Mining Specifications

Glenn Ammons
Dept. of Computer Sciences
University of Wisconsin
Madison, Wisconsin, USA
ammons@cs.wisc.edu

Rastislav Bodík
Dept. of Computer Sciences
University of Wisconsin
Madison, Wisconsin, USA
bodik@cs.wisc.edu

James R. Larus
Microsoft Research
One Microsoft Way
Redmond, Washington, USA
larus@microsoft.com

ABSTRACT

Program verification is a promising approach to improving program quality, because it can search all possible program executions for specific errors. However, the need to formally describe correct behavior or errors is a major barrier to the widespread adoption of program verification, since programmers historically have been reluctant to write formal specifications. Automating the process of formulating specifications would remove a barrier to program verification and enhance its practicality.

This paper describes *specification mining*, a machine learning approach to discovering formal specifications of the protocols that code must obey when interacting with an application program interface or abstract data type. Starting from the assumption that a working program is well enough debugged to reveal strong hints of correct protocols, our tool infers a specification by observing program execution and concisely summarizing the frequent interaction patterns as state machines that capture both temporal and data dependences. These state machines can be examined by a programmer, to refine the specification and identify errors, and can be utilized by automatic verification tools, to find bugs.

Our preliminary experience with the mining tool has been promising. We were able to learn specifications that not only captured the correct protocol, but also discovered serious bugs.

1. INTRODUCTION

It is difficult to imagine software without bugs. The richness and variety of errors require an equally diverse set of techniques to avoid, detect, and correct them. Testing currently is the detection method of choice. However, the high cost and inherent limitations of testing has lead to a renewed interest in other approaches to finding bugs. One of the most promising directions is tools that systematically detect important classes of errors [1–3, 5, 6, 9, 10].

While *program verification* tools do not prevent programming errors, they can quickly and cheaply identify oversights early in the development process, when an error can be corrected by a programmer familiar with the code. Moreover, unlike testing, some verification tools can provide strong assurances that a program is free of a certain type of error.

These tools, in general, statically compute an approximation of a program's possible dynamic behaviors and compare it against a specification of correct behavior. These specifications often are easy to develop for language-specific properties—such as avoiding null dereferences and staying within array bounds. Even when language properties are more difficult to express and check, they potentially apply to every program written in the language, so an investment in a verification tool can be amortized easily.

On the other hand, specifications particular to a program, say of its abstractions or datatypes, may be difficult and expensive to develop because of the complexity of these mechanisms and the limited number of people who understand them. Also, as these specifications may apply to only one program, their benefits are correspondingly reduced. Program verification is unlikely to be widely used without cheaper and easier ways to formulate specifications.

This paper explores one approach to automating much of the process of producing specifications. This technique, called *specification mining*, discovers some of the temporal and data-dependence relationships that a program follows when it interacts with an application programming interface (API) or abstract datatype (ADT). A specification miner observes these interactions in a running program and uses this empirical data to infer a general rule about how programs should interact with the API or ADT. These rules are concisely summarized as state machines that capture both temporal and data dependences. These state machines can be both examined by a programmer, to refine the specification and identify errors, and utilized by automatic verification tools, to find bugs.

Mining proceeds under the assumption that an executing program, which presumably has passed some tests, generally uses an API or ADT correctly, so that if a miner can identify the common behavior, it can produce a correct specification, even from programs that contain errors. Rather than start from the program's text, in which feasible and infeasible paths are intermixed and correct paths are indistinguishable, mining begins with traces of a program's run-time interaction with an API or ADT. These traces are not only limited to feasible paths, but in general do not contain errors.

The program in Figure 1 illustrates these points. The program uses the server-side socket API [7]. It generally observes the correct protocol: create a new socket `s` through a call to `socket`, prepare `s` to accept connections by calling `bind` and `listen`, call `accept` for each connection, service each connection, and finally call `close` to destroy `s`. Unfortunately, the program is buggy: if the `return` statement on line 16 is executed, `s` is never closed.

Even though a program is buggy, individual interaction traces can be correct. Figure 2 shows one such trace. If `cond1` is rarely true, it might be difficult to invent a test to force the program to behave badly. On the other hand, correct traces enable a miner to

```
int s = socket(AF_INET, SOCK_STREAM, 0);
...
bind(s, &serv_addr, sizeof(serv_addr));
...
listen(s, 5);
...
while(1) {
  int ns = accept(s, &addr, &len);
  if (ns < 0) break;
  do {
     read(ns, buffer, 255);
     ...
     write(ns, buffer, size);
     if (cond1) return;
  } while (cond2)
  close(ns);
}
close(s);
```

Figure 1: An example program using the *socket* API.

```
socket(domain = 2, type = 1, proto = 0,
       return = 7)
bind(so = 7, addr = 0x400120, addr_len = 6,
     return = 0)
listen(so = 7, backlog = 5, return = 0)
accept(so = 7, addr = 0x400200,
       addr_len = 0x400240, return = 8)
read(fd = 8, buf = 0x400320, len = 255,
     return = 12)
write(fd = 8, buf = 0x400320, len = 12,
      return = 12)
read(fd = 8, buf = 0x400320, len = 255,
     return = 7)
write(fd = 8, buf = 0x400320, len = 7,
      return = 7)
close(fd = 8, return = 0)
accept(so = 7, addr = 0x400200,
       addr_len = 0x400240, return = 10)
read(fd = 10, buf = 0x400320, len = 255,
     return = 13)
write(fd = 10, buf = 0x400320, len = 13,
      return = 13)
close(fd = 10, return = 0)
close(fd = 7, return = 0)
```

Figure 2: Part of the input to our mining process: a trace of an execution of the program in Figure 1.

infer a specification of the correct protocol. A verification tool that uses the specification to examine all program paths (e.g. [2, 10]) could then find the rare bug.

Our specification mining system is composed of four parts: tracer, flow dependence annotator, scenario extractor, and automaton learner (Figure 4). The tracer instruments programs so that they trace and record their interactions with an API or ADT, as well as compute their usual results. We implemented two tracers. One is a replacement for the C stdio library, which requires recompiling programs. The other is a more general executable editing tool that allows arbitrary tracing code to be inserted at call sites. The tracers produce traces in a standard form, so that the rest of the process is independent of the tracing technology.

Flow dependence annotation is the first step in refining the traces into interaction scenarios, which can be fed to the learner. It connects an interaction that produces a value with the interactions that consume the value. Next, the scenario extractor uses these dependences to extract *interaction scenarios*—small sets of dependent

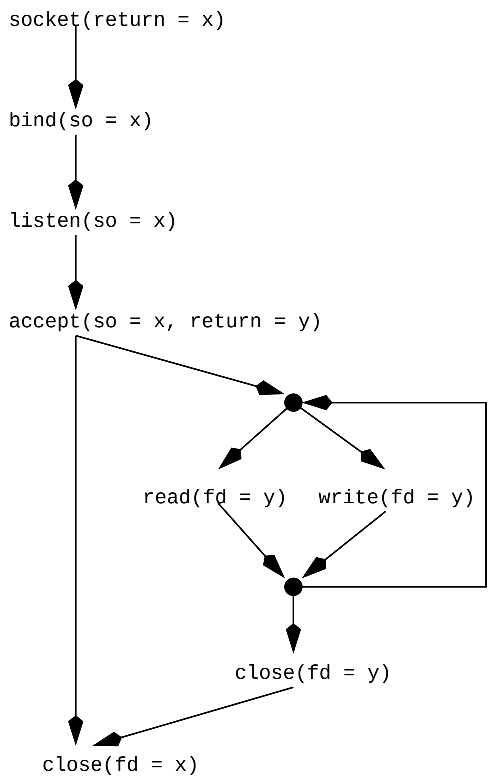


Figure 3: The output of our mining process: a specification automaton for the socket protocol.

interactions—and puts the scenarios into a standard, abstract form.

