column headed **Error kinds** lists the kinds of errors that each specification finds; I discuss my system for classifying errors below.

The remaining columns classify the violations that each specification found in the original 90 traces. Altogether, I found 1839 static call sites at which one or more specifications were violated. For each specification, I ranked these call sites by the severity of their dynamic violations (see below) and inspected the call sites that appeared to generate the worst violations. For each specification, the column headed **False** lists the number of call sites that I decided were not erroneous, despite the fact that they violated the specification. The column headed **Bugs** lists the number of call sites that I decided were erroneous; I sent bug reports for these violations to the programs' authors. Many of the erroneous call sites were inside the Motif widget library [40]; the column headed **Motif bugs** lists the number of Motif bugs. Finally, the column headed **Minor** lists the number of call sites that I did not inspect, because the violations did not appear severe.

There is some overlap among the violations found by these specifications. Specifically, XFreeGC, XGContextFromGC, and XSetFont all find the same violations, and RegionsAlloc finds every violation found by RegionsBig.

I divide errors into four categories (the **Error kinds** column):

- Hard errors. Hard errors are unambiguously erroneous: the usage in question would be incorrect in any program that uses the interface. An example would be acquiring a lock and never releasing it.

  I found just 2 hard errors in the programs I analyzed. These errors were both violations of the XPutImage specification discussed in Section 9.2.2: in one

violation, a program created an image by filling in structure fields, instead of using the X11 routines provided for that purpose, and in the other violation, a program created the image and graphics context on different displays.

- Protocol errors. These errors are only errors for programs that should follow some externally imposed protocol. For example, many protocols exist for communication between X11 programs. These protocols are independent of the X11 interface, although they depend on its mechanisms. I found 19 protocol errors.

- Resource leaks. A leak occurs when a dynamically allocated resource, such as memory or any of a number of resources provided by X11, is retained past its time of potential usefulness. Leaks are hard to identify reliably, because the potentially useful lifespan of a resource might be an entire execution. In this case, there is usually no requirement that programs explicitly deallocate their resources—the system deallocates most resources automatically when execution terminates.

  Unlike protocol errors or hard errors, resource leaks can range in severity. For example, a program that unnecessarily retains ten bytes of dynamically allocated memory has a benign leak, while a program that unnecessarily retains ten megabytes of dynamically allocated memory has a larger problem. I found 8 resource leaks that I judged severe enough to report.

- Performance errors. A performance error is any waste of system resources that is not a leak. Just as the severity of resource leaks depend on how many resources are leaked, the severity of performance errors often depend

on how often an operation is performed: for example, creating one window is reasonable, but creating ten thousand windows is probably an error. I found 32 performance errors that I judged severe enough to report.

Note that some specifications can find both resource leaks and hard errors. For example, the XFreeGC specification captures this informal rule:

> 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.

A violation of the first statement is a potential leak, while a violation of the second statement is a hard error. However, in my experiments, specifications that can find both leaks and hard errors in fact found no hard errors, just leaks.

The rest of this section presents a case studies of specification that find protocol errors, resource leaks, and performance errors.

9.3.1 Protocol errors: XSetSelOwner

Figure 71 lists the XSetSelOwner specification, which finds protocol errors. The specification captures this informal rule:

> The timestamp passed to `XSetSelectionOwner` must come from an event received from the X server.

One way for X11 processes to communicate with one another is via the selection mechanism. The most common use of the mechanism is for pasting text (or other data) from one window to another. Typically, the user highlights some text in

one window (window A) by making a series of mouse clicks and movements; then, he pastes the text into another window (window B) by moving the pointer over window B and clicking a mouse button.

