Chapter 8

Automatic Inference and Effective Application of Temporal Specifications

Jinlin Yang

Microsoft

David Evans

University of Virginia

1

[1]This chapter is partly based on [126, 126, 128–130].

8.1 Introduction

For many years, researchers have claimed that software specifications can be used to improve many software development activities. A formal specification documents important properties and hence is useful in understanding programs [65, 105]. Formal specifications can be used to automatically generate test inputs [24, 35, 92]. Program verification needs a formal specification that defines the correct behaviors of a program [56,65,105]. Other uses include refining a specification into a correct program [1] and protecting a programmer from making changes that violate important invariants [44].

Despite these benefits, formal specifications are rarely available for real systems [67, 79, 81]. To help realize the benefits of formal specifications, this work develops techniques for automatically inferring formal specifications and investigates the uses of inferred specifications in software development.

This work focuses on temporal properties. Temporal properties constrain the order of a program's states [80, 105]. For example, acquiring a lock should always be followed by releasing the lock. Temporal properties are important in many types of systems such as network protocols. Satisfying temporal properties is crucial for establishing a program's correctness properties.

8.1.1 Prior Work Inferring Specifications

Several researchers have recognized the unavailability of specifications as an important problem and studied specification inference and its application to software development [3,4,44,48,64,121,123]. A *specification inference technique* automatically discovers a formal specification of a target program by analyzing program artifacts. A specification inference tool is often called a *specification miner* [4], *specification synthesizer* [3], or *specification prospector* [90]. A program's source code or execution traces are the most common artifacts used by specification miners, though it is also possible to analyze other artifacts including design documents, bug reports, and revision history [22,89].

A static inference technique analyzes source code. In addition to eliminating the need to execute the target program, a static inference technique can theoretically examine all execution paths of a program and therefore can infer precise specifications of a program. However, features common in modern programming languages such as pointers, branches, loops, threads, inheritance, and polymorphic interfaces, can be too expensive to analyze precisely from source code.

In contrast, a dynamic inference technique analyzes execution traces that have precise information about pointers, branches, threads, and the other features of programming executions. However, dynamic analysis only sees execution paths present in the traces and therefore might infer specifications

that are stronger than the program itself when the executions do not cover all possible scenarios.

We take the dynamic inference approach in our work for several reasons. The temporal specifications we aim to infer typically involve objects and threads that are difficult to analyze statically. In addition, we want to apply our techniques to large real systems that typically have complex control flow structures for which static techniques don't scale well.

Previous dynamic specification inference techniques have shown promising results in many areas, including bug detection [60, 89, 98, 106], test case selection [58,61,125], and program steering [86]. However, all of the results to date have been on relatively small execution traces. For example, the largest execution traces analyzed by Daikon have only hundreds of variables, whereas a large system usually has thousands of variables [44, 103].

Scaling dynamic inference to handle large real systems involves several important challenges. The inference technique must effectively deal with imperfect execution traces. An *imperfect execution trace* is a trace that contains event sequences that violate a property specification that is necessary for the correctness of a system. Suppose we want to learn the temporal specification of a type of lock. If our target program neglects to release this type of lock during some executions (due to bugs), running this program would produce an imperfect trace that fails to exhibit the rule for correctly using the lock. A dynamic inference technique needs to effectively deal with such imperfect execution traces otherwise it would risk missing important specifications in the inference results. For example, the Strauss specification miner requires human guidance to tune an imperfect trace so that it can discover important specifications that would otherwise be missing [4]. Daikon requires 100% satisfaction of a pattern [44], which might exclude important specifications if the execution trace is imperfect. Although we aim to handle imperfect traces, we assume that our traces are mostly correct - our target program should exhibit the desirable behavior most of the time.

Our inference technique must be able to select interesting specifications. An *interesting specification* is a specification whose violation would produce bad consequences. For example, we consider specifications about using critical system resources such as locks and transactions to be interesting. Such properties are important because violating them can have serious consequences such as causing system crashes [6,31] and opening security vulnerabilities [18]. Selecting interesting specifications is important because for a large program thousands of properties may be inferred, only a small fraction of which are interesting. We present several techniques that can effectively increase the percentage of interesting properties in the results.

8.1.2 Contributions

We describe dynamic inference techniques for automatically inferring temporal specifications and experimentally evaluate our techniques on real sys-

tems, as well as several different applications of the inferred properties. Our dynamic inference techniques address three limitations of prior inference work described in the previous section:

(a) The inference algorithms scale poorly with the size of the program and the execution traces.

(b) Previous dynamic inference techniques do not work well in situations where perfect traces are not available.

(c) A significant portion of the inferred properties are uninteresting. For small programs, it is feasible to manually select the interesting properties; for large programs, property selection must be mostly automated.

In particular we make the following contributions in the area of scaling dynamic inference techniques:

(a) We develop a scalable inference algorithm that can analyze large execution traces (Section 8.2).

(b) We create a statistical inference algorithm that can deal with imperfect execution traces (Section 8.2.5).

(c) We develop two heuristics for eliminating uninteresting properties (Section 8.2.6). These heuristics increase the percentage of interesting properties in the inference results and are crucial for the approach to be useful in practice.

(d) We present a chaining method for constructing large finite state automata out of a set of smaller ones (Section 8.2.7). This method is useful for presenting a large number of inferred properties in a more readable way.

In order to evaluate our approach, we built a prototype tool called Perracotta and applied it to several real systems including Microsoft Windows and the JBoss Application Server. The results demonstrate that the dynamic analysis technique is useful in several different software development activities. Section 8.3 presents the inference results on real systems with a focus on helping program understanding. Section 8.4 describes other uses of the inferred properties. Section 8.4.1 describes combining Perracotta with two program verification tools. Section 8.4.2 presents the experiments of using Perracotta in program differencing. We show that the technique can aid in program differencing by discovering important differences among multiple versions of real systems.

8.2 Specification Inference

To illustrated our dynamic temporal specification inference approach we use a simple Producer-Consumer program as a running example (Section 8.2.1). Our inference approach follows the steps shown in Figure 8.1. An *instrumentor* instruments the program to monitor information of interest (Section 8.2.2). Then, the instrumented program is executed against a set of test cases to produce execution traces (Section 8.2.3). Next, the inference engine matches the traces against a set of predefined property templates (Section 8.2.4). A post-processor selects the interesting properties out of the matched properties using several heuristics (Section 8.2.6). The chaining method aims to condense the inferred properties so that users can better comprehend them (Section 8.2.7). Section 8.2.8 describes Perracotta, a prototype implementation of our inference approach.

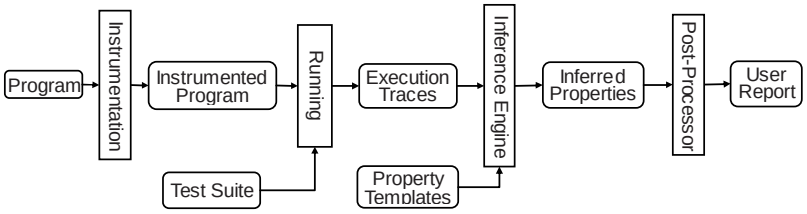


FIGURE 8.1: Overview of our approach.

8.2.1 A Running Example: Producer-Consumer

The Java program in Figure 8.2 implements a simplified version of the Producer-Consumer problem. The Producer and Consumer classes implement a Producer and a Consumer respectively. Only one Producer and one Consumer object exist at any time. The Producer interacts with the Consumer through a global Buffer object, buf, which is a static member of the Heap class. At any time, a Buffer object can only hold one integer element, queue, whose value can be retrieved through the take method and updated through either the add method or the stop method. The buffer is empty when the value of queue is -1. The Producer inserts a new integer into buf by calling its add method and the Consumer removes an integer from buf by calling its take method. The Producer iteratively inserts integers from 1 to n (as designated by program arguments) into buf, while the Consumer takes those numbers from buf and prints them out. After the Producer calls the stop method that writes 0 to buf, the Consumer reads 0, exits the run loop, and terminates.

All three methods of the Buffer class, take, add, and stop, are declared with the Java synchronized keyword to ensure mutual exclusion among mul-

```
class Buffer {
   int queue = -1;
   public synchronized int take() {
      int value;
      while (queue < 0)
         try { wait(); } catch(InterruptedException ex) {}
      value = queue;
      queue = -1;
      notifyAll();
      return value;
   }
   public synchronized void add(int x) {
   while (queue != -1)
         try { wait(); } catch(InterruptedException ex) {}
      queue = x;
      notifyAll();
   }
   public synchronized void stop() {
      while (queue != -1)
         try { wait (); } catch (InterruptedException ex) {}
      queue = 0; notifyAll ();
   }
}

class Heap { static Buffer buf; }

class Producer {
   static public void main(String[] args) {
      Heap.buf = new Buffer();
      (new Consumer()).start();
      for(int i = 1; i < Integer.valueOf(args[0]).intValue(); i++)
         Heap.buf.add(i);
      Heap.buf.stop();
   }
}

class Consumer extends Thread {
   public void run () {
      int tmp = -1;
      while ((tmp = Heap.buf.take ()) != 0)
         System.err.println ("Result: " + tmp);
   }
}
```

FIGURE 8.2: A Java implementation of the simplified Producer-Consumer problem.

tiple threads that access a same Buffer object. We implement synchronization among multiple threads using the commonly used Java wait-notify idiom.

This program exhibits two interesting properties: (1) inserting an integer to buf alternates with removing an integer from buf. Hence, the Producer cannot overwrite a new element before the Consumer retrieves it. Furthermore, the Consumer cannot take an element from an empty buffer. (2) once the Producer calls the stop method, the Consumer must eventually stop.

8.2.2 Instrumentation

A program execution involves a great deal of information: values of parameters and object fields, thread contexts, branches taken, exceptions raised, etc. Some information represents state (e.g., object fields), whereas other information captures control flow (e.g., branches taken, exceptions raised).

Ideally we would record every detail of a program's execution. This is impractical for several reasons. Instrumenting everything without affecting a program's normal behavior is difficult. In a real-time system, the overhead introduced by the instrumentation code might prevent a process from meeting its deadline. Furthermore, collecting all information does not scale to long-running systems as the size of the data becomes too large to be efficiently stored and processed.

Our instrumentor instruments a program at the method level and records the thread contexts, argument values, and return values (Section 8.2.8). Our instrumentor instruments the entrance and exit events of all the methods in the Producer-Consumer program. For example, executing the instrumented Producer-Consumer program with 5 as its input produces the execution trace shown in Figure 8.3. To simplify presentation, we do not include the argument values and return values in this trace.

Each line in the trace corresponds to a single event. An event starts with either Enter or Exit corresponding to the entrance and exit event respectively. The middle part of an event is a method's name. The last part of an event includes the thread context information and, optionally, argument values. For example, Enter:Producer.main():[main] indicates that the main thread enters the main method in the Producer class.

8.2.3 Running

The executions affect the results of any dynamic analysis. Our goal is to develop a dynamic inference technique that works with readily available or easily produced execution traces. Hence, we run a target system's regression test suite if one is available. Furthermore, when generating inputs is necessary, we either randomly select inputs from a program's input domain [39, 122] or exhaustively generate all inputs within certain bounds [13, 24, 78].

For the example, we run the Producer-Consumer program with 100 randomly selected integers between 1 and 10000.

```
Enter:Producer.main():[main]
Enter:Buffer.add():[main]
Enter:Consumer.run():[Thread-1]
Exit:Buffer.add():[main]
Enter:Buffer.take():[Thread-1]
Enter:Buffer.add():[main]
Exit:Buffer.take():[Thread-1]
Exit:Buffer.add():[main]
Enter:Buffer.add():[main]
Enter:Buffer.take():[Thread-1]
Exit:Buffer.take():[Thread-1]
Exit:Buffer.add():[main]
Enter:Buffer.take():[Thread-1]
Enter:Buffer.add():[main]
Exit:Buffer.take():[Thread-1]
Exit:Buffer.add():[main]
Enter:Buffer.take():[Thread-1]
Enter:Buffer.stop():[main]
Exit:Buffer.take():[Thread-1]
Exit:Buffer.stop():[main]
Enter:Buffer.take():[Thread-1]
Exit:Producer.main():[main]
Exit:Buffer.take():[Thread-1]
Exit:Consumer.run():[Thread-1]
```

FIGURE 8.3: A trace of running the Producer-Consumer program.

Each line corresponds to a single event that represents the entrance or exit of a method. We omit the method's signature, argument values, and return values to simplify presentation. For example, Enter:Producer.main():[main] indicates that the main thread enters the main method in the Producer class.

TABLE 8.1: Temporal property templates.

For example, the regular expression of the MultiEffect pattern is $(PSS^*)^*$. Hence, PSS is a string that satisfies the MultiEffect template, whereas PPS or SPS do not satisfy the template.

Name	Regular Expression	Satisfying Example	Violating Examples
Response	*S*(PP*SS*)**	*SPPSS*	*SPPSSP, PSP*
Alternating	*(PS)**	*PSPS*	*PSS, PPS, SPS*
MultiEffect	*(PSS*)**	*PSS*	*PPS, SPS*
MultiCause	*(PP*S)**	*PPS*	*PSS, SPS*
EffectFirst	*S*(PS)**	*SPS*	*PSS, PPS*
CauseFirst	*(PP*SS*)**	*PPSS*	*SPSS, SPPS*
OneCause	*S*(PSS*)**	*SPSS*	*PPSS, SPPS*
OneEffect	*S*(PP*S)**	*SPPS*	*PPSS, SPSS*

8.2.4 Inference Engine

The inference engine produces a set of properties from a trace. Here, we introduce the predefined property templates in Section 8.2.4 and describe our inference algorithm that scales to large execution traces.