The automaton learner is composed of two parts: an off-the-shelf probablistic finite state automaton (PFSA) learner and a postprocessor called the *corer*. The PFSA learner analyzes the scenario strings and generates a PFSA, which should be both small and likely to generate the scenarios. A PFSA is a nondeterministic finite automaton (NFA) in which each edge is labelled by an abstract interaction and weighted by how often the edge is traversed while generating or accepting scenario strings. Rarely-used edges correspond to infrequent behavior, so the corer removes them. The corer also discards the weights, leaving an NFA. A human can validate the NFA by inspection, at which point the NFA specification can be used for program verification. Figure 1 shows a specification for the socket protocol of the program in Figure 1.

This paper makes the following contributions:

- We introduce a new approach, called *specifications mining*, for learning formal correctness specifications. Since specification is the portion of program verification still dependent primarily on people, automating this step can improve the appeal of verification and help improve software quality.

- We use the observation that *common behavior is often correct behavior* to refine the specifications mining problem into a problem of probabilistic learning from execution traces.

- We develop and demonstrate a practical technique for probabilistic learning from execution traces. Our technique reduces specification mining to the problem of learning regular languages, for which off-the-shelf learners exist.

The rest of the paper is organized as follows. Section 2 develops

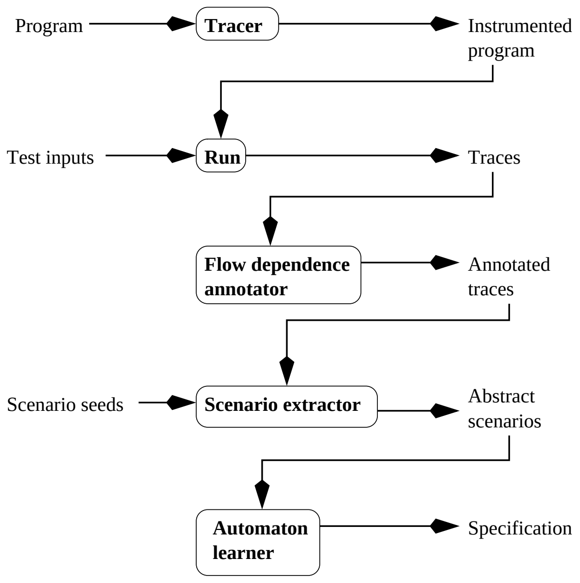


Figure 4: Overview of our specification mining system.

a formal statement of the specification mining problem. Section 3 discusses tracers and the flow dependence annotator. Section 4 describes the scenario extractor and the automaton learner. Section 5 presents a dynamic checker for the mined specifications. Section 6 discusses the results of some experiments with the mining tool and the checker. Section 7 discusses related work and Section 8 concludes the paper.

2. THE PROBLEM

This section develops a formal statement of the specification mining problem. At its most ambitious, specification mining attempts to solve an unsolvable problem:

PROBLEM 2.1. *Let I be the set of all traces of interactions with an API or ADT, and $C \subseteq I$ be the set of all correct traces of interactions with the API or ADT. Given an unlabelled training set T of interaction traces*[1] *from I, find an automaton A that generates exactly the traces in C. A is called a* specification. *An algorithm that finds A is called a* specification miner.

The rest of this section examines successive restrictions of this problem, which lead to a problem that can be attacked with the methods of this paper. These simplifications were choosen to accomodate our techniques, and other restatements of the specification mining problem are certainly possible.

Problem 2.1 can not be solved because it places no restrictions on the set C. If C is not recursively enumerable, then A does not exist. In this paper, we require that C be generated by a finite-state automaton (that is, C is a regular language). This decision is forced by two practical considerations. First, model checkers require finite-state specifications. Second, the algorithms for learning finite-state automata are relatively well-developed.

It is not enough to simply say that C must be regular, because traces of most programs are not regular. For example, consider a C program (`LinkedList`) that takes a number n on the command line, constructs a linked list of size n (allocating the nodes with `malloc`), and then destroys the linked list, deallocating the nodes with `free` in first-allocated, last-freed order. Ignoring the finite arithmetic, the traces do not form a regular language. First, a regular language must be defined over a finite alphabet, but `LinkedList` can make an unbounded number of distinct `malloc` and `free` calls. Second, `LinkedList` always makes a number of `malloc` calls followed by an equal number of `free` calls, which is the canonical non-regular language.

[1]By an "unlabelled training set", we mean that no information is provided as to which of the elements in T are also in C.

Although `LinkedList`'s traces do not form a regular language, its traces contain subtraces that do. Given a trace and an object o mentioned in that trace, consider the subtrace of the trace containing calls to `malloc` that return o and calls to `free` that are passed o. The subtrace is simply a `malloc` call, followed by a `free` call. If the trace mentions n objects, there is one such subtrace for each object. Each subtrace is exactly like all of the others, except for the particular object that it allocates and frees. Now replace that object in each subtrace with a standard name, say o_{std}. Now, all of the subtraces are identical, and the learner has a very strong hint that `free` should always follow `malloc`. We call the renamed subtraces *interaction scenarios*.

Our approach simplifies Problem 2.1 in two ways. First, the learner does not learn directly from traces. Instead, a preprocessor extracts interaction scenarios from the traces. The scenarios manipulate no more than k data objects, for some k; in the `LinkedList` example, $k = 1$. Second, the set C_S of correct scenarios is required to be regular. The simplified specification mining problem can now be defined:

PROBLEM 2.2. *Let I_S be the set of all interaction scenarios with an API or ADT that manipulate no more than k data objects. Let $C_S \subseteq I_S$ be the regular set of all correct scenarios. Given an unlabelled training set T_S of interaction scenarios from I_S, find a finite-state automaton A_S that generates exactly the scenarios in C_S.*

Problem 2.2 is also impossible to solve. The careful reader may have noticed that the training set T_S does not depend on C_S. That is, no matter what C_S is, any subset of I_S is a valid training set! Obviously, under these conditions, there is no basis on which to choose S_S. The definition of Problem 2.2 allowed the training sets to be chosen so liberally in order to allow "noisy" training sets that contain bad examples (that is, bugs) that are not in C_S. A satisfactory definition of noise must wait until the problem has been simplified further. For now, we simplify the problem by assuming that the training "set" is in fact an infinite sequence of scenarios from C_S alone, such that each element of C_S occurs at least once:

PROBLEM 2.3. *Let I_S be the set of all interaction scenarios with an API or ADT that manipulate no more than k data objects. Let $C_S \subseteq I_S$ be the regular set of all such correct scenarios. Finally, let $T_S = c_0, c_1, \ldots$ be an infinite sequence of elements from C_S in which each element of C_S occurs at least once.*

For each $n > 0$, examine the first n elements of T_S and produce a finite-state automaton A_{S_n}, such that the sequence of finite-state automata $A_{S_0}, A_{S_1}, \ldots$ has this property: for some $N \geq 0$, A_{S_N} generates exactly the scenarios in C_S and $A_{S_n} = A_{S_N}$ for all $n \geq N$. We say that the sequence $A_{S_0}, A_{S_1}, \ldots$ identifies *C_S* in the limit.

Perhaps surprisingly, Problem 2.3 is also undecidable. Our definition of Problem 2.3 is inspired by E Mark Gold's seminal paper on language identification in the limit [14], in which Gold shows that regular languages can not be identified in the limit [14, Theorem I.8]. His proof is too long to repeat here, but the idea of the proof is to present the members of an infinite regular language to the learner in such a way that the learner is forced to change its guess infinitely often, cycling through a never ending sequence of finite sublanguages of the infinite language. Intuitively, the learner's dilemma is that any finite sequence of examples from the infinite language is also a sequence of examples from a finite language, and the learner has no basis for preferring one of these over the other. Since C_S is a possibly infinite regular language, Gold's theorem applies to Problem 2.3.