Behind the scenes, the processes responsible for the two windows communicate according to a conventional protocol, which is described in the Interclient Communications Conventions Manual [45]. In this simple example, after the user highlights the text in window A, the process responsible for window A requests that window A be made the owner of the *primary selection*, by calling `XSetSelectionOwner` with appropriate arguments. The X11 server remembers that window A owns the primary selection. Later, when the user clicks the mouse over window B, the process responsible for window B asks the server to forward a request for the primary selection to the current owner. The server posts a `SelectionNotify` event to window A. Window A's process responds by sending the data to window B, and window B's process redisplays window B appropriately.

A central goal of the protocol is to avoid races for selection ownership, which can leave the user confused about which region of text is currently selected. To this end, the protocol requires that processes use timestamps to keep track of when they owned the selection. The XSetSelOwner specification captures one aspect of this part of the protocol. Unfortunately, the interface does not enforce the timestamp conventions: in fact, the interface allows a process to substitute a special constant, `CurrentTime`, for the timestamp in a call of `XSetSelectionOwner`. If the X11 server receives a request with `CurrentTime` instead of a timestamp, it treats the request as if it were stamped with the current time at the server.

Here is an example that shows that violating the XSetSelOwner specification permits confusing races to occur. Suppose that the user selects some text in window A, and that window A's process asks the server for ownership of the primary selection but uses `CurrentTime` instead of a proper timestamp in the request. Next, suppose that the user changes his mind and selects some text in window B; in response, window B's process asks the server for ownership of the primary selection. From the user's point of view, window B should own the selection, because the user selected that text after he selected the text in window A. However, requests can arrive at the X server out of order, and if window A's request arrives after window B's request, the server will assign the selection to window A (because window A's process used `CurrentTime`). This sequence of events is surprising for the user: if he pastes into window C, the data from window A will appear, but the user thinks that his current selection is in window B.

I found 7 programs that violate the XSetSelOwner specification: `coolinput`, `e93`, `gnuplot`, `ups`, `wish8.3`, `xpdf`, and `xpilot`. All of the programs violated the specification by using `CurrentTime` instead of a proper timestamp in an `XSetSelectionOwner` call. Interestingly, 5 of the programs had comments in which the programmer said that he knew that his code violated the protocol, but that writing correct code was either too difficult or perceived to be unnecessary. A typical example is this comment from the code for `wish8.3`:

```
/*
 * Note that we are using CurrentTime, even though ICCCM recommends
 * against this practice (the problem is that we don't necessarily
 * have a valid time to use).  We will not be able to retrieve a
```

```
* useful timestamp for the TIMESTAMP target later.
*/
```

Comments like this suggest that, for the most part, these errors are not careless mistakes. Instead, the ICCCM protocol is too hard to follow—specifically, programmers must jump through hoops to keep track of the timestamp—and even careful programmers have trouble.

In their responses to my bug reports, two developers independently made another argument against the ICCCM protocol [47, 13]. They pointed out that some requests for the selection are not prompted by an X11 event: their example was a program with a built-in scripting language, which allows scripts to set the selection. In this case, it is not clear which timestamp to use when requesting ownership of the selection.

9.3.2 Resource leaks: RegionsAlloc and XFreeGC

Figure 73 lists the RegionsAlloc specification, which finds resource leaks. The specification captures this informal rule:

> 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.

In X11, a `Region` is represents a set of pixels in a plane. They have two main purposes: the X11 library uses them to tell programs which parts of windows must be redrawn, and programs use them to tell the X11 library which clip-mask to use for commands that draw into windows.

Each `Region` owns allocated memory, which can only be deallocated by destroying the `Region` with `XDestroyRegion`. If the `Region` was allocated by the library, the library is responsible for freeing this memory. Otherwise, the program must free it.

I found 12 programs that violate the RegionsAlloc specification. However, all of the violations occurred at 6 different points within the Motif library [40], which is used by all of the programs. At each point, the Motif library created a `Region` and stored it within the data structure for a widget. In Motif, each widget has a `Destroy` method. The `Destroy` methods for the widgets in question would have destroyed the `Region`s, but they were never invoked by the process. In other words, the low-level errors I found (failing to destroy regions) exposed errors at a higher level (failing to destroy widgets).