Property Templates

A property template abstracts a set of concrete properties. Our property templates have two parameters that can be substituted with values to generate concrete properties. Property templates determine the properties that can be inferred. It is essential that these templates capture properties users care about.

Dwyer et al. developed a temporal property pattern library after surveying hundreds of temporal property specifications checked by program verification tools [41]. One pattern in their library is the Response pattern, which constrains the cause-effect relationship between two events P and S so that P's occurrence must be followed by S's occurrence. We use regular expressions to represent these patterns. For example, $[\neg P]^*(P[\neg S]^*S[\neg P]^*)^*$ is the regular expression for the Response pattern. After removing all events other than P and S, the Response pattern can be simplified as $S^*(PP^*SS^*)^*$. The Response pattern does not constrain the number of P events, the number of S events, or whether the S event can occur before the P event. As a result, knowing that two events satisfy the Response pattern does not give us precise information about their relationship.

We use the seven property patterns based on the Response pattern shown in Table 8.1. For example, the MultiEffect pattern is $(PSS^*)^*$; PSS is a string that satisfies the MultiEffect pattern. Furthermore, the MultiEffect pattern only

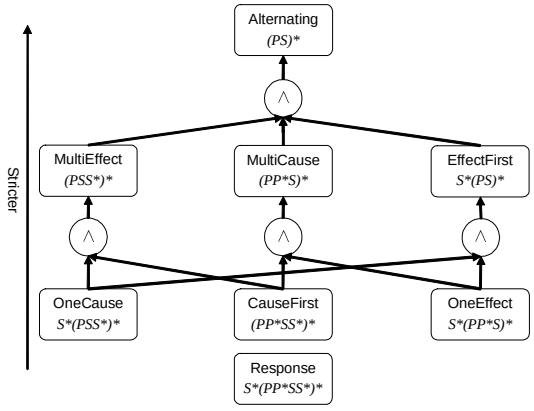


FIGURE 8.4: Partial order of property templates.

Each box shows a property template and its regular expression representation. A pattern A is stricter than another pattern B if $L(A) \subset L(B)$, where $L(A)$ means all the strings accepted by A. The eight patterns form a partial order in terms of their strictness. For example, Alternating is the strictest pattern among them. In addition, these patterns have an internal logical relationship as illustrated by the logical $\wedge$ operators among them. For example, a string satisfies the MultiEffect pattern if and only if it satisfies the OneCause and CauseFirst patterns.

allows one P event to occur between two S events and also requires that the first P event to occur before the first S event. Hence, PPS and SPS do not satisfy the MultiEffect pattern. Another pattern is Alternating that requires a strictly alternating relationship between two events. Its regular expression is $(PS)^*$. We use the notation $P \rightarrow S$ to indicate that P and S satisfy the Alternating pattern.

The Alternating pattern is *stricter* than the MultiEffect pattern because all strings that satisfy the Alternating pattern must also satisfy the MultiEffect pattern but not vice versa. Formally, we say a pattern A is stricter than another pattern B if $L(A) \subset L(B)$, where $L(A)$ is the set of strings accepted by A. The seven new patterns form a partial order in terms of their strictness as illustrated in Figure 8.4. Alternating is the strictest pattern among the seven patterns. In addition, these patterns have internal logic relationship as illustrated by the logical $\wedge$ operators among them. For example, a string satisfies the MultiEffect pattern if and only if it satisfies the OneCause and CauseFirst patterns.

We can derive the strictest pattern a string satisfies by exploring the logical relationship among the patterns. Our algorithm first determines which of the three primitive patterns a string satisfies and then deduces the strictest pattern. For example, if a string satisfies all three primitive patterns, then the strictest pattern it satisfies is Alternating.

In addition to the patterns with two parameters, we also derive two Alter-

nating patterns that have three parameters. The two-effect-alternating pattern, $P \rightarrow S \mid T$, allows P to alternate with either S or T. The two-cause-alternating pattern, $P \mid S \rightarrow T$, requires either P or S to alternate with T. These patterns correspond to properties of real programs. For example, a file, after being successfully opened, can be either read or written, and finally be closed.

Pattern Matching Algorithm

All the patterns described in the previous section have only two or three parameters (e.g., P, S, and T). We present our pattern matching algorithm using the Alternating template as an example, although the essential ideas of the algorithm also work for other templates. Given a trace with N distinct events and L events total, we want to infer which pairs of events can satisfy the Alternating pattern. For example, a hypothetical trace $ABCACBDC$ has four distinct events: A, B, C, and D. Hence, there are 12 ways to instantiate the Alternating template: $(AB)^*$, $(AC)^*$, $(AD)^*$, $(BA)^*$, ..., $(DC)^*$. Of these, the only Alternating property that string satisfies is $A \rightarrow B$.

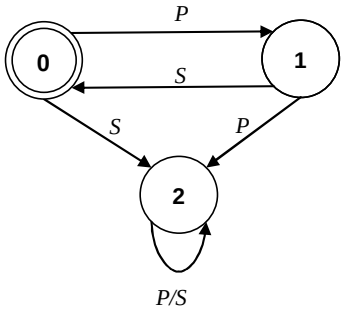
A brute force algorithm would check the string against all 12 instantiations of the property one by one. However, this algorithm does not scale when the number of distinct events becomes large. For N distinct events, there are $N(N-1)$ instantiations of a pattern with two parameters. Checking the string against each instantiation needs to traverse the string once. Hence, the naïve algorithm has running time in $\Theta(N^2L)$.

Next, we introduce a more efficient inference algorithm with time complexity in $\Theta(NL)$ and space complexity in $\Theta(N^2)$.

The algorithm encodes a property template as a table. Figure 8.5(a) shows a finite state machine representation of the Alternating template. State 0 is both the initial state and the accepting state. State 2 is the error state. We can encode the transitions in this FSM as the table shown in Figure 8.5(b). The column header is the current state of the FSM. The row header is the current event in the trace. Given the current state and the current event, our algorithm determines the next state by looking it up in the table. For example, if the current state is state 0 and the event is P, the next state is state 1.

Figure 8.6 shows our inference algorithm. Figure 8.7 shows how our inference algorithm infers which of the 12 instantiations of the Alternating template the trace $ABCACBDC$ satisfies.

The algorithm scans a trace twice. In the first pass, it identifies all the distinct events and creates a mapping between the event names and integer index numbers (lines 3-13). After scanning the whole trace, the algorithm creates state, an $N \times N$ array, for keeping track of the states of the instantiations of the Alternating pattern (line 17). The value of state[i][j] corresponds to the FSM state of the instantiation in which P is $Event_i$ and S is $Event_j$. The elements of this array except the diagonal ones are initialized to 0 since all FSMs start in the initial state (lines 18-23). The diagonal elements are initialized to 2 (i.e., the error state) because the two events cannot be equal (line 21). In



(a) Finite state machine

Event \ Current State	0	1	2
P	1	2	2
S	2	0	2

(b) Table

FIGURE 8.5: Representing the Alternating template in different forms.

the second pass, our algorithm rescans the execution trace (line 27). When it reads an event from the trace, it updates the state array (lines 28-34). Here the key observation is that an event, $Event_k$, could be either the P event or the S event. If $Event_k$ is the P event, our algorithm updates the k-th row of the state array (line 32). If $Event_k$ is the S event, our algorithm updates the k-th column of the state array (line 34). Our algorithm updates the state by looking up the pre-encoded tables of the FSMs (Figure 8.5). After scanning the trace twice, if state[i][j] is in an accepting state, our algorithm outputs $Event_i \rightarrow Event_j$ as a satisfied Alternating property (lines 37-41).

This algorithm has time complexity in $\Theta(NL)$ and space complexity in $\Theta(N^2)$. The loop from line 11 to 13 has running time in $\Theta(L)$: it scans the trace once, updating the event2index mapping for each element. The loop from line 18 to 23 has running time in $\Theta(N^2)$. The loop from line 28 to 34 updates one row and one column of the state array for each event in the trace. Hence, the time complexity of the loop is in $\Theta(NL)$. Finally, the loop from line 37 to 41 has time complexity in $\Theta(N^2)$. Because $L \geqslant N$, the time complexity of the algorithm is in $\Theta(NL)$. The algorithm requires creating an $N \times N$ array. Therefore, its space complexity is in $\Theta(N^2)$.

The Producer-Consumer trace in Figure 8.3 (see page 251) has 10 distinct events. Hence, there are 90 candidate Alternating properties. The inference algorithm determines that the trace satisfies the 17 Alternating properties shown in Figure 8.8. The two properties in boldface represent the property that whenever the Producer sends out the stop signal, the Consumer will stop execution eventually. The next five properties are uninteresting because they correspond to the trivial fact that entering and exiting a method always alternate. The remaining properties reflect the static call graph and are not very interesting either (e.g., Producer.main() calls Consumer.run()). Section 8.2.6 describes the post-processing component that selects interesting properties, eliminates redundant and uninteresting properties, and better presents the results.

```
1   void Infer( RandomAccessFile tracefile )
2
3       byte[ ][ ] ALTERNATING = { {1, 2}, {2, 0}, {2, 2} };
4       // A mapping between an event to its index
5       Hashtable event2index = new Hashtable();
6       // The number of distinct events
7       int n = 0;
8       String current_event = null;
9
10      // First pass: create event table
11      while( ( current_event = tracefile.readLine() ) != null )
12          if( !event2index.contains( current_event ) )
13              event2index.add( current_event, n++ );
14
15      // Create a table for keeping track of the states
16      // state[ i ][ j ] records the current state of the "Event_i → Event_j" template
17      byte state[ ][ ] = new byte[ n ][ n ];
18      for( int i = 0; i < n; i++ )
19          for( int j = 0; j < n; j++)
20              if( i == j )
21                  state[ i ][ j ] = 2;
22              else
23                  state[ i ][ j ] = 0;
24
25      // Second pass
26      // restart from the beginning of the trace file
27      tracefile.seek( 0 );
28      while( ( current_event = tracefile.readLine() ) != null )
29          k = event2index.get( current_event );
30          for( int i = 0; i < n; i++ )
31              // Update the state when current_event is the P event
32              state[ k ][ i ] = ALTERNATING[ state[ k ][ i ] ][ 0 ];
33              // Update the state when current_event is the S event
34              state[ i ][ k ] = ALTERNATING[ state[ i ][ k ] ][ 1 ];
35
36      // Check final state
37      for( int i = 0; i < n; i++ )
38          for( int j = 0; j < n; j++ )
39              // Is the state in an accepting state?
40              if( state[ i ][ j ] == 0 )
41                  System.out.println( i + "→" + j );
```

FIGURE 8.6: The inference algorithm for the Alternating pattern.

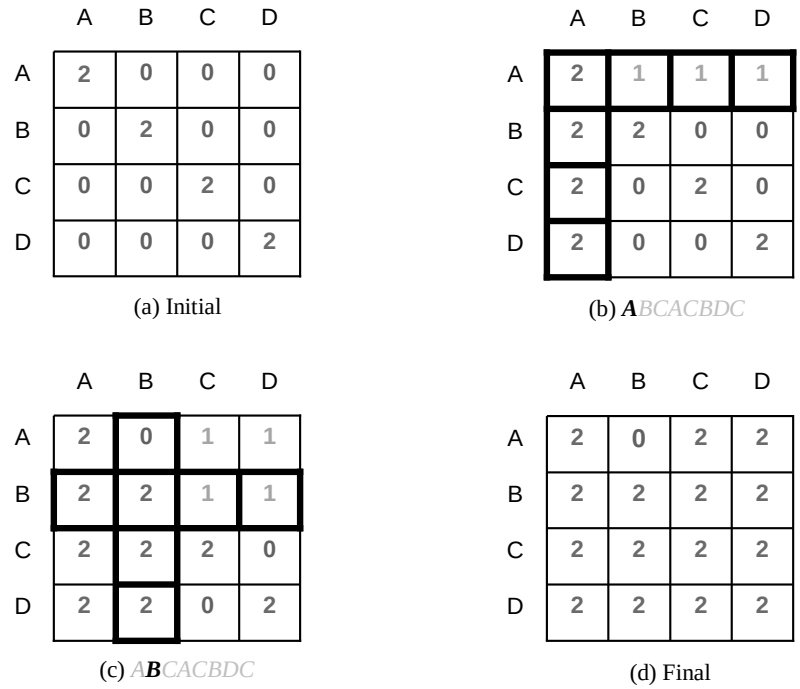


FIGURE 8.7: Inferring Alternating properties from a hypothetical trace $ABCACBDC$.

Enter:Buffer.stop():[main] → Exit:Consumer.run():[Thread-1]
Exit:Buffer.stop():[main] → Exit:Consumer.run():[Thread-1]

Enter:Buffer.add():[main] → Exit:Buffer.add():[main]
Enter:Consumer.run():[Thread-1] → Exit:Consumer.run():[Thread-1]
Enter:Producer.main():[main] → Exit:Producer.main():[main]
Enter:Buffer.stop():[main] → Exit:Buffer.stop():[main]
Enter:Buffer.take():[Thread-1] → Exit:Buffer.take():[Thread-1]

Enter:Producer.main():[main] → Enter:Consumer.run():[Thread-1]
Enter:Producer.main():[main] → Enter:Buffer.stop():[main]
Enter:Producer.main():[main] → Exit:Buffer.stop():[main]
Enter:Producer.main():[main] → Exit:Consumer.run():[Thread-1]
Enter:Consumer.run():[Thread-1] → Enter:Buffer.stop():[main]
Enter:Consumer.run():[Thread-1] → Exit:Buffer.stop():[main]
Enter:Consumer.run():[Thread-1] → Exit:Producer.main():[main]
Enter:Buffer.stop():[main] → Exit:Producer.main():[main]
Exit:Buffer.stop():[main] → Exit:Producer.main():[main]
Exit:Producer.main():[main] → Exit:Consumer.run():[Thread-1]

FIGURE 8.8: Alternating properties inferred from the trace in Figure 8.3.