Gold's paper did not end work on learning regular languages from examples. Subsequent work avoids the dilemma exploited in Gold's proof by providing the learner with extra information that allows it to justify choosing a less general automaton over a more general one (and vice versa). One class of approaches presents the learner with examples generated according to a probability distribution; this sort of approach is particularly interesting to us because it also gives the learner a method for dealing with noise in its input. The task of the learner is to learn a close approximation of the probability distribution:

> *Let I_S be the set of all interaction scenarios with an API or ADT that manipulate no more than k data objects. Let P and $\widehat{P}$ be probability distributions over I_S. We say that $\widehat{P}$ is an ϵ-*good *approximation of P, for $\epsilon \geq 0$, if*
>
> $$D(P, \widehat{P}) \leq \epsilon$$
>
> *where $D(P, \widehat{P})$ is some measure of distance between P and $\widehat{P}$.*

Just as Problem 2.2 restricted C_S to be a regular set, P must be restricted to a manageable class of distributions. We choose the distributions generated by probabilistic finite state automata (PFSAs). A PFSA is a probabilistic analogue of a nondeterministic finite state automaton. That is, a PFSA is a tuple (Σ, Q, q_s, q_f, p) where

- Σ is an output 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.
- $p(q, q', a)$ is a probability function, giving the probability of transitioning from $q \in Q$ to $q' \in Q$ while outputting the symbol $a \in \Sigma$. Note that $p(q_f, q', a) = 0$ for all $q \in Q$ and $a \in \Sigma$.

Thus, a PFSA generates a distribution that assigns positive probabilities to the strings in a regular language. Basing our definition on the standard definition for learning probabilistic finite automata [20], we can now give our final formulation of the specification mining problem:

PROBLEM 2.4. *Let I_S be the set of all interaction scenarios with an API or ADT that manipulate no more than k data objects. Let M be a target PFSA, and P^M be the distribution over I_S that M generates. Intuitively, P^M assigns high probabilities to correct traces and low probabilities to incorrect traces.*

Given a confidence parameter $\delta > 0$ and an approximation parameter $\epsilon > 0$, efficiently find with probability at least $1 - \delta$ a PFSA $\widehat{M}$ such that its distribution $P^{\widehat{M}}$ is an ϵ-good approximation of P^M. "Efficiently" means that the mining algorithm must run in time polynomial in $1/\epsilon$, $1/\delta$, an upper bound n on the number of states of M, and the size of the alphabet Σ of I.

```
int instrumented_socket(int do-
main, int type, int proto)
{
  int rc = socket(domain, type, proto);
  fprintf(the_trace_fp,
          "socket(domain = %d, type = %d, "
          "proto = %d, return = %d)\n",
          domain, type, proto, rc);
  return rc;
}
```

Figure 5: Illustration of trace instrumentation (instrumented version of `socket`).

Unfortunately, with reasonable distance metrics D, it has been shown that Problem 2.4 is not efficiently learnable [16]. An efficient solution has been found for the case where M and $\widehat{M}$ are required to be acyclic and deterministic [24]. Since many interesting specifications of program behavior contain loops, we chose to use a greedy PFSA learning algorithm that is not guaranteed to find an ϵ-good approximation of M, but in practice generates succinct specifications.

3. TRACING AND FLOW DEPENDENCE ANNOTATION

This section describes the tracing and flow dependence annotation that produce the input to the scenario extractor.

Tracing A tracer instruments a program, so that running it produces a trace of its interactions with an API or ADT, as well as its usual results. This paper assumes that a tracer only records function calls and returns, although depending on the API/ADT, the mining system allows tracing other events, such as variable accesses or network messages.

Figure 5 shows an illustration of the trace instrumentation, specifically the C code for an instrumented version of the `socket` call. This wrapper calls the real `socket` and records information about the interaction: the name of the call (`socket`), arguments, and return value. The entire socket API could be traced with an instrumented version of each function.

Our system currently uses two tracers. The first instruments the C stdio library, by capturing all library calls and macro invocations in that API. The second consists of two parts: Perl scripts that automatically generate instrumented versions of the function calls in the X11 API, and a tool that edits program executables to replace calls on these routines with calls to instrumented versions. The latter tool is based on the EEL Executable Editing Library [17] and is very general. It takes as input an executable, a library of instrumented functions, and a file specifying which calls in the executable to replace with calls to instrumented functions. The most time-consuming part of tracing an interface is writing the instrumented version of each API call, but we believe that this step is easily automated.

All tracers record interactions in the same format, so that the rest of the mining system is independent of the particular tracer used. An *interaction skeleton* is of the form

$$\mathit{interaction}(\mathit{attribute}_0, \ldots, \mathit{attribute}_n)$$

where $\mathit{interaction}$ names the interaction (that is, the name of a function) and $\mathit{attribute}_i$ names the ith attribute of the interaction. Skeletons are just a convenient way of grouping interactions. They do not appear in traces. An interaction instantiates a skeleton by assigning values to the attributes:

$$interaction(attribute_0 = v_0, \ldots, attribute_n = v_n)$$

When tracing function calls, interaction attributes usually represent function arguments and return values, as in Figure 2. Structured data can be represented by flattening the structures. For example, given this C code

```
struct S { int x; int y; };
void f(S* s);
```

the tracer could record interactions with `f` with instances of this skeleton

```
f(S, S_x, S_y)
```

By convention, this paper names traces with the letter T and interactions with variations of the letter t. The actual interactions in a trace of length $n + 1$ are numbered from 0 to n; for example, $T = t_0, \ldots, t_n$. The notation $t.a$ denotes the a attribute of the interaction t.

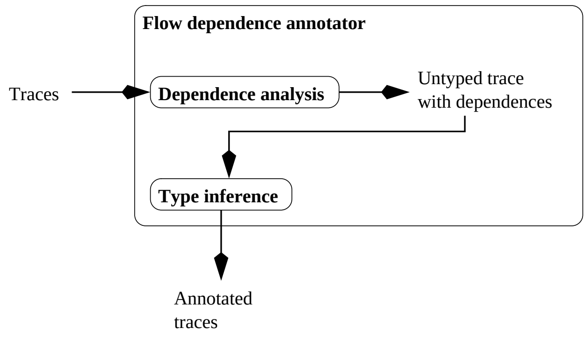


Figure 6: Detailed view of the flow dependence annotator.

Flow dependence annotation Flow dependence annotation annotates each input trace with flow dependences and type assignments. The scenario extractor uses these annotations to extract scenarios—small sets of dependent interactions—from the trace and to put each scenario into a canonical form. The detailed view in Figure 6 shows that flow dependence annotation is a two-step process. First, dependence analysis marks the trace with flow dependences, which constrain how interactions may be reordered and identify related interactions that could be grouped into a scenario for the automaton learner. Next, type inference assigns a type to each interaction attribute in the trace. The scenario extractor uses the types to avoid naming conflicts when it puts a scenario into standard form. Dependence analysis and type inference both examine the entirety of each input trace, so their running time must be nearly linear.

The miner treats all values as abstract objects whose underlying representation is unknown. However, interactions can depend on results from other interactions. For example, in Figure 2, the `bind` call (line 2) depends on the `socket` call (line 1), because the `bind` call uses file descriptor 7 returned by the `socket` call. The order of these two interactions can not be reversed. By contrast, the interactions that manipulate file descriptor 8 (lines 4–9) could be exchanged with the interactions that manipulate file descriptor 10 (lines 10–13), since these groups of operations are independent of each other. More importantly, a scenario that contains all interactions related to the `close` on line 13 should not include the interactions on lines 4–9.

Definers:	`socket.return`
	`bind.so`
	`listen.so`
	`accept.return`
	`close.fd`
Users:	`bind.so`
	`listen.so`
	`accept.so`
	`read.fd`
	`write.fd`
	`close.fd`

Figure 7: Attributes of socket interactions that define and use their values.

Flow dependences connect attributes that change the state of an abstract object (that is, attributes that *define* the object) to interaction attributes that depend on the state of an abstract object (that is, attributes that *use* the object). Ideally, the dependence analyzer would annotate a trace with flow dependences using no information beyond the trace itself. Our current system, however, relies on an expert to tell the analyzer which attributes of interactions may define objects, and which attributes may use objects. This work must be done once for each API/ADT. Extending the system to infer the sets of definers and users automatically is future work.