However, I decided not to report any of these errors, because the leaks were not severe: no program failed to destroy more than 4 Regions, and, moreover, the system reclaims the Regions's memory at program termination.

9.3.3 Performance errors: XInternAtom

Figure 74 lists the XInternAtom specification, which finds performance errors. The specification captures this informal rule:

> For good performance, `XInternAtom` should not be called in the event loop.

Here is what this rule means. An `XInternAtom` call asks the X server to allocate a unique identifier for a character string (in X11 parlance, the server *interns* the

string), so that the identifier can be used in subsequent calls or to communicate with other clients of the server. The main advantage of using an identifier instead of a full string is speed: the identifier takes less time to send across a network than does a long string.

`XInternAtom` also has costs. There is a space cost, because the X server keeps the string and its identifier in an internal hash table until the last connection to the X server closes. There is also a time cost, because `XInternAtom` blocks: a call to `XInternAtom` sends a request to the X server and does not return until the server's reply is received.

The event loop of an X11 program is the period during which the program receives events from the server. This period characterizes the interactive phase of the program. In general, blocking during the event loop (for example, by calling `XInternAtom`) is a bad idea because a blocked program cannot interact with the user.

Sometimes, blocking in the event loop is unavoidable. For example, the program might need some input from the user before it can decide which strings should be interned. This is why violations of the XInternAtom specification are performance errors instead of hard errors.

In fact, I found that 42 programs (out of 72) violate the XInternAtom specification. However, the plot in Figure 75 shows that most programs do not violate the specification very often. On the y-axis of the plot is the number of violations; on the x-axis is the number of programs that commit at least that many violations. 33 out of the 42 programs commit fewer than 500 violations.

The program that commits the most violations is xhtml, which commits 2995 violations. As it happens, most of these violations are due to bad coding practices in the Motif library [40]. For example, the most common violation (701 occurrences) is caused by this code (from the file `TextF.c`):

```
/* ARGSUSED */
static Boolean
SetDestination(Widget w,
               XmTextPosition position,
               Boolean disown,
               Time set_time)
    {
      XmTextFieldWidget tf = (XmTextFieldWidget) w;
      Boolean result = TRUE;
      Atom MOTIF_DESTINATION = XInternAtom(XtDisplay(w),
                                           XmS_MOTIF_DESTINATION,
                                           False);
```

`SetDestination` is called frequently by `TextField` widgets, which allow users to enter a small amount of text. For example, the routine is called every time the user changes the position of the cursor within the text field. Many other common user interactions also cause `SetDestination` to be called, and each call forces a slow round-trip with the X server. All but one of these round-trips are completely unnecessary, because `XmS_MOTIF_DESTINATION` is a constant string.

Similar errors pervade the Motif library, and these errors affect all of the many programs that use the library. Often, as in this case, the problem could be fixed simply by making a variable static. A better solution (used in the GTK+ toolkit [42]) is to replace calls to `XInternAtom` with calls to a wrapper routine that caches requests and only calls `XInternAtom` for strings whose identifiers are not in the cache.

For all trace events matching

```
XCreateRegion returns (?  = X) | XPolygonRegion()
| DestroyRegion(r = X)
```

with STM

```
return XCreateRegion(def ?)
return XPolygonRegion(def ?)
XDestroyRegion(def use r)
```