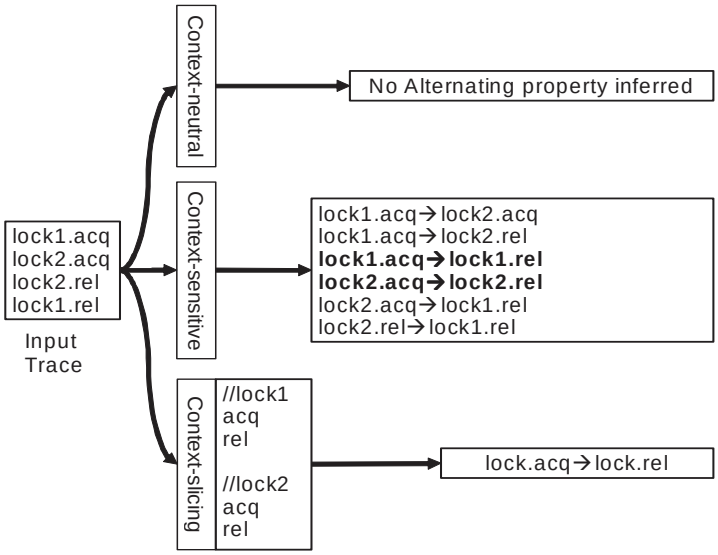


FIGURE 8.9: Context handling techniques.

Context-neutral does not differentiate between *lock1.acq/rel* and *lock2.acq/rel* and does not infer any Alternating property. *Context-sensitive* differentiates between the methods of *lock1* and *lock2* and infers six Alternating properties. However, only two of the six properties shown in boldface correspond to the property that acquiring a lock should alternate with releasing a lock. *Context-slicing* slices the original trace by the identity of the lock and produces two subtraces. Hence, *context-slicing* infers lock.acq → lock.rel.

Handling Context Information

A major advantage of dynamic analysis over static analysis is the ready availability of precise context information including threads, objects, argument values, and return values. This section presents our techniques for using context information to infer more precise properties.

We use three general approaches: *context-neutral*, *context-sensitive*, and *context-slicing*. The context-neutral approach treats two events with same static signature but different context information as the same event, whereas the context-sensitive approach considers them as two distinct events.

For example, consider the example trace in Figure 8.9. The context-neutral approach sees two distinct events (lock.acq and lock.rel), but the context-sensitive approach sees four events (lock1.acq, lock1.rel, lock2.acq, and lock2.rel). Context-neutral analysis does not infer lock.acq → lock.rel. On the other hand, context-sensitive analysis infers six Alternating properties, only two of which are useful (shown in boldface in Figure 8.9). Neither context-sensitive nor context-neutral analysis infers that lock.acq and lock.rel alternate for a same lock object. To infer this property, we need to generalize the results of the context-sensitive analysis by slicing the original trace into separate traces

based on object identity. We call this the *context-slicing* approach. Context-slicing produces two traces from which our inference algorithm infers lock.acq $\rightarrow$ lock.rel. In addition to slicing on object identities, context-slicing can also slice thread identities, argument values, and return values.

The results of context-sensitive analysis are the most complete, but are not useful without generalization. Context-slicing is a simple way to generalize the results of context-sensitive analysis. A limitation of context-slicing is that it cannot detect properties that involve more than one context. For example, if slicing is done by threads, an Alternating pattern between P in one thread and S in another thread would not be detected.

8.2.5 Approximate Inference

The algorithm in Figure 8.6 only infers properties that are completely satisfied by the trace. For example, P and S satisfy the Alternating template in $PSPSPSPSPSPSPSPSPSPS$ but not in $PSPSPSPSPSPSPSPSPSP$ because the last P does not have a corresponding S. This 100% satisfaction requirement is a big limitation of the original algorithm when applied to traces from real systems. This algorithm would miss many interesting properties because the traces are imperfect.

Imperfect Traces

An *imperfect trace* is a trace that contains event sequences that violate a property specification that is necessary for the correctness of a system.

Bugs in a program are the most insurmountable reason for imperfect execution traces. The hypothetical buggy program in Figure 8.10 illustrates this. Suppose the code on line 5 has no side effect on either i or $lock$. The while loop exits when i becomes greater than 10. During each iteration except the last one, the loop body acquires a lock, does some work, and releases the lock. On the last iteration, however, the program does not release the lock. As a result, executing the program produces the trace $PSPSPSPSPSPSPSPSPSP$, where P represents lock.acquire and S represents lock.release. Although this trace does not satisfy the $P \rightarrow S$ property, $P \rightarrow S$ is the predominant pattern in the trace.

In addition to buggy programs, sampling can also cause imperfect traces. Monitoring the complete execution of long running programs such as operating systems is impractical. In practice, traces typically sample partial execution by recording either the complete run-time data for a short period or randomly selected data for the whole execution. In either case, the trace does not capture the whole execution.

Finally, instrumentation tools have some limitations on the data they can capture. For example, an instrumentor might not be able to record argument values and return values. So, the execution trace can miss information such as the identity of a lock. When there are multiple instances of a lock, the identity

```
1       int i=0;
2       while(true){
3           lock.acquire();
4           // do something here
5           ……
6           if(++i>10)
7               break;
8           lock.release();
9       }
```

FIGURE 8.10: A hypothetical program that forgets to release a lock.

serves as a way to differentiate operations on the locks. If a trace does not have the identity of a lock or other ways to distinguish different locks, all calls to acquire or release different locks will appear to be the same events and may violate the Alternating property.

Detecting the Dominant Behavior

To deal with the imperfect traces, we adapt the algorithm from Figure 8.6 (page 256). The new algorithm decides what fraction of an execution trace satisfies a property template instead of just determining satisfaction of a property template. The new algorithm partitions the original trace into subtraces, decides whether each subtrace satisfies a pattern, and computes the fraction of the subtraces that satisfy a pattern.

We define a subtrace using a regular expression. We call this regular expression the *monitor template* and its finite state machine the *monitor FSM*. We call the finite state machine of a property template (e.g., Alternating) the *property FSM*. We use $P_{property\ template}$ to represent the percentage of a trace that satisfies a property template. For example, P_{AL} indicates the percentage of the subtraces that satisfy the Alternating template.

One intuitive definition of a subtrace is P^+S^+. In addition, the leading S events form a subtrace and so do the trailing P events. For example, we partition $PSPSPSPSPSPSPSPSPSP$ into 10 subtraces $PS, PS, PS, PS, PS, PS, PS, PS, PS, P$. The first nine subtraces satisfy the $P \rightarrow S$ property, whereas the last one does not. Therefore, 90% of the subtraces satisfy $P \rightarrow S$ and $P_{Alternating}$ is 0.90.

The new algorithm tracks one monitor FSM and one or more property FSMs. As with the original algorithm, it scans the execution trace twice. The two algorithms differ in how they update the states during the second scan. Figure 8.11 shows the pseudo-code of the new algorithm. When the new algorithm reads an event (line 2), it first updates all the property FSMs (line 3) in the same way as the original algorithm (lines 25-29 in Figure 8.6 on page 256). Then the new algorithm updates the state of the monitor FSM (line 4). If the monitor FSM reaches one of its accepting states (line 5), this indicates

```
1   while (not at the end of the trace)
2       read the next event from the trace
3       update the property FSMs
4       update the monitor FSM
5       if (the monitor FSM is in an accepting state)
6           Increase the counter of the monitor FSM by one
7           for each property FSM
8               if (the property FSM is in an accepting state)
9                   increase the counter of the property FSM by one
10              reset the property FSM to its start state
```

FIGURE 8.11: The approximate inference algorithm.

the end of a subtrace. The new algorithm increases the counter of the monitor FSM by one (line 6) and then checks whether the property FSMs reach their accepting states (line 7-8). If a property FSM is in an accepting state, this subtrace satisfies the property FSM and hence the new algorithm increases the property FSM's counter by one (line 9). Finally, the new algorithm resets the property FSMs to their starting states (line 10) before analyzing the next subtrace. Like the original algorithm, this algorithm has running time in $\Theta(NL)$ ans space complexity in $\Theta(N^2)$.

8.2.6 Property Selection

When processing a large trace that has many distinct events, our inference technique typically infers thousands of properties, which are too many to be effectively used in practice. Hence, one big challenge is to select a subset of the properties that are mostly interesting. An *interesting* property is a property for which developers are likely to make mistakes and violation of which would produce bad consequences. For example, we consider properties of critical system resources such as locks and transactions to be interesting. Such properties are important because violating them can have serious consequences such as causing system crashes [6, 31] and opening security vulnerability [18]. Next, we describe two heuristics for selecting interesting properties.

Static Call Graph Based Heuristic

One way to identify interesting properties is based on a program's static call graph [57]. The key observation is that a property is more likely to be interesting when the two events it involves are not reachable in the static call graph.

Figure 8.12(a) illustrates this idea. Suppose our inference technique infers two Alternating properties: KeSetTimer → KeSetTimerEx and ExAcquireFastMutexUnsafe → ExReleaseFastMutexUnsafe. In the first property, KeSetTimer is a wrapper of KeSetTimerEx. Therefore, whenever KeSetTimer is called, KeSetTimerEx must also be called. In the second property, ExAcquireFastMutexUnsafe

```
void KeSetTimer( ) {
   KeSetTimerEx( );
}
```

```
void X( ) {
   …
   ExAcquireFastMutexUnsafe(&m);
   …
   ExReleaseFastMutexUnsafe(&m);
   …
}
```

(a) A concrete example

```
A( ) {
   …
   B ( );
   …
}
```

```
X( ) {
   …
   C ( );
   …
   D ( );
   …
}
```

(b) Abstract form

FIGURE 8.12: Two scenarios of static call graph.

and ExReleaseFastMutexUnsafe do not call each other. Thus, their executions are asynchronous.

Figure 8.12(b) shows an abstract form of the two scenarios. $A \rightarrow B$ and $C \rightarrow D$ are Alternating properties. The second property is usually more interesting than the first one for two reasons. The first property represents a trivial relationship that can be easily discovered by constructing a static call graph. On the other hand, the second property captures two events that do not have an obvious static relationship. In addition, the second property captures a protocol between two functions, which developers are more likely to forget. The developers are obligated to call the pair of functions together. In contrast, the first property represents a delegation relationship between the two functions, which does not suggest any obligation on the developers.

Naming Similarity Heuristic

The second heuristic exploits the naming conventions used in many real systems. For example, Microsoft uses the Hungarian Notation [113]. The key observation behind the naming similarity heuristic is that a property is more likely to be interesting if the events it involves have similar names. For example, KeAcquireInStackQueuedSpinLock and KeReleaseInStackQueuedSpinLock only differ by one word and clearly appear to represent a desirable locking discipline relationship.

The naming similarity heuristic partitions the two events' names into words and then computes a similarity score of the two names. For a property $A \rightarrow B$, where A has w_A words, B has w_B words, and A and B have w words in common, this heuristic computes the similarity score of A and B as

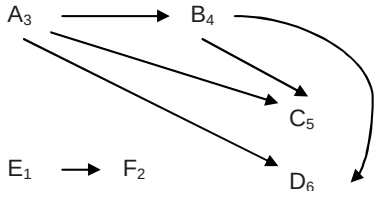


FIGURE 8.13: Alternating Chains.

$$Similarity_{AB} = \frac{2w}{w_A + w_B}$$

We distinguish words by capital letters or underscores. For example, KeAcquireInStackQueuedSpinLock has seven words, and so does KeReleaseInStackQueuedSpinLock (i.e., Ke, Acquire/Release, In, Stack, Queued, Spin, and Lock). Therefore, $w_A = w_B = 7$. There are six common words and so $w = 6$. As a result, the similarity score of these two names is 85.7%.

The naming similarity heuristic is especially effective for selecting properties that tend to involve events with similar function names. For example, locking disciplines and resource allocation/deletion protocols usually have a pair of functions with similar names. Although this heuristic does eliminate some interesting properties that involve events with very different names, our goal is to increase the density of interesting properties in the results. The experimental results (Section 8.3.3) demonstrate that the naming heuristic is effective for achieving this goal.

8.2.7 Chaining Method

The chaining method presents the inferred Alternating properties in a condensed form so that users can gain a better picture of how a system works. Next, we introduce a *property graph* that is a graph representation of a set of Alternating properties, prove several important properties of the property graph, and define the *chaining problem*. We prove the chaining problem is in NP-complete, and describe a brute force algorithm for solving the chaining problem. Despite being exponential in general, this algorithm performs well in our experiments due to the low density of property graphs for real programs.

Property Graph

We map a set of Alternating properties to a directed graph $G =< V, E >$, where V is a set of nodes and E is a set of edges. Each distinct event corresponds to a node in V. Therefore, $|V| = N$. An edge from node i to node j corresponds to an Alternating property $Event_i \rightarrow Event_j$. We call G the *property graph*. Figure 8.13 shows a property graph involving six events and six

Alternating properties (the subscripts correspond to the topological numbers explained later in this subsection).

If all the Alternating properties in a property graph have $p_{AL} = 1.0$, the property graph is a directed acyclic graph (DAG). In other words, for any i and j, if there is a path from node i to node j, there does not exist a path from node j to node i. To prove this, we first prove the following lemma: if there is a path from node i to node j, the first $Event_i$ must occur before the first $Event_j$ in the trace.