For simplicity, the examples in this paper assume that only socket-valued attributes of the interactions in Figure 2 carry dependences. Figure 7 lists attributes of interactions in Figure 2 that define and use socket values. We constructed this table as follows. For each socket, the kernel maintains a hidden data structure. Some of the fields of that structure carry the state of the socket: whether the socket is closed or open, whether or not it can accept connections, and so on. Other fields simply hold data: bytes that are outstanding, the port to which the socket is connected, and so on. Definers in Figure 2 typically modify one or more of the state fields of the data structure. Users typically read one or more of those fields. Fields of the structure that merely hold data are ignored.

Creating Figure 7 required expert knowledge. However, note that whenever the state fields of a socket's data structure change, the set of API calls that may follow also changes. For example, after a socket is closed, `read` and `write` calls are no longer allowed. The fact that `close` changes the state of the socket can be inferred from the trace: before a `close`, there are `read`s and `write`s; after a `close`, there are no `read`s and `write`s. That is, interactions that change state also change the sorts of interactions that may follow. In future work, we hope to replace the expert with an automatic tool that uses this fact to infer the sets of definers and users.

Given the lists of attributes that define or use objects, dependence analysis is a dynamic version of the reaching definitions problem. The analyzer traverses the trace $T = t_0, \ldots, t_n$ in order from t_0 to t_n, maintaining a table M that maps values to attributes of actual interactions. Initially, M is empty. If $t_i.a$ defines an object o, the annotator updates M to map o to $t_i.a$. If $t_i.a$ uses an object o and M maps o to $t_{i'}.a'$, then the analyzer places a flow dependence from $t_i.a$ to $t_{i'}.a'$. The running time of the algorithm scales linearly in the length of the trace. The space required scales linearly in the number of different values referenced by the trace.

For notational convenience, we introduce a relation d_f such that $d_f(t_i.a, t_{i'}.a')$ if and only if there is a flow dependence from $t_i.a$ to $t_{i'}.a'$. The relation is extended from interaction attributes to interactions in the natural way: $d_f(t_i, t_{i'})$ holds if and only if there

```
Type(socket.return) = T0
Type(bind.so) = T0
Type(listen.so) = T0
Type(accept.so) = T0
Type(accept.return) = T0
Type(read.fd) = T0
Type(write.fd) = T0
Type(close.fd) = T0
```

Figure 8: The only valid typing for the skeleton attributes used by the trace in Figure 2.

is some f and f' such that $d_f(t_i.a, t_{i'}.a')$.

Type inference is the next step in the flow dependence annotator. Type inference assigns a type to each skeleton attribute that is involved in dependences. If a value never flows between an instance of one skeleton attribute and an instance of another skeleton attribute, then type inference assigns the skeleton attributes separate types. Strictly speaking, flow dependences alone give the scenario extractor enough information to extract scenarios and put them into a standard form. However, as Section 4 explains, the scenario extractor can use the assurance that values will never flow between certain attributes in a scenario to reduce naming conflicts. Type inference infers a typing that satisfies this condition:

> If $d_f(t_i.a, t_{i'}.a')$, then the typing gives the skeleton attribute of $t_i.a$ and the skeleton attribute of $t_{i'}.a$ the same type.

Figure 8 gives a typing for the skeleton attributes used by the socket trace in Figure 2. In this example, every skeleton attribute must have the same type because all socket attributes in the trace are on some dependence chain with an instance of a `close.fd` attribute.

The inference algorithm uses Tarjan's union-find algorithm [26] and requires time nearly linear in the trace. The type inferer starts with an initial typing that gives each skeleton attribute its own unique type. Then, the inferer visits each dependence $d_f(t_i.a, t_{i'}.a')$ and unifies the types of the skeleton attribute of $t_i.a$ and the skeleton attribute of $t_{i'}.a$. Type inference is complete when all dependences have been visited.

4. SCENARIO EXTRACTION AND AUTOMATON LEARNING

This section explains how the scenario extractor and automaton learner work. The first tool extracts interaction scenarios—small sets of interdependent interactions—from annotated traces and prepares them for the automaton learner. The second tool infers specifications from scenarios, not complete traces, for two reasons.

The primary reason is that scenarios are much shorter than traces and the running time of our PFSA learner increases as the third power of the length of its input—this is typical for automaton learners.

Also, we restrict scenarios to refer to a small number of objects by bounding the size of the scenario. Section 2 argues that bounding the number of objects makes specification mining tractable. Bounding the number of objects is not a severe limitation because verification tools can verify that the specification holds for multiple bindings of program objects to specification objects. For example, although the protocol specified in Figure 1 mentions two objects, x and y, a tool that attempts to verify the program in Figure 1 might bind y to more than one instance of `ns` as it simulates the loop in lines 7–17.

The scenario extractor simplifies and standardizes the scenarios before passing them to the automaton learner, because our specification mining system uses an off-the-shelf PFSA learner. An alternative, which we have not tried, is to design a special-purpose learner for scenarios. Both schemes have benefits and costs.

There are several off-the-shelf learners that learn PFSAs and similar automata from strings. Since our design transforms scenarios to strings, a new learner can be substituted for the learner currently used. If the new learner learns PFSAs, no changes to the mining system are necessary. If the learner does not learn PFSAs, the corer may need to be changed, but none of the components before the automaton learner in Figure 4 would require modification. In our experience, this flexible design was helpful. Before settling on the PFSA learner as use it now, we tried and rejected one other PFSA learner [21].

On the other hand, a special-purpose learner could defer decisions that our mining system now must make before invoking the off-the-shelf learner. For example, when the scenario extractor replaces the concrete values in a scenario with abstract names, it does so without regard to the names given to values in other scenarios. Although the extractor always names equivalent scenarios in the exact same way (see below for details), when two scenarios are "close" but not equivalent, the extractor's choice of names can prevent the PFSA learner from merging states that it would be able to merge with a different naming.

4.1 Scenario extraction

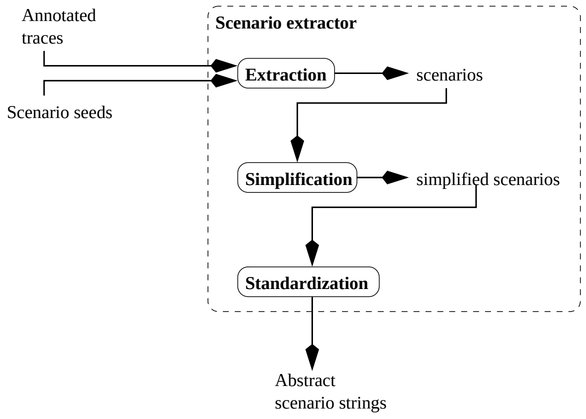


Figure 9: Detailed view of the scenario extractor.

Figure 9 is a detailed view of the scenario extractor. It receives two inputs. The first is a set of traces, annotated as described in Section 3. In addition, the user controls which scenarios will be extracted by supplying a set of scenario seeds. Each seed is an interaction skeleton. The extractor searches the input traces for interactions that match the seeds and extracts a scenario from each interaction. For example, suppose the extractor was given the trace of socket interactions in Figure 2 and `accept(so, return)` as the seed. The extractor would produce two scenarios, one around the `accept` on line 4 and the other around the `accept` on line 10.

Extraction Producing scenarios from input traces is the first step of the extraction process. Informally, a scenario is a set of interactions related by flow dependences. Formally, given an annotated trace

```
1 socket(domain = 2, type = 1, proto = 0,
         return = 7)
2 bind(so = 7, addr = 0x400120, addr_len = 6,
       return = 0)
3 listen(so = 7, backlog = 5, return = 0)
4 accept(so = 7, addr = 0x400200,
         addr_len = 0x400240,
         return = 8) [seed]
5 read(fd = 8, buf = 0x400320, len = 255,
       return = 12)
6 write(fd = 8, buf = 0x400320, len = 12,
        return = 12)
7 read(fd = 8, buf = 0x400320, len = 255,
       return = 7)
8 write(fd = 8, buf = 0x400320, len = 7,
        return = 7)
9 close(fd = 8, return = 0)
```

Figure 10: A scenario extracted from around line 4 of Figure 2, with $N = 10$

$T = t_0, \dots, t_n$, a scenario is a set $S \subseteq \{t_0, \dots, t_n\}$ with the property:

> If $t_{i_0} \in S$, $t_{i_n} \in S$, and $t_{i_0}, \dots, t_{i_n}$ is a chain of flow dependent interactions in T, then $t_{i_j} \in S$ for any $0 \le j \le n$.