a scenario matches

```
((XCreateRegion returns (?  = X) [seed];
  XDestroyRegion(r = X); XDestroyRegion returns ();
  (ϵ | DEFINE(X)))
 | (DEFINE(X); XCreateRegion returns (?  = X) [seed];
    XDestroyRegion(r = X); XDestroyRegion returns ();
    (ϵ | DEFINE(X))))
| ((ϵ | DEFINE(X));
   XPolygonRegion () [seed]; XPolygonRegion returns (?  = X);
   XDestroyRegion(r = X); XDestroyRegion returns ();
   (ϵ | DEFINE(X)))
| ((DEFINE(X);
    ((XCreateRegion returns (?  = X);
      XDestroyRegion(r = X) [seed]; XDestroyRegion returns ();
      (ϵ | DEFINE(X)))
     | (XPolygonRegion returns (?  = X);
        XDestroyRegion(r = X) [seed];
        XDestroyRegion returns (); (ϵ | DEFINE(X)))))
   | (XCreateRegion returns (?  = X);
      XDestroyRegion(r = X) [seed]; XDestroyRegion returns ();
      (ϵ | DEFINE(X)))
   | (XPolygonRegion returns (?  = X);
      XDestroyRegion(r = X) [seed]; XDestroyRegion returns ();
      (ϵ | DEFINE(X))))
```

Figure 73: RegionsAlloc. A Strauss specification for the informal rule, "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."

For all trace events matching

```
XInternAtom(display = X)
```

with STM

```
XInternAtom(use display)
XFilterEvent(event -> def display)
XtDispatchEvent(event -> def display)
XtCallActionProc(event -> def display)
return XtAppNextEvent(event -> def display)
return XtNextEvent(event -> def display)
return XWindowEvent(event -> def display)
return XCheckWindowEvent(event -> def display)
callback_XtEventHandler(event -> def display)
callback_XtActionProc(event -> def display)
```

no scenario matches

```
(XFilterEvent(event -> display = X)
 | XtDispatchEvent(event -> display = X)
 | XtCallActionProc(event -> display = X)
 | (XtAppNextEvent();
    XtAppNextEvent returns (event -> display = X))
 | (XNextEvent();
    XNextEvent returns (event -> display = X))
 | (XWindowEvent();
    XWindowEvent returns (event -> display = X))
 | (XCheckWindowEvent();
    XCheckWindowEvent returns (event -> display = X))
 | callback_XtEventHandler(event -> display = X)
 | callback_XtActionProc(widget -> event -> display = X));
XInternAtom(display = X) [seed]
```

Figure 74: XInternAtom. A Strauss specification for the informal rule, "For good performance, `XInternAtom` should not be called in the event loop."

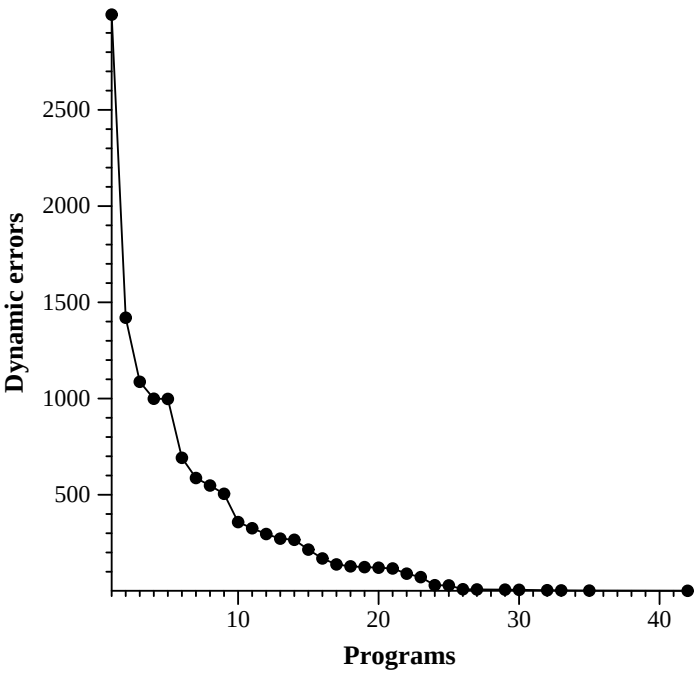

Figure 75: Programs versus violations of the XInternAtom specification. On the y-axis is the number of violations; on the x-axis is the number of programs that commit at least that many violations.