We prove the lemma by induction on l, the length of a path from node i to node j. The base case is $l = 1$. In this case there is an edge from node i to node j. According to the definition of the property graph, $Event_i \rightarrow Event_j$ is true. Therefore, according to the definition of the Alternating property, the first $Event_i$ must occur before the first $Event_j$ in the trace. For $k > 1$, assume that the lemma holds for all l where $1 \leqslant l < k$. Next, we prove the lemma when $l = k$. Suppose there is a path from node i to node j and this path's length is k. If the next to last node on this path is node p, there is a path from node i to node p and this path's length is $k - 1$. According to the induction hypothesis, the first $Event_i$ must occur before the first $Event_p$ in the trace. In addition, there is an edge from node p to node j. According to the induction hypothesis, the first $Event_p$ must occur before the first $Event_j$ in the trace. As a result, the first $Event_i$ must occur before the first $Event_j$ in the trace.

Using the above lemma, we prove by contradiction that a property graph is a DAG. If a property graph G is not a DAG, then G contains a cycle. There must exist two nodes, i and j, such that there exists a path P from i to j and a path P' from j to i. According to the above lemma, the existence of P implies that the first $Event_i$ must occur before the first $Event_j$ in the trace. In addition, the existence of P' implies the first $Event_j$ must occur before the first $Event_i$ in the trace, which is a contradiction. Hence, a property graph is a DAG.

When a property graph includes Alternating properties with $p_{AL} < 1.0$, it might still be a DAG. As explained later, our chaining algorithm first checks if a property graph (with properties whose $p_{AL} < 1.0$) has any cycles. Our algorithm only performs the chaining operation when the property graph is a DAG. The rest of this section assumes a property graph is a DAG.

A *topological number* of a node in a DAG is an integer such that if there exists an edge from node i to node j, then the topological number of node i is less than the topological number of node j [27]. Because a property graph is a DAG, we can sort the nodes based on their topological numbers. The subscript of each node in Figure 8.13 indicates its topological number.

An *alternating chain* is a subgraph $G' =< V', E' >$ of a property graph $G =< V, E >$, where $V' \subseteq V$ and $E' = \{(i, j) \mid i, j \in V' \text{ and } (i, j) \in E\}$, such that if the topological number of node i is less than the topological number of node j, then $(i, j) \in E'$. By definition, an edge in a property graph is a trivial alternating chain. For example, in Figure 8.13, the subgraph consisting of A and B is a trivial alternating chain. In addition, the subgraph consisting

of A, B, and C is an alternating chain, while the subgraph consisting of A, B, C, and D is not an alternating chain.

A maximal alternating chain is an alternating chain $G' =< V', E' >$ of a property graph $G =< V, E >$ such that $\forall i \in V - V'$, $G'' =< V'', E'' >$ is not an alternating chain, where $V'' = V' \cup \{i\}$ and $E'' = E' \cup \{(i, j) \mid j \in V' \text{ and } (i, j) \in E\} \cup \{(j, i) \mid j \in V' \text{ and } (j, i) \in E\}$. For example, in Figure 8.13, the subgraph consisting of A, B, and C and the subgraph consisting of E and F are maximal alternating chains.

The *chaining problem* is the problem that, given a property graph G and an integer k, determine whether there exists an alternating chain of k nodes in G. The *chain enumeration problem* is the problem that, given a property graph G, identify all the maximal alternating chains of G.

Chaining is in NP-Complete

Given any subgraph of a property graph, we can easily verify in polynomial time whether the subgraph is an alternating chain. Hence, the chaining problem is in NP.

We prove the NP-hardness of the chaining problem by reducing the clique problem to the chaining problem. The *clique problem* in graph theory states "for an undirected graph G and an integer k, does G have a complete subgraph that has k nodes" [77]. The clique problem is a well-known NP-Complete problem. Next we construct a polynomial-time reduction from the clique problem to the chaining problem.

Given an undirected graph G, we can convert all of its undirected edges to directed edges in polynomial time. We call the resulting directed graph G'. The conversion works by following the standard black-gray-white color depth-first-search (DFS) algorithm [27]. Figure 8.14 shows the transformation algorithm. Initially, all the nodes are marked as white (lines 7-8). Next, the algorithm checks each node and performs DFS for white nodes (lines 9-11). DFS is a recursive function that takes a node i to be explored as its argument (line 13). The node that is currently being explored is marked as gray (line 14). Next, all the adjacent nodes of i are explored (lines 16-17). If the adjacent node j is a white node, the algorithm converts the corresponding edge to a directed edge from i to j and continues to explore j (lines 19-22); if j is a gray node, the algorithm converts the corresponding edge to a directed edge from j to i (lines 23-25); if j is a black node, the algorithm does not do anything (lines 26-27). After all the adjacent nodes of i have been explored, node i is marked as black (line 29). The time complexity of the conversion algorithm is polynomial because DFS is in $\Theta(|V| + |E|)$.

Now we prove by contradiction that the resulting graph G' is a DAG. Suppose there is a cycle, $i, ..., j, i$, in G'. Without loss of generality, let us assume that i is the first node in the cycle that is explored during the conversion process. In other words, when i changes color from white to gray, the other nodes on the cycle are still white. According to the algorithm, there are two

```
1   int total;                  // the number of nodes
2   int undirected[ ][ ];   // total x total, an undirected graph with no self-loop
3   int directed[ ][ ];     // total x total
4   int color[ ];               // total
5
6   void undirected2directed ( )
7       for ( int i = 0; i < total; i++ )
8           color[ i ] = WHITE;
9       for ( int i = 0; i < total; i++ )
10          if (color [ i ] == WHITE)
11              DFS ( i );
12
13  void DFS ( int i )
14      color [ i ] = GRAY;
15
16      for ( int j = 0; j < total; j++ )
17          if ( undirected [ i ][ j ] == CONNECTED )
18              switch ( color [ j ] )
19                  case WHITE:
20                      directed [ i ][ j ] = CONNECTED;
21                      DFS ( j );
22                      break;
23                  case GRAY:
24                      directed [ j ][ i ] = CONNECTED;
25                      break;
26                  case BLACK:
27                      break;
28
29      color [ i ] = BLACK;
```

FIGURE 8.14: Algorithm for transforming an undirected graph to a DAG.

ways the edge from j to i can be created. First, when the adjacent nodes of j are explored (hence, j is gray), i (one of j's adjacent nodes) is white (lines 19-22). This case is impossible due to our assumption that i is the first node on the cycle that has gray color. Second, when the adjacent nodes of i are explored, j is gray too (lines 23-25). So the algorithm creates an edge from j to i (line 24). This also contradicts our assumption that j is white when i is gray. Therefore, it is impossible to have an edge from j to i in G', which contradicts the last edge on the cycle.

Given an instance of the clique problem (determining whether an undirected graph G has a complete subgraph with k nodes), the above transformation converts the clique problem to an instance of the chaining problem (determining whether the resulted DAG G' has an alternating chain with k nodes). Next we show that if we have a solution to the chaining problem, we can use it to solve the clique problem. We use $[i, j]$ to represent a directed edge from i to j and (i, j) to represent an undirected edge between i and j. Suppose our algorithm determines that G' has an alternating chain, $X' =< V_{X'}, E_{X'} >$ with k nodes. The counterpart of X' in G is $X =< V_X, E_X >$, where $V_X = V_{X'}$ and $E_X = \{(i, j) | i, j \in V_X, [i, j] \text{ or } [j, i] \in E_{X'}\}$. According to the definition of Alternating Chain, $\forall i, j \in V_{X'}$, $[i, j]$ or $[j, i] \in E_{X'}$. Therefore, $\forall i, j \in V_X$, $(i, j) \in E_X$. Hence, X is a clique with k nodes in G.

Thus, the clique problem reduces to the chaining problem so the chaining problem is also in NP-hard. We already proved the chaining problem is in NP, so the chaining problem is NP-Complete.

The Chaining Algorithm

Assume $P \neq NP$, no algorithm can have a better asymptotic worst-case performance than a brute force algorithm. The performance of a brute force algorithm is highly dependent on the structure of the property graph. For a directed graph with n nodes and m edges, we compute its edge density as $2m/n^2$. Intuitively, a brute force algorithm generally performs better on a sparse property graph than a dense property graph because there are fewer edges (and so much fewer combinations of edges) to try in a sparse property graph. In our experiments, all of our property graphs are very sparse with densities around 10%. Next we present a brute force algorithm whose worst-case performance is exponential. The running time scales with the edge density, which we have found to be low in practice. We have applied our chaining algorithm to analyze a property graph with 91 nodes and 490 edges in less than one minute.

Given a property graph G, we can convert all its directed edges to be undirected. We call the resulted graph $G_{undirected}$. If a subgraph C of G is an alternating chain, then the corresponding subgraph $C_{undirected}$ of $G_{undirected}$ must be a clique, and vice versa. Finding all the maximal alternating chains in G, therefore, can be solved by finding all the maximal cliques in $G_{undirected}$.

Our chaining algorithm first identifies all the connected components in

```
1    for each cc in G_undirected
2        chaining ( cc );
3
4    chaining ( cc )
5        Vector worklist = new Vector();
6        add all the edges of cc to worklist;
7        while ( worklist is not empty )
8            clq = worklist.remove ( 0 );
9            boolean cannotExpand = true;
10           for each node i of cc
11               if ( i is in clq )
12                   continue;
13               if ( there is an edge between i and each node of clq )
14                   newclq = clq U { i };
15                   cannotExpand = false;
16                   for each element x of worklist
17                       if ( x is a subset of newclq )
18                           remove x from worklist;
19                   insert newclq to the beginning of worklist;
20                   break;
21           if ( cannotExpand )
22               print out clq in topological order;
```

FIGURE 8.15: The chaining algorithm.

$G_{undirected}$ using a depth-first-search [27]. Then the algorithm identifies the maximal cliques in each connected component. To convert a maximal clique to an alternating chain, we output the clique in topological order. Figure 8.15 shows the chaining algorithm.

The chaining algorithm is a work-list algorithm. The algorithm first creates a worklist (line 5). The worklist stores the cliques that will be examined. Each clique is represented as a set of nodes. The worklist is a set of sets of nodes, initially containing one set for each edge in the graph consisting of the endpoints of that edge (line 6). The algorithm removes the first clique, clq, from the worklist (line 8). For each node of the current connected components, the algorithm first tests if it is in clq (lines 11-12). For a node i that is not in clq, the algorithm tests if there is an edge between i and each node of clq (line 13). If so, the algorithm expands clq to newclq (line 14), removes all the cliques in worklist that are subset of newclq (lines 15-18), and inserts newclq to the beginning of worklist (line 19). The boolean variable, cannotExpand, indicates whether clq can be expanded (lines 9 and 15). If cannotExpand is true after processing clq, the algorithm outputs clq according to the topological order in the directed property graph (lines 21-22).

Next we analyze the complexity of our chaining algorithm. Suppose a connected component, cc, has n nodes and m maximal cliques. Suppose the ith maximal clique has n_i nodes. In order to construct the maximal cliques, the while loop (lines 8-22) executes $\sum_{i=1}^{m} n_i$ times because the clique grows

```
Chain #1
Enter:Buffer.add():[main] →
Exit:Buffer.add():[main]

Chain #2
Enter:Buffer.take():[Thread-1] →
Exit:Buffer.take():[Thread-1]

Chain #3
Enter:Producer.main():[main] →
Enter:Consumer.run():[Thread-1] →
Enter:Buffer.stop():[main] →
Exit:Buffer.stop():[main] →
Exit:Producer.main():[main] →
Exit:Consumer.run():[Thread-1]
```

FIGURE 8.16: Alternating chains for the Producer-Consumer program.

one node in each loop. The for loop (lines 11-20) executes n_i times. During each for loop, the algorithm needs to check up to n_i edges (line 13). In addition, when a larger clique, newclq, is formed, the algorithm needs to remove redundant cliques from the worklist (lines 16-18). A clique, x, is redundant if it is a subset of newclq. When newclq has p nodes, in worst-case, the number of redundant cliques is in $\Theta(C_{n_i}^{p-1})$. Therefore, the for loop is in $\Theta(n_i(n_i + C_{n_i}^{p-1}))$. To construct the ith maximal clique, the algorithm's complexity is in $\Theta(\sum_{p=1}^{n_i}(n_i(n_i + C_{n_i}^{p-1}))) = \Theta(n_i 2^{n_i} + n_i^3)$. Therefore, the worst-case complexity of our chaining algorithm is still exponential. The complexity of the chaining algorithm varies by the density of the property graph. In practice, the density of the property graph tends to be small (typically around 10% in our experiments). Therefore, the performance of our algorithm is acceptable in practice.

Figure 8.16 shows the three chains constructed for the Producer-Consumer program. The longest chain (#3) has six events and represents an important property: after the Producer calls the stop method, the Consumer eventually stops. Each of the other two chains has only two events and is uninteresting because these Alternating properties correspond to the trivial fact that entering a method alternates with exiting the method.

8.2.8 Perracotta

We adapted a Java instrumentation tool and implemented the inference engine in a prototype tool called Perracotta. In addition to implementing the inference algorithm, Perracotta also implements the two selection heuristics, the chaining method, and the context-handling techniques.