The extractor builds a scenario around each interaction in the trace that matches a scenario seed. For any scenario S, $seed(S) \in S$ is the interaction that initially matches the seed.

A user-tunable parameter N restricts the number of interactions in the extracted scenarios. Each scenario contains at most N ancestors and at most N descendants of the seed interaction. The extractor prefers ancestors and descendants whose position in the input trace is close to the position of seed interaction.

Once an interaction t_s matching a seed is found, the extractor uses a two-step algorithm to produce a scenario. First, the extractor constructs the sets:

$$\begin{aligned} S_a &= \{N \text{ closest ancestors of } t_s\} \\ S_d &= \{N \text{ closest descendants of } t_s\} \\ S_{ad} &= \{t_s\} \cup S_a \cup S_d \end{aligned}$$

The extractor uses a simple prioritized worklist algorithm to construct the set of ancestors (descendants). The initial worklist is the set of immediate ancestors (descendants) of t_s. Repeatedly, until the worklist is empty or N ancestors (descendants) are found, the extractor removes from the worklist the ancestor (descendant) whose position in the trace is nearest t_s, adds it to the set of ancestors (descendants), and adds its immediate ancestors (descendants) to the worklist.

The result, S_{ad}, is not necessarily a scenario, because interactions along some flow dependence chains from ancestors of t_s to descendants of t_s might be missing. Any such interactions must lie in the trace between the earliest ancestor t_a in S_{ad} and the latest descendant t_d in S_{ad}, and must be reachable both by following flow dependences from some ancestor of t_s and by following flow dependences in reverse from some descendant of t_s. Thus, the extractor searches depth-first forwards from each element of S_a and backwards from each element of S_d to construct

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

Figure 11: The simplification of the scenario in Figure 10.

```
1  socket(return = x0:T0)                            (A)
2  bind(so = x0:T0)                                  (B)
3  listen(so = x0:T0)                                (C)
4  accept(so = x0:T0, return = x1:T0) [seed]         (D)
5  read(fd = x1:T0)                                  (E)
7  read(fd = x1:T0)                                  (E)
6  write(fd = x1:T0)                                 (F)
8  write(fd = x1:T0)                                 (F)
9  close(fd = x1:T0)                                 (G)
```

Figure 12: Scenario string for the simplified scenario from Figure 11.

$$\begin{aligned} S_{ar} &= \{t \in [t_a, t_d] \mid \exists t' \in S_a . t' \text{ reaches } t\} \\ S_{dr} &= \{t \in [t_a, t_d] \mid \exists t' \in S_d . t' \text{ reachesinreverse } t\} \end{aligned}$$

The final scenario is $S = S_{ad} \cup (S_{ar} \cap S_{dr})$. Figure 10 shows a scenario extracted from the trace in Figure 2 with $N = 10$, around the `accept` on line 4. The seed is marked. Also note that the interactions in S inherit the dependences from the annotated trace.

Simplification Given the extracted scenarios, simplification eliminates all interaction attributes that do not carry a flow dependence in any training traces. The typing inferred by the dependence annotator (see Section 3) assigns a type to an skeleton attribute if and only if an instance of that attribute is involved in a flow dependence somewhere in a trace. So, simplification preserves an interaction attribute if and only if the corresponding skeleton attribute is typed. Figure 11 is the simplified version of the scenario in Figure 10.

Standardization Standardization converts a scenario into a scenario string for the PFSA learner. Standardization improves the performance of the PFSA learner by producing scenario strings so that similar scenarios receive similar strings.

Figure 12 shows the result of standardizing the scenario in Figure 11. Standardization applies two transformations: naming and reordering.

Naming replaces attribute values with symbolic variables. In Figure 12, value 7 is replaced with the symbolic name `x0:T0`, and value 8 is replaced with the symbolic name `x1:T0`. Naming exposes similarities between different scenarios by naming flow dependences. For example, a scenario extracted around line 10 of Figure 2 manipulates different socket values (7 and 10 instead of 7 and 8), but naming still calls one of these values `x0:T0` and the other `x1:T0`.

When a value flows from one attribute to another, naming indicates the dependence by assigning the same name to both attributes. The dependence annotation typing (section 3) guarantees that, if two skeleton attributes are assigned different types, values never flow between instances of those attributes. Thus, naming uses a separate namespace for attributes of each type. Figure 13 illus-

```
Original  S0                         S1
     1    A(x=0, y=0) [seed]         E(x=0, v=1) [seed]
     2    B(x=0, y=0)                B(x=0, y=0)
     3    C(x=0, y=0)                C(x=0, y=0)

Untyped
     1    A(x=x0, y=x1)              E(x=x0, v=x1)
     2    B(x=x0, y=x1)              B(x=x0, y=x2)
     3    C(x=x0, y=x1)              C(x=x0, y=x2)

 Typed
     1    A(x=x0:T0, y=x0:T1)  E(x=x0:T0, v=x0:T2)
     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 13: Two nearly equivalent scenarios and their scenario strings, with untyped and typed naming.

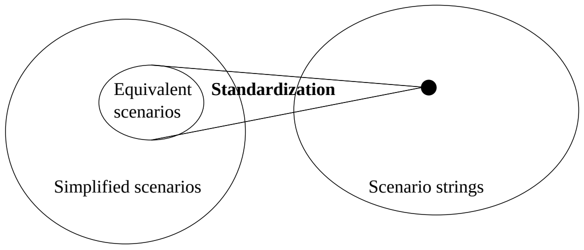


Figure 14: Standardization, as a many-to-one mapping.

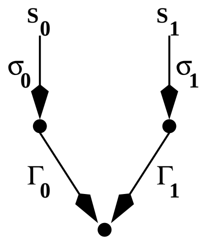
trates how separate namespaces help expose more similarities to the PFSA learner. Lines 2 and 3 of S_0 and S_1 are the same, but line 1 differs in each scenario. Assume that naming assigns names to each interaction in turn, starting at the seed interaction. Without types, naming treats lines 2 and 3 differently.

Reordering standardizes the order of scenario interactions. A scenario contains interactions that are partially ordered by flow, anti, and output dependences. That is, each scenario corresponds to a directed acyclic graph (DAG). The order in which the interactions appear in the original traces is just one legal total order. Reordering puts two scenarios with the same DAG into the same total order, even when their trace order differs, so that a PFSA learner is presented with fewer distinct strings. In Figure 12, reordering swapped the `write` on line 6 with the `read` on line 7.

To a PFSA learner, each interaction in a scenario string is merely an atomic letter. To emphasize this point, the right-hand side of Figure 12 replaces each interaction with a shorthand letter. Standardization uses a small number of letters to represent a given set of scenarios. Using a small alphabet increases the PFSA learner's opportunities to find similarities in the scenario strings. Also, PFSA learners run more slowly with large alphabets.

The rest of this section discusses our standardization algorithm in detail. At a high level, standardization is a many-to-one mapping from simplified scenarios to scenario strings (Figure 14). Under this mapping, the preimage of a scenario string is a set of *equivalent* scenarios. Intuitively, equivalent scenarios manipulate abstract objects in the same way. In the following, we define equivalence, present our standardization algorithm, and show that equivalence characterizes the scenarios that standardization maps to the same scenario string.

Let $S = s_0, \ldots, s_n$ be a simplified scenario. A *dependence-preserving permutation* of S is a permutation σ of S such that if $d(s_i, s_{i'})$, then $\sigma(i) < \sigma(i')$. That is, the permutation does not



$$\begin{tikzcd}
S_0 \arrow[d, "\sigma_0"'] & & S_1 \arrow[d, "\sigma_1"] \\
\bullet \arrow[dr, "\Gamma_0"'] & & \bullet \arrow[dl, "\Gamma_1"] \\
& \bullet &
\end{tikzcd}$$

Figure 15: Equivalent scenarios.