Chapter 10

Final Thoughts

The specifications and errors that I found with Strauss (Chapter 9) are strong evidence for my thesis that specifications can be mined. However, Strauss is still a long way from being a practical system. This chapter discusses some of the issues that stand between this promising beginning and the wide application of specification mining (Section 10.1), as well as what I would do differently if I could start this project over with the benefit of perfect hindsight (Section 10.2).

10.1 Suggestions for future work

10.1.1 Increasing automation

Strauss automates much of the process of specification mining but still relies on a person to validate specifications. I doubt that people will rely on machines to validate specifications any time soon, but I believe that future miners will be less likely to generate invalid specifications (or specifications that are too complex to be validated) than is Strauss. They will achieve this by exploiting knowledge that Strauss currently ignores.

The specifications that I found with Strauss followed a few simple patterns (Appendix B). It seems likely that certain patterns occur more often in valid specifications than do others. If this conjecture is true, biased miners, which prefer specifications that follow common patterns to specifications that follow uncommon patterns, would be much less likely to suggest specifications that a specification developer would reject as invalid.

Another approach could simplify complex specifications by exploiting the labels that the specification developer provides when he debugs a specification with Cable. The labels distinguish scenarios that should be accepted by the final specification from scenarios that should not be accepted. Currently, Strauss keeps the state transition model constant and learns a specification that accepts only appropriately labeled scenarios.

However, the labels can also be regarded as labels on *seed events*. Taking that point of view and applying Ockham's Razor, a specification miner could learn specifications for several variations of the original state transition model and choose

the simplest specification that still distinguishes events with different labels.

Finally, Strauss treats the library as a black box. This is an appropriate assumption for systems where the code for the library is unavailable; on the other hand, if library code is available, there is no reason not to use it. For example, value flow within the library might suggest good state transition models. Also, analyzing a library's error-handling code can be particularly fruitful, as recently demonstrated for Java classes by Whaley, Martin, and Lam [54].

10.1.2 Mining without traces

Mining from traces has disadvantages. Most importantly, gathering a trace requires running a program, which is impossible for incomplete programs and difficult for many large systems. Also, gathering representative traces requires representative inputs, which may not exist. Finally, mining from traces can be slow. Although Strauss scales well, in the sense that its mining time is roughly linear in the size of the input traces, traces can be very large. For example, Chapter 6 discussed a trace with almost three and a half million events! Processing very large traces takes a long time, even with linear algorithms.

Mining directly from program text would avoid these disadvantages. Programs can be analyzed even if they cannot be run. Also, programs are much smaller than traces, and (unlike traces) large programs can be decomposed and analyzed modularly.

Unfortunately, analyzing programs is harder than analyzing traces. By definition, a trace records a feasible execution of a program, but it is impossible, in

general, to decide which program executions are feasible. Even statically determining the arguments of a call requires points-to analysis, another hard problem [7].

Regardless, Hallem and others have made progress recently on mining simple temporal specifications statically [27, 54]. Ultimately, I suspect that the most successful specification miners will be hybrids, which combine sophisticated static analyses with summaries (not full traces) of dynamic executions. Such miners could even work on-line (as a program runs), if the overhead can be made low enough.

10.1.3 Static checking

The dynamic checkers described in Chapter 8 cannot find errors that are not exercised. This is a severe limitation; if specification mining is to be adopted widely, we must have tools that can verify mined specifications statically.

There are two ways that a static checker for Strauss specifications might work. The first kind of tool would construct a scenario that satisfies a specification's automaton, for each seed event encountered while simulating a trace through some abstraction of the program. Conceptually at least, this approach is a direct extension of the dynamic checkers to the static world.

An alternative is to translate a Strauss specification into a traditional temporal formula, which can be checked with a model checker. This is probably the approach to try first.