Instrumentation

To instrument Java programs, we adapted the Java Runtime Analysis Toolkit (JRat) [75]. JRat has two important components: an instrumentor and a runtime system. The instrumentor component uses the Byte Code Engineering Library (BCEL) to parse and insert hooks into Java bytecode [7]. When an instrumented Java application executes, the hooks generate events for method entrances, method exits, and exceptions. The JRat runtime system processes the events by delegating them to one or more handlers. Different handlers process the events in different ways. JRat provides an event handler Service Provider Interface (SPI). Users can develop their own handlers by implementing this SPI and can configure the handlers that the JRat runtime system uses by either setting an environment variable or supplying a configuration file. We developed an event handler for collecting method execution traces. For each Java method, our handler records its entrance, exit, signature, arguments, return value, and any exceptions generated. If an argument is of a primitive type, the handler outputs its value. If an argument is of an object type, the handler outputs its hashcode as its object ID.

For small C programs, we manually instrument the source code to monitor function calls. For Windows kernel, we use a Vulcan-based instrumentor [117]. This instrumentor works on x86 binaries and can monitor function calls and thread information. However, this instrumentor cannot monitor argument values and return values.

Inference Engine

Perracotta implements the inference engine and post-processing components. Perracotta is implemented in 12000 lines of Java code. Perracotta can be run in two modes: the strictest pattern mode for inferring the strictest pattern and the approximate mode for inferring properties whose satisfaction ratio is greater than a threshold between 0.0 and 1.0. In addition, it also has an interface that accepts user-specified templates and determines the satisfaction ratios. To implement the static call graph based heuristic for Java, Perracotta has a module for computing the static call graph. For C/C++, it accepts static call graph as a textual file. In addition, Perracotta can eliminate uninteresting properties based on the naming similarity based heuristic. Furthermore, Perracotta implements the chaining method for the Alternating properties as presented in Section 8.2.7.

Perracotta also provides several utility programs that slice traces as described in Section 8.2.4. The *ThreadSlicer* slices the trace into subtraces based on the thread identities. Similarly, the *ObjectSlicer* slices the trace into subtraces based on the object identities.

8.3 Inference Experiments

This section presents experiments applying Perracotta to several programs. We evaluate the usefulness of the inferred properties in revealing important information of a program. Section 8.4 investigates other uses of the inferred properties including program verification (Section 8.4.1) and program differencing (Section 8.4.2).

Complex systems are hard to understand in that it is challenging to recognize delocalized plans [83]. *Delocalized plans* are "programming plans realized by lines scattered in different parts of the program." [83]

To test the hypothesis and also evaluate the scalability, accuracy, and cost of our technique, we conducted several case studies on a wide range of programs:

(a) *Bus Simulator*, a collection of student submissions for an assignment of implementing a multi-threaded C program in a graduate course taught at the University of Virginia.

(b) *Daisy*, a prototype implementation of a Unix-like file system in Java. Daisy was developed as a common testbed for different program verification tools [30].

(c) *OpenSSL*, a C implementation of the Secure Sockets Layer (SSL) specification [99, 115]. Our experiment focuses on its handshake protocol.

(d) *JBoss*, a Java application server conforming to the J2EE specification [72, 74]. Our experiment focuses on its transaction management module.

(e) *Microsoft Windows kernel APIs*, a set of about 1800 functions written in C or C++. These functions are the foundation of the Windows operating system. Our experiment is on Windows Vista.

Table 8.2 summarizes the characteristics of these six testbeds. Bus Simulator and Daisy are small prototype programs. They are good for proof-of-concept experiments. The other three programs are complex and widely used. They are valuable for understanding the strength and weakness of our technique when it is applied to real systems. Our testbeds differ in the quality of specification available. The Bus Simulator comes with instructor's English description of a list of properties a valid implementation ought to have. Daisy does not have any temporal specification except a list of properties in English provided by one of its developers. OpenSSL has an extensive specification, the SSL specification [99, 115]. The SSL specification includes a detailed description as well as a finite state automaton of the handshake protocol. The JBoss application server implements the J2EE specification [72, 74]. In particular, its transaction management module implements the Java Transaction

TABLE 8.2: Characteristics of Testbeds.

Name	Category	Language	Size	Maturity	Temporal Specifications Available?	Additional Experiments
Bus Simulator	Student multi-threaded program	C/C++	259	Prototype	Yes, in English	Program differencing
Daisy	Unix-like file system	Java	2K	Prototype	Limited	Program verification
OpenSSL (Handshake Protocol)	Network protocol	C	32K (418)[1]	Production	Yes, SSL specification in English and FSM	Program differencing
JBoss (Transaction Management)	Network middleware	Java	1M (7K)[2]	Production	Yes, JTA specification in English and FSM	
Windows Vista Kernel APIs	OS kernel	C/C++	50M (1800 APIs)	Production	Limited, MSDN and MS internal document including SLAM	Program verification

1. In OpenSSL version 0.9.7d, the implementation of the SSL specification (i.e., **.c* and **.h* files in the *ssl* directory) has thirty-two thousand lines and the implementation of the SSL handshake protocol on the server's side (i.e., the *ssl3_accept* function in the *s3_srvr.c* file) has 418 lines.

2. In JBoss version 4.0.2, all the **.java* files have one million lines. The transaction management module (i.e., all the **.java* files belonging to the *org.jboss.tm* package) has seven thousand lines.

API (JTA) specification [76], which has an extensive English description and an object interaction diagram illustrating the temporal behavior of the components participating in a transaction. For the Windows kernel APIs, some of their temporal rules are documented either publicly in the MSDN library or privately in some internal documents. These documented rules are, however, by no means complete as our experiment discovered many undocumented important rules. We include two systems with extensive specifications (OpenSSL and JBoss) so that we can compare the inferred properties against the existing specifications. These existing specifications serve as a guideline of what properties are important and interesting, without which it would be much more difficult to tell whether the inference approach produces useful results.

Next, Sections 8.3.1 to 8.3.3 present the inference results for all the testbeds except OpenSSL and Bus Simulator. We defer the presentation of the experiments on OpenSSL and Bus Simulator to Section 8.4.2 because the focus of these experiments is on evaluating the usefulness of the inferred properties in program differencing. Our experimental results strongly support that Perracotta can be useful for program understanding. Perracotta discovered interesting temporal properties for all the testbeds. For the Windows kernel, Perracotta inferred 56 interesting properties, many of which were undocumented. For JBoss, Perracotta inferred a 24-event finite state machine that was consistent with the JTA specification. Many of the JBoss properties represent delocalized plans as the events cross multiple "distant" modules of the target systems. Discovering these delocalized plans is valuable because they can be hard to discover by manual inspection and violating them when modifying the system could introduce serious errors.

8.3.1 Daisy

Daisy is a prototype Unix-like file system implemented in 2000 lines of Java code [30]. Daisy's architecture has four layers as shown in Figure 8.17. At the bottom, Daisy emulates the hard drive using a RandomAccessFile (RAF) object. Above it, the disk layer abstracts the hard drive into byte streams. The next layer abstracts the byte streams into blocks. The top layer provides an interface for files and directories.

We used JRat to monitor the invocation of all Daisy methods except those inherited from the Object class (e.g., toString). We created a wrapper for the Java RandomAccessFile class so that JRat can monitor its methods. JRat recorded a method's signature, thread, this object, and arguments.

To execute Daisy, we adapted the test harness in the Daisy distribution. Our test harness, DaisyTest, takes four parameters: F, the number of files to be created initially, T, the number of threads to be created, N, the number of iterations each thread executes, and R, the seed for the random number generator. The main thread of DaisyTest first creates F files and T threads. Next, each child thread makes a sequence of N calls to randomly selected

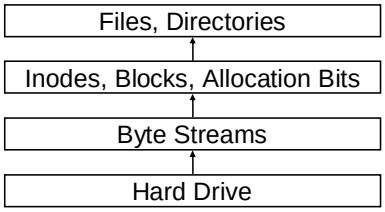


FIGURE 8.17: Daisy's System Architecture.
Daisy emulates the hard drive through a Java RandomAccessFile object.

methods of the DaisyDir class (one of read, write, set_attr, or get_attr). These methods are invoked with arguments randomly selected within the valid range.

Inference Results

We ran DaisyTest with $F = 5$, $T = 5$, $N = 15$, and $R = 0$. This execution produced a trace of about 70000 events. We used Perracotta to slice the original trace by threads and obtained six subtraces (five for the child threads and one for the main thread). We ran Perracotta in approximate mode with 0.70 as the acceptance threshold for p_{AL}. Our analysis only considered the 40 distinct events that occurred more than 10 times in the trace. Perracotta inferred 70 properties, 52 of which had a satisfaction ratio less than one. We applied Perracotta's chaining method to infer nine Alternating Chains.

The six shortest chains, with length from one to three events, are uninteresting because they correspond to wrapper functions. The other three chains also contain uninteresting edges due to wrapper functions. Next we applied Perracotta's call graph based heuristic to eliminate these wrapper properties and got eight properties.

Several properties provide insight into the temporal behaviors of Daisy. For example, DaisyDisk.readAllocBit $\rightarrow$ DaisyLock.relb ($p_{AL} = 0.97$) indicates that reading the allocation bit of a block (DaisyDisk.readAllocBit) often alternates with releasing the lock on the block (DaisyLock.relb). Because these two methods are not eliminated by the call graph based heuristic, they represent an interesting pair of asynchronous operations. These two methods do not necessarily alternate because DaisyLock.relb is called in several places (e.g., Daisy.read and Daisy.write) where DaisyDisk.readAllocBit is not called. Another interesting property is LockManager.acq $\rightarrow$ LockManager.rel ($p_{AL} = 0.86$) that captures an important locking relationship. The satisfaction ratio of this property is less than 1.0 because the traces are not sliced by objects.

Next, we used Perracotta to slice the this object and a method's first argument. Perracotta inferred two properties with $p_{AL} = 1.0$: Mutex.acq $\rightarrow$ Mutex.rel and LockManager.acq $\rightarrow$ LockManager.rel. Slicing on other arguments did not lead Perracotta to infer more properties. Object slicing missed some use-

ful properties that involve more than one object such as LockManager.acq → Mutex.rel.

We also ran Perracotta with the two-effect-alternating and the two-cause-alternating patterns (Section 8.2.4). Perracotta inferred an important property: RAF.seek → RAF.readByte | RAF.writeByte. We found a race condition in Daisy that violates this property using the Java PathFinder model checker (Section 8.4.1).

In summary, the inferred properties represent interesting temporal behaviors of Daisy. Several of the inferred properties such as DaisyDisk.readAllocBit → DaisyLock.relb involve more than one class, which indicate delocalized plans that would be useful to aid programmers in understanding the system.

8.3.2 JBoss Application Server

An *application server (AS)* is middleware that provides important services such as transactions, security, and caching for running web applications [72]. A web application built upon an application server reuses these well-tested services and components.

A *Java application server* is an application server that runs on a Java virtual machine [72]. The J2EE specification defines the interface between a web application and a Java application server [72]. JBoss is open-source and is currently one of the most widely used Java application servers [74].

We are particularly interested in the APIs of the transaction management service because a transaction occurs in multiple stages with certain temporal ordering constraints. The Java Transaction API (JTA) specification defines the interfaces between a transaction manager and the other participants in a distributed transaction system: the application, the resource manager, and the application server [76]. The JTA specification has an object interaction diagram [51] as an illustration of how an application server may handle a transactional connection request from an application (note that the diagram is just one typical scenario, but not a specification) [76]. An application server starts a transaction by first calling the begin method of the transaction manager (TM). Next the AS tries to get a transactional resource from the resource adapter (RA). The AS calls the enlistResource method to declare its ownership of a resource. Then the application does its work. To finish a transaction, the AS calls the delistResource method to release its ownership of the corresponding resource and then commits the transaction. The transaction commission follows a two-stage commit protocol that first prepares and then commits the transaction.

Inference Results

We obtained the source code of the JBoss Application Server version 4.0.2 from *www.jboss.org* (the latest one at the time of our experiments). The source code distribution included about 4000 fully automated regression test cases.

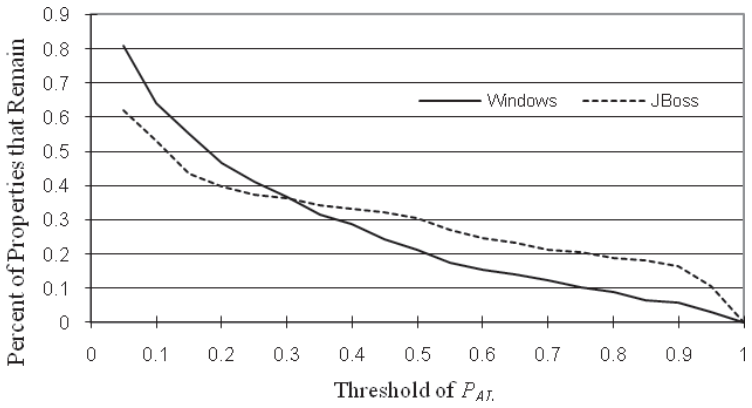


FIGURE 8.18: Inferred properties versus the acceptance threshold for p_{AL}.

We used JRat to instrument all method invocations of the transaction management module (i.e., all classes in the org.jboss.tm package) and ran the regression test suite. After dropping events that occurred fewer than 10 times, the execution trace contained 2.5 million events with 91 distinct events (we only monitored the entrance events). Perracotta analyzed the trace in 80 seconds on a machine with one 3GHz CPU, 1GB RAM, and Windows XP Professional.

Figure 8.18 shows the percentage of all instantiations of the Alternating pattern with p_{AL} greater than an acceptance threshold that increases from 0 to 1. The other line is for the Windows experiments described in Section 8.3.3. We arbitrarily picked 0.9 as the acceptance threshold to select properties. The initial result had 490 properties, too many to inspect manually. Next, the chaining method converted the properties to 17 chains. We applied the chaining method before applying other heuristics because other heuristics might prevent a long chain from being formed. Then we pruned the results by applying the static call graph based heuristic, which reduced the number of chains to the 15 chains shown in Table 8.3.