```
Naive($S$)
    MaxSize := maximum size of a scenario
    $X$ := a totally ordered set of MaxSize symbolic names
    AllStrings := $\emptyset$
    Permutes := all dependence-preserving permutations of $S$
    Foreach $\sigma \in$ Permutes
        Namings := all dependence-preserving namings of $\sigma(S)$ from $X$
        Foreach $\Gamma \in$ Namings
            Add $\Gamma(\sigma(S))$ to AllStrings
    Return the lexicographically smallest element of AllStrings
```

Figure 16: Naive standardization algorithm.

swap the source and sink of any dependence. Figure 12 illustrates a dependence-preserving permutation that swaps the `read` on line 6 with the `write` on line 7.

A *naming* Γ of S replaces each value in S with a symbolic name, taken from a set X. If $s_i.a$ is an attribute in S, $\Gamma(s_i.a)$ is the symbolic name given to that attribute in $\Gamma(S)$. We say that Γ is *dependence-preserving* if, for any $s_i.a$ and $s_{i'}.a'$, $d(s_i.a, s_{i'}.a') \equiv \Gamma(s_i.a) = \Gamma(s_i.a')$.

Now let $S_0 = s_{0,0}, \ldots, s_{0,n}$ and $S_1 = s_{1,0}, \ldots, s_{1,n}$ be two simplified scenarios. S_0 and S_1 are *equivalent* iff there are dependence-preserving permutations σ_0 of S_0 and σ_1 of S_1 and dependence-preserving namings Γ_0 of $\sigma_0(S_0)$ and Γ_1 of $\sigma_1(S_1)$ such that $\Gamma_0(\sigma_0(S_0)) = \Gamma_1(\sigma_1(S_1))$ (Figure 15). In fact, the choice of σ_0 and Γ_0 is not important. We assert that if S_0 and S_1 are equivalent, then for any dependence-preserving σ_0 of S_0 and dependence-preserving Γ_0 of $\sigma_0(S_0)$, there is a dependence-preserving σ_1 of S_1 and a dependence-preserving Γ_1 of $\sigma_1(\Gamma_1)$ such that $\Gamma_0(\sigma_0(S_0)) = \Gamma_1(\sigma_1(S_1))$.

Figure 16 presents a naive standardization algorithm. `Naive` tries all dependence-preserving permutations of S and all dependence-preserving namings of each permutation and returns the scenario string that comes first in lexicographic order. If `Naive` assigns S_0 and S_1 the same scenario string, then S_0 and S_1 are equivalent, since the algorithm has found permutations and namings that make them equal. And, if S_0 and S_1 are equivalent, then `Naive` generates the same `AllStrings` set for both of them. So, equivalence characterizes the preimage of `Naive`, as promised. However, the running time of the algorithm is exponential in $|X|$ and $|S|$.

The algorithm in Figure 17 removes the exponential behavior in $|X|$ by considering only one standard naming for each permuted scenario. This optimization is safe because if S_0 and S_1 are equivalent up to a dependence-preserving naming, then they differ only in their values, and `StandardName` does not depend on the identities of the values at attributes, but only on their types and the dependences that they carry.

`StandardName` draws names from separate name spaces for separate types. The `GetNextName` operation returns the next

```
NameInteraction($s$)
  Foreach attribute $s.a$
    Type := the type of $s.a$'s skeleton attribute
    Value := the value at $s.a$
    NameSpace := name space for Type
    If NameSpace[Value] has not been set
      NameSpace[Value] := GetNextName(NameSpace)
    Replace Value with NameSpace[Value] in $s.a$

StandardName($S = s_0, \ldots, s_n$)
  $i_s$ := index of the seed in $S$
  NameInteraction($s_{i_s}$)
  dist := 1
  While $i_s - \texttt{dist} \geq 0$ or $i_s + \texttt{dist} \leq n$
    If $i_s - \texttt{dist} \geq 0$ NameInteraction($s_{i_s - \texttt{dist}}$)
    If $i_s + \texttt{dist} \leq n$ NameInteraction($s_{i_s + \texttt{dist}}$)
    dist := $\texttt{dist} + 1$

Better($S$)
  Reset all name spaces
  AllStrings := $\emptyset$
  Permutes := all dependence-preserving permutations of $S$
  Foreach $\sigma \in$ Permutes
    $S_{named}$ := StandardName($\sigma(S)$)
    Add $S_{named}$ to AllStrings
  Return the lexicographically smallest element of AllStrings
```

Figure 17: Better standardization algorithm.

available name in a name space in some fixed order, and resetting a name space causes it to begin again with the first name in the space. StandardName names the seed interaction first, and then works outward. Because the name of a value must be chosen consistently, the constraints on naming increase as interactions are named. Interactions near seeds are most likely to be similar across scenarios, so they are named first.

The worst-case running time of Better is still exponential in $|S|$. We can not expect to do better in the worst-case, because Better can be used to solve the DAG-isomorphism problem by encoding arbitrary DAGs as scenarios, and DAG-isomorphism is NP-complete. However, better performance is possible in the common case, since trace scenarios are not arbitrary DAGs. In particular, the interactions in a scenario have names and named attributes. Our final standardization algorithm (Figure 18) uses those names to reduce the number of permutations it considers.

Standardize considers only dependence-preserving permutations that put the skeletons of the interactions in the smallest possible lexicographic order. Although there can be an exponential number of such orderings, there is often only one. In that case, the set of interactions Selected (line *) always has one element, and the recursion never branches. With an appropriate implementation of OfLeastKinds (which sorts the interactions), the algorithm runs in that case in time proportional to $n \log n$. In our experience, the time spent running Standardize is an insignificant part of the scenario extraction time.

4.2 Automaton learning

This section presents the algorithms and data structures used in learning the specification automaton. The automaton A is an NFA with edges labelled by standardized interactions, whose language includes the most common substrings of the scenario strings extracted from the training traces, plus other strings that the PFSA learner adds as it generalizes. Automaton learning has two steps. First, an off-the-shelf learner learns a PFSA. Then, the corer removes infrequently traversed edges and converts the PFSA into an

```
OfLeastSkeletons($S$)
  return $\{s \in S \mid \neg\exists s'$.skeleton of $s'$ precedes
         skeleton of $s$ lexicographically$\}$

RestrictedPermutations($S$, Pos)
  Permutes := $\emptyset$
  Ready := $\{s \in S \mid \neg\exists s' \in S.d(s', s)\}$
* Selected := OfLeastKinds(Ready)
  Foreach $s \in$ Selected
    Rest := RestrictedPermutations($S - \{s\}$, Pos $+ 1$)
    Foreach $\sigma_r \in Rest$
      $\sigma$ := $\sigma_r \cup \{\texttt{Pos} \rightarrow s\}$
      Permutes := Permutes $\cup \{\sigma\}$
  Return Permutes