10.1.4 Modeling a heap

The semantic traces described in Chapter 3 have no heap. Consequently, Strauss cannot learn specifications that involve points-to relations or reachability.

The ability to express such specifications would be useful in many domains. For example, in the X Toolkit Intrinsics library, each widget contains a pointer to its class—there is a class for scrollbars, edit boxes, buttons, and many others. Widget classes are organized in an extensible hierarchy. A developer who wishes to implement a special kind of widget—say, a new kind of scrollbar—does so by creating a class, which he inserts in the hierarchy underneath the base class by setting appropriate pointers. Therefore, expressing a rule about a certain kind of widget (say, scrollbars) requires some notion of reachability: a widget is a scrollbar iff the scrollbar widget class can be reached by following pointers from the widget's class.

Modeling a heap would have a large impact on the design of the miner. Currently, Strauss always maintains the distinction between different trace values. A miner that modeled a heap could not afford this restriction.

10.2 What I would do differently

This section answers the question: "if you could start this project over with the benefit of perfect hindsight, what would you do differently?"

All along, this project lacked a concrete consumer for the specifications. If I could start over, I would try very hard to mine specifications for a specific, existing program-verification tool. Preferably, the tool would be a static checker, because

I believe that the specifications in Chapter 9 would find more bugs if they were checked statically.

On a related note, I would also start with more concrete examples of the sorts of specifications I wished to mine. This project started with the idea of mining temporal specifications, and a few example specifications from a manual on inter-client communication in X11 [45]. It should have started with more real specifications that could be checked by a real checker, because mining those specifications would have been a good test of any proposed miner.

Finally, I would try harder to develop an efficient and easy-to-use infrastructure. It took me months to write a tracer and gather the traces for the X11 interface; I should have looked hard for an easier interface to trace. Now that I have the traces, mining a specification for a new state transition model takes about forty-five minutes on an eight processor SMP, not counting time spent debugging the specification with Cable or debugging Strauss itself. In my experience, about one in four STMs leads to a useful specification, so this cost is a major deterrent to using Strauss. More importantly, it is also a deterrent to experimenting with new ideas.

Appendix A

Descriptions of specifications

PrsAccelTbl Accelerator tables must be parsed with `XtParseAcceleratorTable` before they are used.

PrsTransTbl Translation tables must be parsed with `XtParseTranslationTable` before they are used.

Quarks The `name` and `the_class` arguments of `XrmQGetResource` must come from calls to `XrmPermStringToQuark`.

RegionsAlloc 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 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 `XtRemoveTimeOut` can only remove time-outs added with `XtAddTimeOut`.

XFreeGC 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 `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 After calling `XSetSelectionOwner`, selection ownership must be verified by calling `XGetSelectionOwner`.

XInternAtom For good performance, `XInternAtom` should not be called in the event loop.

XPutImage The image and graphics context passed to `XPutImage` must have been created on the same display.

XSaveContext 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 `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 The timestamp passed to `XSetSelectionOwner` must come from an event received from the X server.

XtFree 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 The timestamp passed to `XtOwnSelection` must come from an event received from the X server.

XtRealizeProc 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.

Appendix B

Specification patterns

B.0.1 The Consumer pattern

Name Consumer

Intent To describe a specification where the state of `X`, which is used at a seed event `F(a = X)`, must be produced by an event `G(a = X)`.

Model specification

For all trace events matching

```
F(a = X)
```

with STM

```
F(use a)
G(def a)
and maybe others
```

a scenario matches

```
G(a = X); F(a = X) [seed]
```

Examples and known uses Consumer specifications are often used to model resources that must be allocated or reserved before they are used. For example, `X` could represent a lock, `G(a = X)` could represent acquiring `X`, and `F(a = X)` could represent releasing `X`.

Relationships This pattern is the converse of the Producer pattern.

B.0.2 The Producer pattern

Name Producer

Intent To describe a specification where the state of `X`, which is produced by a seed event `F(a = X)`, must be consumed by an event `G(a = X)`.