Comparison with JTA Specification

The longest chain (Figure 8.19) has 24 events including not only the public methods declared in the JTA specification but also private methods internal to the implementation of JBoss. After omitting the private methods, we obtain the shorter chain shown in Figure 8.20. The *TxManager* and *TransactionImpl* classes implement the JTA *TransactionManager* and *Transaction* interfaces respectively. This chain is almost identical to the object interaction diagram in the Java Transaction API (JTA) specification except that Perracotta does not infer the alternating relationship between enlistResource and delistResource. This is because whenever enlistResource is called, either delistResource or com-

TABLE 8.3: The JBoss AS TM Alternating Chains.

No	JBoss Transaction Management Module Alternating Chains
1	org.jboss.tm.TransactionImpl.lock org.jboss.tm.TransactionImpl.unlock
2	org.jboss.tm.TxManager.setRollbackOnly org.jboss.tm.TxManager.rollback
3	org.jboss.tm.TxManager.setTransactionTimeout org.jboss.tm.TxManager.suspend
4	org.jboss.tm.TxUtils.isActive org.jboss.tm.TxManager.commit
5	org.jboss.tm.usertx.server.UserTransactionSessionImpl.getInstance org.jboss.tm.usertx.server.UserTransactionSessionImpl.getTransactionManager
6	org.jboss.tm.GlobalId.computeHash org.jboss.tm.XidFactory.extractLocalIdFrom
7	org.jboss.tm.XidImpl.getLocalId org.jboss.tm.TransactionImpl.getGlobalId
8	org.jboss.tm.TransactionLocal.storeValue org.jboss.tm.TxManager.storeValue
9	org.jboss.tm.XidImpl.getFormatId org.jboss.tm.XidImpl.getGlobalTransactionId org.jboss.tm.XidImpl.getBranchQualifier
10	org.jboss.tm.TransactionImpl.checkWork org.jboss.tm.TransactionImpl.rollbackResources org.jboss.tm.TxManager.incRollbackCount
11	org.jboss.tm.TransactionLocal.getValue org.jboss.tm.TxManager.getValue
12	org.jboss.tm.TransactionPropagationContextUtil.getTPCFactoryClientSide org.jboss.tm.TxManager.getTransactionPropagationContext org.jboss.tm.TxManager.getTransaction
13	org.jboss.tm.TransactionLocal.initialValue org.jboss.tm.TransactionLocal.set org.jboss.tm.TransactionLocal.containsValue org.jboss.tm.TxManager.containsValue
14	org.jboss.tm.TransactionImpl.associateCurrentThread org.jboss.tm.TransactionImpl.commit org.jboss.tm.TxManager.resume org.jboss.tm.TxManager.suspend
15	See Figure

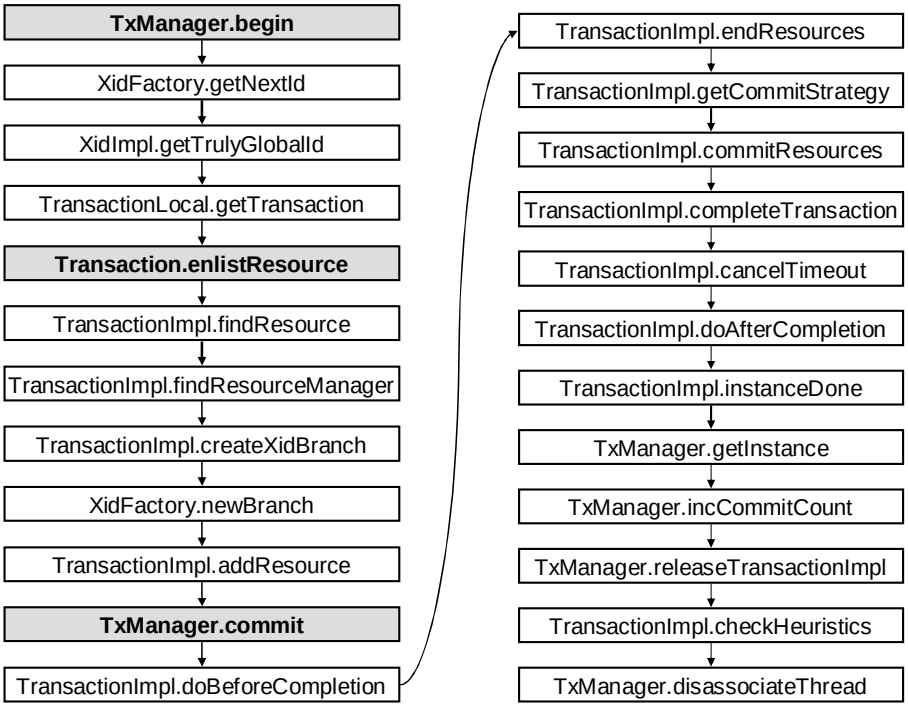


FIGURE 8.19: Alternating Chain for the JBoss AS TM module.
Grayed events are public methods.

mitResources must be called. Therefore, a resource does not have to be delisted. As shown in Figure 8.20, Perracotta incorrectly infers enlistResource → commitResources as it is the dominant behavior in the trace.

The longest chain reveals more than just how the public APIs interact. It also provides insight into the internal implementation such as starting and committing a transaction, which would help new developers understand JBoss.

In summary, Perracotta successfully infers a complex finite state machine that is consistent with the JTA specification. This demonstrates that Perracotta can help programmers understand a real legacy system. Suppose there is no specification available for the transaction management module, it would have been a great challenge for programmers to discover its temporal behaviors by hand because these properties cross the boundary of many modules and represent a non-trivial delocalized plan. These properties would also be difficult for static analysis to infer because JBoss extensively uses polymorphic interfaces, pointers, and exception handlers. Precisely handling all these features is still beyond the state-of-the-art of static analysis. In addition, the static call graph based heuristic and the chaining method are helpful for reducing the number of properties and presenting the results. On the other hand, the naming similarity based heuristic is not very useful in the experiment since

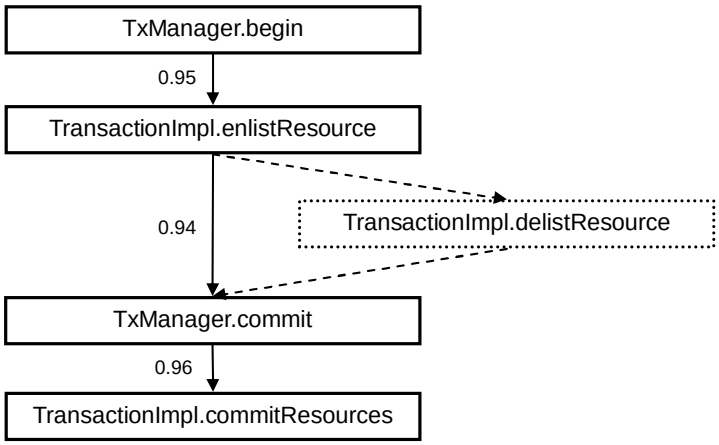


FIGURE 8.20: Alternating Chain for the public APIs of the JBoss AS TM module.

being a Java program, JBoss has few methods that implement basic resource management and locking disciplines.

8.3.3 Windows

In this experiment, we applied Perracotta to infer API rules for the latest (as of summer 2005) kernel (*ntoskrnl.exe*) and core components (*hal.dll* and *ntdll.dll*) of Windows Vista. Perracotta not only inferred many documented properties, but also found several important but undocumented properties. We checked 10 arbitrarily selected properties using the ESP verifier, which found a serious deadlock bug in the NTFS file system (see Section 8.4.1).

Inference Results

We obtained 17 execution traces from a developer in the Windows core development team. This developer instrumented APIs of the Windows kernel and core components and collected these traces by running some typical Windows applications (e.g., Windows MediaPlayer, Windows MovieMaker). He collected these traces mainly for performance tuning and debugging. We did not have any control of generating execution traces because these traces had already been generated before we started our experiments. In particular, the execution traces included the thread context information, but did not include the values of function arguments.

The lengths of the traces ranged from about 300,000 to 750,000 events, for about 5.8 million total events. The number of distinct events in each trace varied from around 30 to 1300. On average, each execution trace had about 500 distinct events. Perracotta analyzed all traces in 14 minutes on a machine

TABLE 8.4: Impact of selection heuristics.

Prop	Name Similarity (>0.5)		Call Graph Only				Both	
	Prop	Reduction	Unreachable	Unknown	Total	Reduction	Prop	Reduction
7611	185	97.6%	3280	3326	6606	23.5%	142	98.13%

running Windows XP Professional with one 3GHz CPU and 1GB RAM. As with JBoss, we set the acceptance threshold for p_{AL} to 0.90. Perracotta inferred 7611 properties, too many to manually inspect. We randomly selected 200 inferred properties and found that only 2 of them are interesting. So we applied the call graph and naming similarity heuristics to select the interesting properties. We used the static call graph of ntoskrnl.exe generated by ESP [31]. After using both selection heuristics, 142 properties remained.

Table 8.4 summarizes the impact of the two heuristics. The naming similarity based heuristic alone reduces the number of properties from 7611 to 185, which is a 97.6% reduction. Although the static call graph based heuristic has a smaller reduction rate than the naming similarity, it is still very helpful for reducing the number of properties as indicated by the 23.5% reduction.

We manually inspected the 142 properties and identified the 56 (40%) interesting ones shown in Table 8.5. The properties we deemed interesting are relevant to either resource allocation/deallocation or locking discipline. The heuristics increased the density of interesting properties and therefore were effective. The approximation algorithm is essential for detecting useful properties such as ObpCreateHandle $\rightarrow$ ObpCloseHandle and ExCreateHandle $\rightarrow$ ExDestroyHandle that otherwise would be missing.

We compared the 56 inferred properties against those checked by the Static Driver Verifier (SDV), and found that Perracotta inferred four of the 16 sequencing properties that the SDV checked [116]. For example, KeAcquireQueuedSpinLock $\rightarrow$ KeReleaseQueuedSpinLock is one of the four properties. Perracotta missed seven properties such as KeAcquireSpinLock $\rightarrow$ KeReleaseSpinLock that the SDV checked because our execution traces did not include those events. Perracotta missed five other properties that the SDV checked because the property templates could not express them. For example, our property templates cannot represent the property that an event only happens once. Therefore, Perracotta cannot infer the property that IoInitializeTimer is called only once.

Perracotta also inferred two important properties KiAcquireSpinLock $\rightarrow$ KiReleaseSpinLock and KfAcquireSpinLock $\rightarrow$ KfReleaseSpinLock that the SDV did not check because these functions were internal to Windows and therefore were invisible to the device driver developers.

The Windows experimental results are encouraging. The inferred properties capture critical rules which Windows developers are expected to follow when using the Windows kernel and other core components. Perracotta's approximate inference algorithm effectively handles imperfect traces. In all the

TABLE 8.5: Selected properties inferred for Windows.
Properties in gray shade are neither documented anywhere in MSDN nor checked by SDV.

P_{AL}	Property
1.00	ExAcquireFastMutex→ExReleaseFastMutex
1.00	ExAcquireRundownProtectionCacheAwareEx→ ExReleaseRundownProtectionCacheAwareEx
1.00	HMFreeObject→HMAllocObject
1.00	HvpGetCellMapped→HvpReleaseCellMapped
1.00	IoAcquireVpbSpinLock→IoReleaseVpbSpinLock
1.00	KeAcquireQueuedSpinLock→KeReleaseQueuedSpinLock
1.00	KefAcquireSpinLockAtDpcLevel→KefReleaseSpinLockFromDpcLevel
1.00	KeInitThread→KeStartThread
1.00	KeSuspendThread→KeResumeThread
1.00	KfAcquireSpinLock→KfReleaseSpinLock
1.00	KiAcquireSpinLock→KiReleaseSpinLock
1.00	LdrLockLoaderLock→LdrUnlockLoaderLock
1.00	MiMapPageInHyperSpace→MiUnmapPageInHyperSpace
1.00	MiSecureVirtualMemory→MiUnsecureVirtualMemory
1.00	MmSecureVirtualMemory→MmUnsecureVirtualMemory
1.00	NtfsAcquireFileForCcFlush→NtfsReleaseFileForCcFlush
1.00	NtGdiDdGetDC→NtGdiDdReleaseDC
1.00	NtUserBeginPaint→NtUserEndPaint
1.00	ObpAllocateObjectNameBuffer→ObpFreeObjectNameBuffer
1.00	PopAcquirePolicyLock→PopReleasePolicyLock
1.00	RtlAcquirePebLock→RtlReleasePebLock
1.00	RtlAcquireSRWLockExclusive→RtlReleaseSRWLockExclusive
1.00	RtlActivateActivationContext→RtlDeactivateActivationContext
1.00	RtlDeleteTimer→RtlCreateTimer
1.00	RtlLockHeap→RtlUnlockHeap
1.00	RtlpAllocateActivationContextStackFrame→RtlpFreeActivationContextStackFrame
1.00	RtlpAllocateUserBlock→RtlpFreeUserBlock
1.00	RtlpFindFirstActivationContextSection→RtlFindNextActivationContextSection
1.00	RtlValidSid→RtlCopySid
1.00	SeCaptureSid→SeReleaseSid
1.00	SeLockSubjectContext→SeUnlockSubjectContext
1.00	xxxBeginPaint→xxxEndPaint
0.99	ObpCreateHandle→ObpCloseHandle
0.99	_GetDC→_ReleaseDC
0.99	RtlpRemoveListLookupEntry→RtlpAddListLookupEntry
0.99	GreLockDisplay→GreUnlockDisplay
0.99	RtlActivateActivationContextUnsafeFast→ RtlDeactivateActivationContextUnsafeFast
0.98	CmpRemoveFromDelayedClose→CmpAddToDelayedClose
0.98	KeAcquireInStackQueuedSpinLock→KeReleaseInStackQueuedSpinLock
0.98	SeCreateAccessState→SeDeleteAccessState
0.98	KeAcquireInStackQueuedSpinLockRaiseToSynch→ KeReleaseInStackQueuedSpinLockFromDpcLevel
0.97	IoAllocateIrp→IoFreeIrp
0.96	CmpLockRegistry→CmpUnlockRegistry
0.96	ObAssignSecurity→ObDeassignSecurity
0.96	VirtualAllocEx→VirtualFreeEx
0.95	ExCreateHandle→ExDestroyHandle
0.95	ExpAllocateHandleTableEntry→ExpFreeHandleTableEntry
0.95	CmpFreeDelayItem→CmpAllocateDelayItem
0.94	ExInitializeResourceLite→ExDeleteResourceLite
0.94	RtlEnterCriticalSection→RtlLeaveCriticalSection
0.93	MiGetPreviousNode→MiGetNextNode
0.92	RtlValidAcl→RtlCreateAcl
0.92	ObFastReferenceObject→ObFastDereferenceObject
0.92	PsChargeProcessPoolQuota→PsReturnSharedPoolQuota
0.91	EtwpInitializeDll→EtwpDeinitializeDll
0.90	IoFreeMdl→IoAllocateMdl

experiments, Perracotta inferred interesting properties whose p_{AL} was below 1.0. These properties involve operations on key system resources such as locks and file handles.