Standardize($S$)
  Reset all name spaces
  AllStrings := $\emptyset$
  Permutes := RestrictedPermutations(S, 0)
  Foreach $\sigma \in$ Permutes
    $S_{named}$ := StandardName($\sigma(S)$)
    Add $S_{named}$ to AllStrings
  Return the lexicographically smallest element of AllStrings
```

Figure 18: Final standardization algorithm.

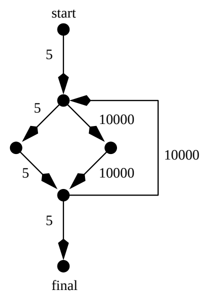


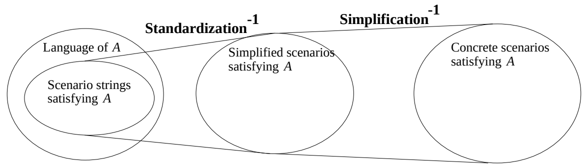
Figure 19: A PFSA for which dropping edges with low weights does not identify the hot core. Edge labels are omitted.

NFA.

The PFSA learner is an off-the-shelf learner [22] that learns a PFSA that accepts the training strings, plus other strings. The learner is a variation on the classic k-tails algorithm [4]. Briefly, the k-tails algorithm works as follows. First, a retrieval tree is constructed from the input strings. The algorithm then computes all strings of length up to k (*k-strings*) that can be generated from each state in the trie. If two states q_a and q_b generate the same k-strings, they are merged. The process repeats until no more merges are possible. The PFSA learner modifies k-tails by comparing how likely two states are to generate the same k-strings.

The resulting PFSA accepts a *superset* of all the strings in the training scenarios, due to the generalizations performed by the learner. The parameter N that controls the size of the extracted is chosen by the user to be large enough to include all of the interesting behavior. It is therefore very likely that the ends of the training scenarios contain *uninteresting* behavior. This is in fact what we see experimentally: the typical PFSA has a "hot" core with a few transitions that occur frequently, with the core surrounded by a "cold" region with many transitions, each of which occurs infrequently. The corer whittles away the "cold" region, leaving just the "hot" core.

The corer can not simply drop edges with low weights. Consider the PFSA in Figure 19 (edge labels are not important and are omit-



Standardization^{-1}
Simplification^{-1}
Language of A
Scenario strings satisfying A
Simplified scenarios satisfying A
Concrete scenarios satisfying A

Figure 20: Scenarios that satisfy A.

ted). Four edges have a weight of 5, which is low compared to the three edges with a weight of 10000. However, any string through this PFSA must traverse the edge out of the start state and the edge into the end state. Despite their low weight, a string is more likely to traverse these edges than it is to traverse the edges with a weight of 10000. Thus, a better measure of an edge's "heat" is its likelihood of being traversed while generating a string from the PFSA. The problem of computing this measure is known as the *Markov chain problem* [15]. The problem reduces to inverting a square matrix with the number of rows and columns equal to the number of transitions in the PFSA.

After computing the heat of each edge, the corer removes all edges below a cutoff parameter, removes unreachable states from the PFSA, and drops the frequencies on the edges. The result is an NFA, which a human can validate by inspection.

5. VERIFICATION

This section discusses how verification tools can use the miner's specifications. Program verification tools distinguish programs that satisfy a specification from programs that do not. Before we can discuss these tools, we must clarify what we mean by "satisfying a specification".

Let A be a specification. By construction, the language of A contains a set of scenario strings (Figure 20). The containment might be strict since the automaton learner can introduce strings into A that are not scenario strings. Because standardization is a many-to-one mapping (see Figure 14), each scenario string corresponds to a set of simplified scenarios. In turn, each scenario string corresponds to a set of concrete scenarios. Figure 20 shows the chain of mappings. We say that the scenarios of Figure 20 *satisfy* A.

Now let T be an interaction trace. We say that T satisfies A if for every seed interaction $i_s \in T$, there is an interaction scenario S_{i_s} seeded by i_s such that S_{i_s} satisfies A. We say that a *program P satisfies a specification A* if any interaction trace T of P's execution satisfies A.

Constructing program verification tools for specifications is outside the scope of this paper, but is the subject of ongoing research. There are two ways that such a tool could work. First, the tool could construct a scenario that satisfies A for each interaction seed encountered while simulating some abstraction of P, reporting an error if no such scenario can be constructed for some seed. Alternatively, the tool could first translate A into an automaton that generates traces instead of scenario strings. The trace automaton generates all traces that satisfy A. The verification tool would then exhaustively search for a trace that is not in A, reporting an error if one is found. Both sorts of tools must be able to simulate simplification and standardization.

Figure 21 shows a trace verification algorithm (not a *program* verification algorithm) that works in the first way. This is the algorithm used in our experiments (see Section 6). `Verify` takes a trace, a specification, and a maximum scenario size. It attempts to verify that the trace satisfies the specification by extracting successively larger scenarios until it finds a satisfactory one or until it reaches the maximum scenario size. Because interactions in the trace are not necessarily ordered as they were in the training traces, the algorithm does not use exactly the same extraction algorithm as the learner. Instead, `Extract`$^*(t_i, \texttt{Size})$ returns all scenarios seeded by t_i with a total of exactly `Size` ancestors and descendants. The distance between the seed and its ancestors and descendants is not important.

```
Satisfies($S$, Spec)
  If $S$ is in the language of Spec
    Return true
  Else Return false

Verify($T = t_0, \ldots, t_n$, Spec, MaxSize)
  Loop:
  Foreach $t_i \in$ SeedsOf($T$)
    Size := 0
    While Size $\leq$ MaxSize
      Scenarios := Extract$^*(t_i$, Size$)$
      Foreach $S \in Scenarios$
        $S_{std}$ := Standardize($S$)
        If Satisfies($S_{std}$, Spec)
        Next Loop
      Size := Size + 1
    Return Fails($t_i$)
```

Figure 21: Trace verification algorithm.

6. EXPERIMENTAL RESULTS

This section presents the results of an experiment in mining specifications from traces of X11 programs.

We analyzed traces from programs that use the Xlib and X Toolkit Intrinsics libraries for the X11 windowing system. The traces record an interaction for each X library call and callback from the X library to client code. The interaction attributes include all arguments and return values of calls, plus the fields of the structures that represent X protocol events. The tracing tool uses the Executable Editing Library (EEL) [17] to instrument Solaris/SPARC executables.

Traces were collected from full runs of widely distributed programs that use the X11 selection mechanism. We studied the selection mechanism since the Interclient Communication Conventions Manual (ICCCM) [25] gives English descriptions of several rules for how well-behaved programs should use the mechanism. The experiment concentrated on a rule that specifies how programs obtain ownership of the selection: the rule says that a client calling `XtOwnSelection` or `XSetSelectionOwner` must pass in a timestamp derived from the X event that triggered the call.

Table 1 lists each program studied, its origin (either the X11 distribution or the X11 `contrib` directory), the number of static calls to the X library routines chosen as seeds, and the number of training scenarios extracted from each trace. One of the authors gathered the traces by running each program for a few minutes, while trying to exercise the selection code by doing cut-and-paste operations, as well as exercising as much other functionality as possible in a short time.

Specification mining depends on a sizable training set of well-debugged traces. In our case, the number of training traces was small, and as it turned out, several contained violations of the rule. As a result, the miner was not able to discover the rule when trained on all of the programs. In order to learn the rule, we needed to remove the buggy traces from the training set. We hypothesized that our miner could help find the bugs, even with a poor training

Name	Source	Static seeds	Scenarios
bitmap	distrib	1	6
xclipboard	distrib	2	2
xconsole	distrib	1	1
xcutsel	distrib	1	4
xterm	distrib	1	6
clipboard	contrib	1	2
cxterm	contrib	1	9
display	contrib	4	16
e93	contrib	1	2
kterm	contrib	1	4
nedit	contrib	2	2
pixmap	contrib	1	11
rxvt	contrib	1	4
ted	contrib	3	9
test_canvas	contrib	1	4
ups	contrib	1	3
xcb	contrib	1	11

Table 1: X11 client programs studied in the experiment.

Name	Verifies?	Reason for failure	Action
xcb	n/a	n/a	accept
bitmap	no	spec. too narrow	accept
ups	no	bug!	reject
ted	no	spec. too narrow	accept
rxvt	yes	n/a	accept
xterm	no	spec. too narrow	accept
display	no	spec. too narrow	accept
xcutsel	no	spec. too narrow	accept
kterm	yes	n/a	accept
pixmap	yes	n/a	accept
cxterm	yes	n/a	accept
xconsole	no	benign violation	reject
nedit	no	spec. too narrow	accept
e93	no	bug!	reject
xclipboard	no	benign violation	reject
clipboard	no	benign violation	reject

Table 2: Results of processing each client program, in the order in which they were processed.

set. Identifying the buggy traces without the miner would require inspecting each trace manually for bugs.

Using the miner, we predicted that, while we would have to inspect the first few traces, once a few correct traces had been collected, the miner's rule could be used to automatically validate the remaining traces. In this experiment, we arranged the client programs in random order and went through the following iterative process:

```
Run the first program and gather a trace
Mine a specification from the trace
Expert examines the specification
Expert extracts hot core
If the specification is not correct
  Select another random order and start over
For each remaining client program in order
    Run the program and gather a trace
    Verify the trace against the specification
    If verification succeeds
      Add the trace to the training set
      Generate a new specification
    Else
      Examine the scenarios that failed
      If no scenario violates the ICCCM rule
        Add the trace to the training set
        Generate a new, more general specification
      Else
        Report the bug
```

For each trace that fails to verify, the expert either marks it as buggy or includes it in the training set. The expert decides whether the initial specification is correct: in our experiment, we accepted the initial specification if we did not see any obvious bugs in the first set of training scenarios. The expert also needs to extract the hot core, since the training set is too small to use the corer.

The experiment tested three hypotheses:

Hypothesis 1 The process will find bugs and reduce the number of traces that the expert must inspect.

Hypothesis 2 The miner's final specification will match the rule in the ICCCM.

Hypothesis 3 The corer and the human will agree on which states in the final PFSA belong in the final specification.

Table 2 lists the client programs in the order in which they were processed. Out of the first six traces accepted (not including the initial trace), five were rejected by an overly narrow specification. At this point, the specification seemed to stabilize: out of the next four accepted, only one was initially rejected by the dynamic verifier. The expert did not have to inspect 4 out of the 16 the traces, which supports the second part of **Hypothesis 1**. We conjecture that if the process had continued, the false rejection rate would have continued to drop.

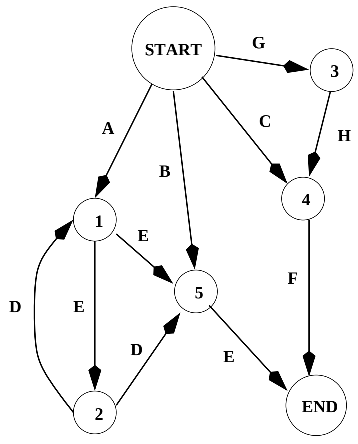
Five of the programs violated the rule in the ICCCM. We found three programs with benign violations of the specification and two programs with bugs. The specification applies to programs that use the selection mechanism to do cut-and-paste, while the programs with benign violations used the selection mechanism to implement their own communication protocol. These violations indicate that the rule described by the ICCCM is not universally applicable and that the document should be clarified. Thus, the specification miner helped find bugs and an documentation omission (an unexpected benefit).

Figure 22 is the specification from the experiment. For legibility's sake, the figure omits some arguments. These arguments did not participate in dependences within the core of the specification. The specification is compact, with six states and nine edges. It also matches the English rule very closely, with most complexity arising from the several ways in which the X API receives an event. In addition, the specification exposes a common pattern in which the client calls XSetSelectionOwner repeatedly until XGetSelectionOwner indicates that the call was successful.

Our final hypothesis was that the corer and the expert would agree on which states in the final PFSA should be thrown out. The final PFSA had 27 states. The expert, who did not have access to the corer's results, threw out 15 of these and retained 12; the remaining twelve were merged to form the NFA in Figure 22. The corer and the expert disagreed on five out of the 27 states, or 19%. The corer assigned likelihoods lower than 6% to 13 of the 15 deleted states, and likelihoods higher than 13% to 9 of the 12 retained states. The other 2 deleted states had likelihoods of 13% and 20%, while the remaining retained states had likelihoods of 5%, 6%, and 9%. Thus,

7. RELATED WORK

Ernst et al. also proposed automatic deduction of formal specifications [11]. Their Daikon tool works by learning likely invariants involving program variables from dynamic traces. The resulting



A = XNextEvent(time = X21_0)
B = XNextEvent(time = X21_0) or B = XtDispatchEvent(time = X21_0)
or B = XIfEvent(time = X21_0)
C = XtDispatchEvent(time = X21_0) or C = XtEventHandler(time = X21_0)
or C = XtLastTimeStampProcessed(time = X21_0)
D = XGetSelectionOwner
E = XSetSelectionOwner(time = X21_0)
F = XtOwnSelection(time = X21_0)
G = XtActionHookProc(time = X21_0)
H = XInternAtom

Figure 22: The NFA from the selection ownership specification.

formal specifications is the key difference between their approach and ours. 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, our specifications express temporal and data-dependence relationships among calls to an API. Our 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.

Recently, Ernst et al. presented techniques for suppressing parts of their learned specifications that are not useful to a programmer [19]. In the context of our temporal specifications, this result corresponds to appropriately selecting the heavy core of the initial PFSA.

Another related tool is Houdini [12], 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 our tool is on discovering temporal properties that hold across all programs that use an interface.

Other authors have described tools that extract automaton-based models. Cook and Wolf describe a tool for extracing FA models of software development processes from traces of events [8]. Our work differs in that we extract specifications from program traces, which must be reduced to a simpler form before they are palatable for an FA learner. Ghosh et al. describe several techniques for learning the typical behavior of programs that make system calls [13, 18]. Since they intend their models for intrusion detection, the models need only characterize a particular program's behavior, while our miner 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 [27]. Their system also extracts models that apply only to a single program. Finally, Reiss and Renieris also extract structure from traces [23], but they model the sequence of operations on individual objects, not the data and temporal dependences across several objects.

8. CONCLUSION

This paper addresses an important problem in the program-verification tool-chain, namely the problem of semi-automatic formulation of correctness specifications that could be accepted by model checkers and other similar tools. We have formulated the problem as a machine learning problem and provided an algorithm based a reduction to finite automaton learning. While some more experimental work remains ahead of us, initial experience is promising.

Acknowledgements

We thank Anand Raman, Peter Andreae, and Jon D. Patrick for allowing us to use their PFSA learner, as well as the anonymous referees for their many helpful comments on an early draft of this paper. This work was supported in part by the National Science Foundation with grants CCR-0093275 and CCR-0103670, the University of Wisconsin Graduate School, and donations from IBM and Microsoft Research.

9. REFERENCES

[1] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213, July 2001.

[2] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, number 2057 in Lecture Notes in Computer Science, pages 103–122, May 2001.

[3] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SOGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ACM SIGPLAN Notices, pages 97–103, July 2001.

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

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

[6] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP18)*, pages 73–88, October 2001.

[7] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP. Client-server Programming and Applications, BSD Socket Version*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1993.

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

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

[10] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in system code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP18)*, pages 57–72, October 2001.

[11] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions in Software Engineering*, 27(2):1–25, February 2001.

[12] Flanagan and Leino. Houdini, an annotation assistant for ESC/java. In *International Symposium on FME 2001: Formal Methods for Increasing Software Productivity, LNCS*, volume 1, 2001.

[13] Anup K. Ghosh, Christoph Michael, and Michael Shatz. A real-time intrusion detection system based on learning program behavior. In *RAID 2000*, volume 1907 of *Lecture Notes in Computer Science*, pages 93–109, 2000.

[14] E Mark Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[15] Michel Gondran and Michel Minoux. *Graphs and Algorithms*. John Wiley and Sons, 1984.

[16] Michael Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. On the learnability of discrete distributions. In *Proceedings of the Twenty-sixth ACM Symposium on Theory of Computing*, pages 273–282, 1994.

[17] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[18] Christoph Michael and Anup Ghosh. Using finite automata to mine execution data for intrusion detection: a preliminary report. In *RAID 2000*, volume 1907 of *Lecture Notes in Computer Science*, pages 66–79, 2000.

[19] William G. Griswold Michael D. Ernst, Adam Czeisler and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

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

[21] Anand Raman, Peter Andreae, and Jon Patrick. A beam search algorithm for pfsa inference. *Pattern Analysis and Applications*, 1(2), 1998.

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

[23] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 221–232, Los Alamitos, California, May12–19 2001. IEEE Computer Society.

[24] Dana Ron, Yoram Singer, and Naftali Tishby. On the learnability and usage of acyclic probabilistic finite automata. In *Proceedings of the 8th Annual Conference on Computational Learning Theory*, pages 31–40. ACM Press, New York, NY, 1995.

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

[26] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[27] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.