Model specification

For all trace events matching

```
F(a = X)
```

with STM

```
F(def a)
G(use a)
and maybe others
```

a scenario matches

```
F(a = X) [seed]; G(a = X)
```

Examples and known uses Producer specifications are often used to model resources that must be released after they are used. For example, `X` could represent a lock, `F(a = X)` could represent acquiring `X`, and `G(a = X)` could represent releasing `X`.

Relationships This pattern is the converse of the Consumer pattern.

B.0.3 The Union pattern

Name Union

Intent To describe a specification that is the union of a specification Spec_0 with another specification Spec_1, where both specifications have the same value flow model.

Model specification

For all trace events matching

$\text{SeedPattern}(\text{Spec}_0)|\text{SeedPattern}(\text{Spec}_1)$

with STM

$\text{VFM}(\text{Spec}_0)$, where $\text{VFM}(\text{Spec}_0) = \text{VFM}(\text{Spec}_1)$

a scenario matches

$\text{FA}(\text{Spec}_0)|\text{FA}(\text{Spec}_1)$

Examples and known uses This pattern can be used to construct complex specifications from specifications that follow simple patterns like producer and consumer. For example, the C standard library allows memory allocation either with `malloc` or `calloc`. The fact that memory must eventually be freed (by calling `free`) no matter which allocation call is used could be modeled with a union of two producer specifications. Also, the fact that all memory that is freed must have been allocated with either `malloc` or `calloc` could be modeled with a union of two consumer specifications.

Relationships None.

B.0.4 The MultiConsumer pattern

Name MultiConsumer

Intent To describe a specification where, given a seed event `F(a = X, b = Y)`,

- the state of `X`, which is consumed by `F`, must be produced by an event `G(a = X)`, and
- the state of `Y`, which is consumed by `F`, must be produced by an event `H(a = Y)`, and

Model specification

For all trace events matching

```
F(a = X, b = Y)
```

with STM

```
F(use a, use b)
G(def a)
H(def a)
```

a scenario matches

```
G(a = X); H(a = Y);
F(a = X, b = Y) [seed]
```

Examples and known uses This pattern is often used to describe binary operations whose operands must be of the same type. For example, the Quarks specification says that the `name` and `the_class` arguments of `XrmQGetResource` must be `Quarks`, which come from calls to `XrmPermStringToQuark`.

Relationships This pattern is one possible extension of the consumer pattern to two state variables.

B.0.5 The MultiConsumerAgrees pattern

Name MultiConsumerAgrees

Intent To describe a specification where, given a seed event `F(a = X, b = Y, c = Z)`,

- the state of `Y`, which is consumed by `F`, must be produced by an event `G(a = X, b = Y)`, and
- the state of `Z`, which is consumed by `F`, must be produced by an event `H(a = X, b = Z)`, and
- all three events use the same `X`.

Model specification

For all trace events matching

```
F(a = X, b = Y, c = Z)
```

with STM

```
F(use a, use b, use c)
G(use a, def b)
H(use a, def b)
```

a scenario matches

```
G(a = X, b = Y); H(a = X, b = Z);
F(a = X, b = Y, c = Z) [seed]
```

Examples and known uses This pattern is common in object-oriented interfaces, with `X` representing the receiver object (`this` in C++ parlance). For such interfaces, a MultiConsumerAgrees specification asserts that three messages must have the same receiver object. For example, the first argument to many X11 routines represents a connection to the X server; since a program may make more than one X server connection in its lifetime, it is sometimes necessary to assert that a sequence of calls uses the same connection.

Relationships This pattern is one possible extension of the consumer pattern to three state variables.

B.0.6 The TiedConsumer pattern

Name TiedConsumer

Intent To describe a specification where `X` and `Y`, which are consumed by a seed event `F(a = X, b = Y)`, must be produced simultaneously by some preceding event `G(a = X, b = Y)`.