Violating these rules could result in system crashes that would be difficult to diagnose. Furthermore, many of the inferred properties are not properly documented. In our personal communication with several Windows developers, they expressed the need for a tool to help them learn the important rules of Windows such as the ones Perracotta infers. For example, two summer interns in the Windows group were assigned a task of writing a program using a type of queue in Windows kernel. Unfortunately they could not find any documentation about this queue. As a result, they had to manually look at programs to distill the rules about how to use the queue, which was very time-consuming.

8.4 Using Inferred Properties

This section presents experiments using the inferred properties. Section 8.4.1 describes experiments using a verification tool to check the inferred properties. Section 8.4.2 describes experiments using the inferred properties to identify differences among multiple versions of a program.

8.4.1 Program Verification

Program verification techniques try to decide whether or not a program conforms to a specification. Although verifying any non-trivial property is undecidable [110], recently researchers have achieved much practical success in verifying important generic safety properties [15] as well as application-specific properties [6,31,45,120]. For example, Microsoft uses the Static Driver Verifier (also known as the SLAM project) to check device drivers against a list of driver related rules. The adoption of such tools, however, is limited by the unavailability of property specifications.

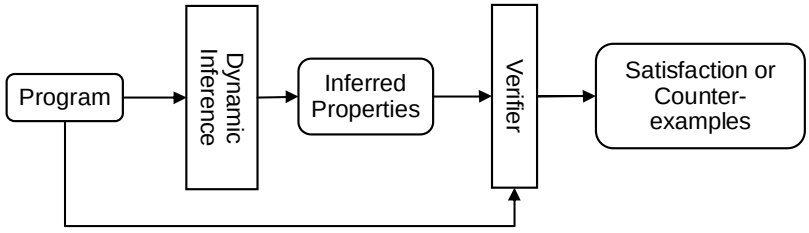


FIGURE 8.21: Use inferred properties in program verification.

We present two case studies on the Daisy file system and the Microsoft Windows kernel respectively. Figure 8.21 illustrates this process. In particular, we used the ESP verifier to check the Windows kernel properties and the Java PathFinder model checker to verify the Daisy properties [31, 120].

Daisy

Section 8.3.1 introduced Daisy, a prototype Unix-like file system [30]. We applied Java PathFinder (JPF) [120] to verify the 22 inferred properties show in Table 8.6. Java PathFinder is an explicit-state model checker for Java programs [120]. It checks deadlocks, race conditions, unhandled exceptions, and user-specified assertions. It can scale to a program of about 10000 lines. Upon finding a violation of a property, it produces an execution path illustrating the problem. It has been used to find complex concurrent bugs in real systems [120].

TABLE 8.6: Results of checking Daisy properties with Java PathFinder.

No	Causing Event (*P*)	Effect Event (*S* or *S*|*T*)	JPF Found Violation?
1	Daisy.alloc()	DaisyDisk.writeAllocBit(blockno, ...)	
2	Daisy.creat()	Daisy.alloc()	
3	Daisy.creat()	Daisy.ialloc()	
4	Daisy.creat()	DaisyDisk.writeAllocBit(blockno, ...)	
5	Daisy.get_attr(inodeno, ...)	Daisy.get_attr(inode, ...)	
6	Daisy.ialloc()	Daisy.alloc()	
7	Daisy.ialloc()	DaisyDisk.writeAllocBit(blockno, ...)	
8	Daisy.iget(inodeno)	DaisyDisk.readi(inodeno, inode)	✓
9	Daisy.iget(inodeno)	DaisyLock.acqi(inodeno)	✓
10	Daisy.iget(inodeno)	DaisyLock.reli(inodeno)	✓
11	Daisy.read(inodeno, ...)	Daisy.read(inode, ...)	✓
12	Daisy.write(inodeno, ...)	Daisy.write(inode, ...)	✓
13	DaisyDir.writeLong(inodeno, ...)	Utility.longToBytes(...)	
14	DaisyDisk.readi(inodeno, inode)	DaisyLock.reli(inodeno)	
15	DaisyLock.acqb(blockno)	DaisyLock.relb(blockno)	
16	DaisyLock.acqi(inodeno)	DaisyDisk.readi(inodeno, inode)	✓
17	DaisyLock.acqi(inodeno)	DaisyLock.reli(inodeno)	✓
18	LockManager.acq(lockno)	LockManager.rel(lockno)	✓
19	Mutex.acq()	Mutex.rel()	✓
20	Petal.read(location, ...)	RAF.length()	
21	Petal.write(location, ...)	RAF.writeByte(...)	
22	RAF.seek(location)	RAF.readByte() | RAF.writeByte(...)	✓

```
1  public class Alt {
2      private final static byte[ ][ ] rule = { { 1, 2 }, { 2, 0 }, { 2, 2 } };
3      private byte currState;
4      public Alt() {
5          currState = 0;
6      }
7      public synchronized void update(int event) {
8          Verify.beginAtomic();
9          currState = rule[currState][event];
10         assert (currState != 2);
11         Verify.endAtomic();
12     }
13     public synchronized void checkExitState() {
14         assert (currState == 0);
15     }
16 }
```

FIGURE 8.22: The Java code for monitoring Alternating properties.

Setup. We developed a Java class, Alt, for checking the Alternating template (Figure 8.22). We encode a safety property as an assertion on the FSM's state, which says the current state cannot be an error state (line 10). Our instrumentor instruments Daisy so that it calls the update method whenever the P or S event occurs. Our instrumentor also inserts a call to the checkExitState method (line 13-15) to ensure that the current state is in an accepting state before the program terminates (to catch bugs such as when a lock is not released.

JPF did not support many Java native classes such as RandomAccessFile (RAF). We created an array to emulate RAF. We also created a simpler test harness, DaisyTestSimple, which creates one file and two threads. Each thread either reads from or writes to the created file once. We used JPF's Verify.random in place of Java's random number generator so that JPF would automatically explore all possible results of the random number generator.

In our preliminary experiments, JPF did not finish analyzing several properties within 24 hours. JPF allows users to indicate a sequence of statements as an atomic segment by enclosing the statements between Verify.beginAtomic and Verify.endAtomic. This significantly reduces the number of states JPF has to check. To improve performance, we enclosed the initialization code of DaisyTestSimple and the monitoring code in atomic segments (line 8-11 in Figure 8.22).

Results. When JPF finds a counterexample, it might be a bug. For example, consider the RAF.seek $\rightarrow$ (RAF.read | RAF.write) property (number 22 in Table 8.6), which says whenever RAF.seek is called, RAF.read or RAF.write must be called before the next invocation of RAF.seek. JPF detected a violation of this property, where RAF.seek was called twice without a call to either RAF.read

or RAF.write in between. Diagnosing this problem revealed a race condition in Daisy that was also detected by several other verification tools [30]. After one thread moves the file pointer to location A, another thread starts executing and moves the file pointer to location B. If the first thread is scheduled to execute again, it would write to an incorrect position.

A counterexample can also result from a faulty property inferred from inadequate execution. For example, Perracotta inferred DaisyLock.acqi(inodeno) → DaisyLock.reli(inodeno) (number 17 in Table 8.6). These two methods operate on the lock of the inode whose inode number equals inodeno. Although this property appears to be valid, JPF found a counterexample. Inspecting the code revealed a subtle and interesting aspect of Daisy. DaisyLock.acqi(inodeno) calls LockManager.acq(lockno) that calls Mutex.acq() corresponding to the Mutex object that has the lockno. Similarly, DaisyLock.reli(inodeno) calls LockManager.rel(lockno) that calls Mutex.rel() corresponding to the Mutex object that has the lockno. Therefore, as long as the implementation of Mutex guarantees synchronized access to an inode, an upper level class (e.g., DaisyLock) does not have to ensure synchronization. JPF detected counterexamples to properties 8 to 12 and 16 to 19 in Table 8.6 for a similar reason. Although such counterexamples do not reveal bugs, they provide insight into some important yet subtle properties of Daisy.

Furthermore, the counterexample of Mutex.acq() → Mutex.rel() (number 19 in Table 8.6) revealed a limitation of our inference technique. Figure 8.23 shows the implementation of the two methods. Recall that our instrumentor monitors the entrance of a method. Hence, the Mutex.acq() event corresponds to entry of the acq() method of the Mutex class (line 4). Similarly, Mutex.rel() event corresponds to entry of the rel() method of the Mutex class (line 14). Therefore, Mutex.acq() does not have to alternate with Mutex.rel() because Mutex.acq() does not correspond to the start of the critical section (line 12).

Windows

We use the ESP verifier [31] to verify the inferred properties of the Windows kernel from Section 8.3.3. ESP is a validation tool for typestate properties [118]. Typestates are more expressive than ordinary types: for an object created during program execution, its ordinary type is invariant through the lifetime of the object, but its typestate may be updated by certain operations. ESP allows a user to write a custom specification encoded in a finite state machine to describe typestate transitions. Based on the specification, ESP instruments the target program with the state-changing events. It then employs an inter-procedural data-flow analysis algorithm to compute the typestate behavior at every program point [108]. ESP handles inter-procedural analysis through the use of partial transfer functions via function summaries.

From the 56 inferred properties (see Table 8.5), we randomly selected ten locking properties and checked them using ESP. ESP found one previously unknown deadlock bug in the NTFS file system. The property is a typical

```
1  class Mutex {
2      boolean locked;
3      ……
4      synchronized void acq() {
5        while (locked) {
6            try {
7                this.wait();
8            } catch (Exception e) {
9                System.out.println(e);
10           }
11       }
12       locked = true;
13     }
14     synchronized void rel() {
15       locked = false;
16       this.notify();
17     }
18 }
```

FIGURE 8.23: The Mutex class in Daisy.

locking discipline property: acquiring a specific type of kernel Mutex must be followed by releasing the same Mutex. Figure 8.24 shows a snippet of the buggy code (the function names have been changed to protect Microsoft's proprietary information). Whenever GetData is called from PushNewInfo, the Mutex will be acquired twice (line 3 and 13), causing the system to deadlock. ESP clearly showed this execution path. To fix it, the second argument on line 6 should be changed to TRUE. The Windows development team confirmed that this was a real bug and subsequently fixed the problem.

For all 10 properties, ESP produced false positives. The number of false positives was large for some properties. For example, ESP found 600 counterexamples of one property. We manually examined several counterexamples that all turned out to be false positives due to known limitations of ESP. Manually examining all counterexamples would take too much time and might not be worthwhile as previous experience indicated the false positive rate could be high. Instead, we wrote a simple Perl script to recognize the syntactical pattern of known false positives, ran the script on the counterexamples, and only manually examined those counterexamples that the script did not recognize. This process helped us eliminate those cases that were highly likely false positives. For this property, the script matched all 600 counterexamples mentioned earlier with known false positive patterns.

One common type of false positive we observed was caused by the imprecision of pointer analysis. For example, ESP does not precisely analyze the target of a function pointer and therefore might consider infeasible paths. About half of the 600 false positives were caused by the imprecision of function pointer analysis. Another type of false positive we observed was caused by the imprecision in modeling non-linear arithmetic operators. ESP uses a theorem

```
1    void PushNewInfo ( struct1 s1, struct2 s2 ) {
2        …
3        acquire ( s1.mutex);
4        …
5        if ( s2.flag )
6            GetData ( s1, FALSE );
7        …
8    }
9
10   void GetData ( struct1 s1, boolean locked ) {
11       …
12       if ( !locked ) {
13           acquire ( s1.mutex );
14       …
15   }
```

FIGURE 8.24: The NTFS bug in Windows Vista.

prover to decide whether a branch should be taken. Because its theorem prover cannot precisely model the effect of non-linear arithmetic operators such as shift, ESP might explore infeasible paths.