Model specification

For all trace events matching

```
F(a = X, b = Y)
```

with STM

```
F(use a, use b)
G(def a, def b)
```

a scenario, in context, matches

```
(G(a = X, b = Y) |
 (.*; DEFINE(X); DEFINE(Y)))
F(a = X, b = Y) [seed];
```

Note that this model is conservative: it never rejects seed events that satisfy the pattern's intent, but it does accept some seed events that do not satisfy the intent. Consider this trace:

```
G(a = 10, b = 10); G(a = 20, b = 20); F(a = 10, b = 20);
```

The `F` event has this (named) scenario, in context:

```
DEFINE(X); DEFINE(Y); F(a = X, b = Y) [seed]
```

This scenario matches the expression in the model specification, but the `F` event does not satisfy the intent of the pattern.

In fact, it is impossible to model this pattern exactly in Strauss, because Strauss assumes that values are independent.

Examples and known uses This pattern models specifications where two arguments are considered as one logical argument. For example, in the X11

interface, the call `XSaveContext` adds a (key, value) pair to a hash table, which is maintained by the library. When it is done with the (key, value) pair, the application must call `XDeleteContext` with the same key.

Logically, the key should be one argument, but in fact the library forms the key by composing a `rid` argument (which gives a resource ID) with a `context` argument (which gives a context type). These arguments are passed to both `XSaveContext` and `XDeleteContext` and must agree.

Relationships This pattern is one possible extension of the Producer pattern to two state variables, and is the converse of the TiedProducer pattern.

B.0.7 The TiedProducer pattern

Name TiedProducer

Intent To describe a specification where `X` and `Y`, which are produced by a seed event `F(a = X, b = Y)`, must be consumed simultaneously by some following event `G(a = X, b = Y)`.

Model specification

For all trace events matching

```
F(a = X, b = Y)
```

with STM

```
F(def a, def b)
G(use a, use b)
```

a scenario, in context, matches

```
F(a = X, b = Y) [seed];
(G(a = X, b = Y) |
 (.*; USE(X); USE(Y)))
```

Note that this model is conservative: it never rejects seed events that satisfy the pattern's intent, but it does accept some seed events that do not satisfy the intent. Consider this trace:

```
F(a = 10, b = 20); G(a = 10, b = 10); G(a = 20, b = 20);
```

The `F` event has this (named) scenario, in context:

```
F(a = X, b = Y) [seed]; USE(X); USE(Y)
```

This scenario matches the expression in the model specification, but the `F` event does not satisfy the intent of the pattern.

In fact, it is impossible to model this pattern exactly in Strauss, because Strauss assumes that values are independent.

Examples and known uses This pattern models specifications where two arguments are considered as one logical argument. For example, in the X11

interface, the call `XSaveContext` adds a (key, value) pair to a hash table, which is maintained by the library. When it is done with the (key, value) pair, the application must call `XDeleteContext` with the same key.

Logically, the key should be one argument, but in fact the library forms the key by composing a `rid` argument (which gives a resource ID) with a `context` argument (which gives a context type). These arguments are passed to both `XSaveContext` and `XDeleteContext` and must agree.

Relationships This pattern is one possible extension of the producer pattern to two state variables, and is the converse of the TiedConsumer pattern.

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] Denholm, D. Xsetselectionowner bug in gnuplot-3.7.1, May 2003. Personal communication.

[14] 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.

[15] 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).

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

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

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

[27] 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.

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

[29] 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.

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] Mocenigo, J. Grappa: A Java graph package. URL: `http://www.research.att.com/~john/Grappa`.

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

[38] 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).

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

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

[41] 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.

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

[43] 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).

[44] 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.

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

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

[47] Squires, T. Xgetselectionowner bug in e93-1.3r2X, May 2003. Personal communication.

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

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

[50] 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.

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

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

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

[54] 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.

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