Diagnosing the counterexamples of a property took at least a day of human effort. Some types of false positives are very difficult to diagnose. For example, one counterexample caused by the imprecise modeling of the shift operator took four people including two ESP developers and one Windows developer eight hours to diagnose. Initially, three of the four people believed it was a bug in Windows until the fourth person found it to be a false positive after careful inspection of the code.

Even though diagnosis of the counterexamples was very time-consuming, it increased our confidence in the correctness of the Windows code. In addition, some of the false positives motivated the ESP developers to enhance ESP to reduce the false positives and to make diagnosis of counterexamples easier.

8.4.2 Program Differencing

Inferring the differences between two versions of a program is an important problem in program evolution [11, 70, 71, 73, 93]. We hypothesize that Perracotta can infer useful properties for differentiating programs. In particular, comparing the inferred properties can increase users' confidence that desirable temporal properties are preserved by modifications. Furthermore, inconsistencies among the inferred properties can reveal interesting facts such as bug fixes or program enhancement.

This section presents experiments applying Perracotta to two families of programs: student implementations of a multi-threaded programming assignment in a graduate software systems course and archived versions of OpenSSL [99]. Because all programs in each case implement the same informal specification, any differences in the inferred temporal properties are likely to be

```
Bus waiting for trip 1
Passenger 0 gets in
Bus drives around Charlottesville
Passenger 0 gets off
Bus waiting for trip 2
Passenger 1 gets in
Bus drives around Charlottesville
Passenger 1 gets off
Bus stops for the day
```

FIGURE 8.25: Sample output of Bus Simulator with $n = 2$, $C = 1$, and $T = 1$.

interesting. For these experiments, we use the acceptance threshold of 1.0 for all the properties, thus only consistently true properties are considered. We use Perracotta to infer the strictest properties for each version of a program. We call the inferred properties the *signature* of each version and compare the signatures of program versions.

Tour Bus Simulator

The first experiment used submissions for an assignment in a graduate software systems course taught at the University of Virginia in Fall 2003. The assignment was a multi-threaded program simulating the operation of city bus with an informal specification, paraphrased below:

Write a program that takes three inputs: n, the number of passengers, C, the maximum number of passengers the bus can hold (C must be $\leq n$), and T, the number of trips the bus takes, and simulates a tour bus transporting passengers around town. The passengers repeatedly wait to take a tour of town in the bus, which can hold a maximum of C passengers. The bus waits until it has a full load of passengers, and then drives around town. After finishing a trip, each passenger gets off the bus and wanders around before returning to the bus for another trip. The bus makes up to T trips in a day and then stops.

The assignment also specified the format of input and output. Figure 8.25 shows the output of a typical execution when $n = 2$, $C = 1$, and $T = 1$.

Because the outputs corresponded to events of interest to us, we did not need to instrument the programs. Instead, we mapped the output logs directly to event sequences. In the mapping, we considered these five events:

(a) wait (*Bus waiting for trip n*)

(b) drives (*Bus drives around Charlottesville*)

(c) stops (*Bus stops for the day*)

(d) gets in (*A passenger gets in*)

(e) gets off (*A passenger gets off*)

TABLE 8.7: Bus Simulator Properties.

P	*S*	Property in Correct Versions	Property in Faulty Version
wait	drives	Alternating	MultiEffect
wait	gets off	MultiEffect	CauseFirst
drives	gets off	MultiEffect	CauseFirst
wait	gets in	MultiEffect	MultiEffect
gets in	drives	MultiCause	MultiCause
gets in	gets off	CauseFirst	CauseFirst
drives	stops	MultiCause	N/A
gets in	stops	MultiCause	N/A
wait	stops	MultiCause	N/A
gets off	stops	MultiCause	N/A

Note that the numbers of the trip and passenger were ignored in our event mapping to reduce the number of distinct events.

A correct solution must satisfy several temporal properties:

(a) The bus always drives with exactly C passengers.

(b) No passenger jumps off or on the bus while it is running.

(c) No passenger starts another trip before getting off the bus.

(d) All passengers get off the bus before passengers for the next trip begin getting on.

We analyzed eight different submissions, all of which were previously evaluated as correct by a grader.

Inference Results. We executed each solution 100 times with randomly generated parameters ($20 \leq C \leq 40$, $C + 1 \leq n \leq 2C$, and $1 \leq T \leq 10$). We used Perracotta to infer the strictest pattern the execution traces satisfy (Section 8.2.4). Perracotta inferred the same set of temporal properties for seven out of the eight submissions. Table 8.7 summarizes the results.

Perracotta inferred the Alternating pattern, wait $\rightarrow$ drives, for seven of the programs. In the other program, the strictest pattern inferred for wait and drives was MultiEffect. Recall that the regular expression of the MultiEffect pattern is $(PSS^*)^*$. The result indicates multiple drives events corresponded to one wait event. This led us to find the bug shown in Figure 8.26. At the end of function go_for_drive, the bus thread releases the lock (line 12). This effectively allows the passenger threads to compete for the lock (line 2) and to possibly "get in" the bus before the bus starts waiting for passengers. In most cases, the bus thread can successfully obtain the lock (line 2) before it has been filled to capacity (i.e., num_riders $<$ capacity on line 3), so it can generate the wait event (line 4). However, the bus can be already full when the bus thread obtains the lock (i.e., num_riders $\geqslant$ capacity on line 3), in which case it does not produce the wait event. In such situations, wait and drives do not alternate. One way to fix this bug would be to use a conditional variable to synchronize the

```
1   void go_for_drive() {
2       pthread_mutex_lock (&mutex[mutex_lock]);
3       if (num_riders < capacity) {
4           printf ("Bus waiting for trip %d\n", num_trips);
5           pthread_cond_wait (&cond[cond_shuttle_full], &mutex[mutex_lock]);
6       }
7       printf ("Bus drives around Charlottesville\n");
8       sleep (3);
9       pthread_cond_broadcast (&cond[cond_ride_over]);
10      num_riders = 0;
11      num_trips--;
12      pthread_mutex_unlock (&mutex[mutex_lock]);
13  }
```

FIGURE 8.26: A synchronization bug in one bus implementation.

bus thread and the passenger threads and make sure the bus thread generates the wait event before it broadcasts that condition.

Another difference concerned the property between drives and gets off. In seven of the implementations, drives and gets off satisfied MultiEffect, but in the other implementation the strictest property satisfied is CauseFirst ($(PP^*SS^*)^*$), which meant it was possible for the bus to drive around Charlottesville more than once without allowing passengers to get off between these trips. This turned out to be another bug of missing synchronization between the bus thread and the passenger threads. As shown in Figure 8.26, the bus thread broadcasts that the ride is over to all passenger threads after driving around the city (line 9). Then it should wait for all the passengers to get off before starting the next trip. If the bus thread runs before any passengers depart, it will still be full and will begin the next trip. The third difference in the property between wait and gets off was caused by the same bug.

OpenSSL

Our second experiment considered six versions of OpenSSL [99], a widely used open source implementation of the Secure Sockets Layer (SSL) specification. The SSL protocol provides secure communication over TCP/UDP using public key cryptography [115]. We focused on the *handshake* protocol that performs authentication and establishes important cryptographic parameters before data transmission starts.

Figure 8.27 illustrates the handshake protocol (derived from the SSL specification) [115]. The three boxes with dashed outlines contain internal events introduced by the OpenSSL implementation but not specified in the SSL specification. The remaining boxes contain sequences of events corresponding to messages defined by the SSL handshake protocol. We gave each server event a more descriptive name and showed the original server event in the parentheses.

The handshake protocol begins when the server receives a ClientHello message from a client. Then the server sends out five messages consecu-

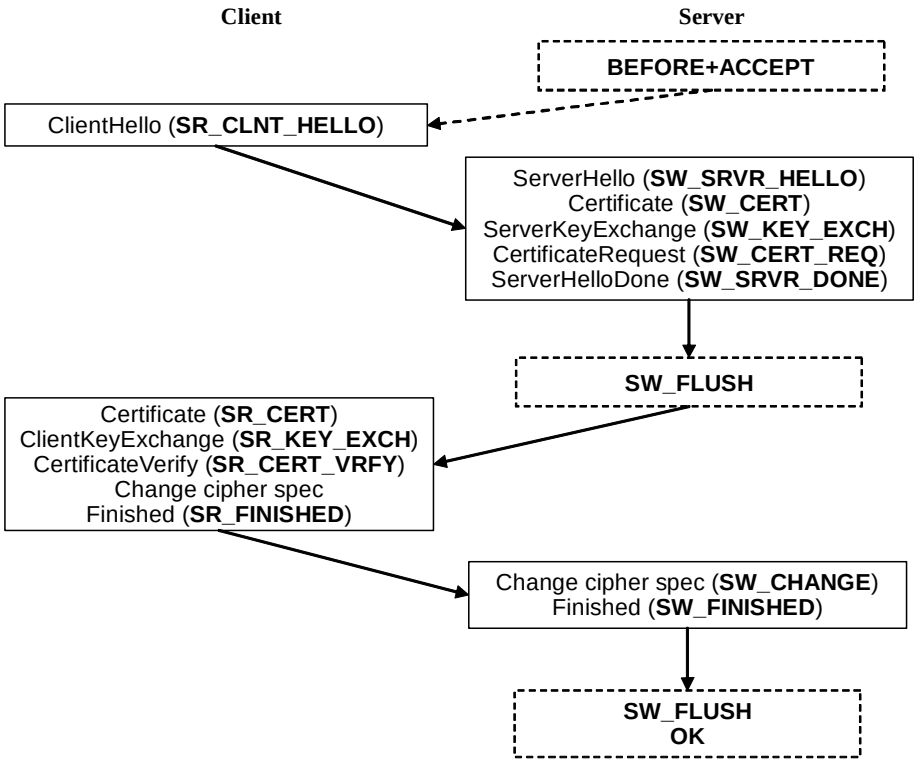


FIGURE 8.27: SSL handshake protocol states.

```
BEFORE+ACCEPT → SR_CLNT_HELLO → SW_SRVR_HELLO → SW_CERT →
SW_KEY_EXCH → SW_CERT_REQ → SW_SRVR_DONE → SR_CERT →
SR_KEY_EXCH → SR_CERT_VRFY → SR_FINISHED → SW_CHANGE →
SW_FINISHED → OK
```

FIGURE 8.28: A server event trace of normal handshake process.

tively (ServerHello, Certificate, ServerKeyExchange, CertificateRequest, and ServerHelloDone). Next, the server enters the SR_CERT state in which it tries to read certificate from the client (whether or not the client sends its certificate depends on the server's certificate request message). Then the server reads four consecutive messages from the client (Certificate, ClientKeyExchange, CertificateVerify, and Finished). If no error occurs, the server sends out its ChangeCipherSpec message and wraps up the handshake by sending its Finished message.

As shown in the dash lined boxes in Figure 8.27, the server implements several additional internal states which are also monitored. First, the server always initializes its state to BEFORE+ACCEPT at the beginning of the handshake. After sending each batch of messages, the server flushes the socket by entering the SW_FLUSH state. In addition, OK is another internal state indicating that the server cleans things up and is ready for data transmission. Figure 8.28 shows a typical event trace.

In the OpenSSL server implementation, the handshake process is encapsulated in the function, ssl3_accept. This function implements the protocol state machine as an infinite loop that checks the current protocol state, sends or receives messages, and advances the state accordingly. We manually instrumented this function to monitor the 15 events shown in Figure 8.27.

Running OpenSSL. We adapted the OpenSSL client to generate OpenSSL server traces. We were particularly interested in analyzing the server's behavior when the client does not follow the protocol correctly, since this is often a source of errors. Therefore, we modified the OpenSSL client so that it transitioned to some randomly selected state different from the original one with 5% probability.

Our test harness was based on a simple OpenSSL-based implementation of the HTTPS protocol: wclient and wserver (version 20020110) originally developed by Eric Rescoria [109]. We added one more command-line option to wclient so that we could seed the random number generator with a specific integer to reproduce of the experiments. We modified wserver so that it only accepted one connection and exited after that. We also added handler functions to print out SIGSEGV and SIGPIPE signals.

We selected six versions of OpenSSL: 0.9.6, 0.9.7, 0.9.7a, 0.9.7b, 0.9.7c, and 0.9.7d. For each version, we executed both wclient and wserver in tandem 1000 times on two machines, which produced 1000 server traces. We used the default keys and certificates supplied in the example program's package and the default choice of cryptographic algorithm.

Results. We first applied Perracotta to all traces. Then we partitioned

	0.9.6	0.9.7	0.9.7a	0.9.7b	0.9.7c	0.9.7d
SW_CERT→SW_KEY_EXCH		✓	✓	✓	✓	✓
SR_KEY_EXCH→SR_CERT_VRFY	✓	✓	✓	✓		
SW_SRVR_DONE→SR_CERT		✓				

TABLE 8.8: Alternating properties satisfied by six versions of OpenSSL.

the traces into four groups: 1) correct client (i.e., the client neither changes to any unintended state nor generates segmentation faults); 2) faulty client (i.e., the client changes its state to some unintended one at least once) but no errors generated in traces; 3) segmentation fault; 4) faulty client generating errors other than segmentation faults. We applied Perracotta to each group of traces.

Table 8.8 highlights three key differences among the inferred Alternating properties. In the first row, Perracotta inferred SW_CERT → SW_KEY_EXCH, for all versions except 0.9.6. Investigating the traces revealed that the server sometimes crashed after entering SW_CERT state with a SIGPIPE signal. We found this was caused by a documented critical bug in earlier versions of OpenSSL [99].

In the second row, Perracotta inferred SR_KEY_EXCH → SR_CERT_VRFY, for all versions up to 0.9.7b but not for 0.9.7c and 0.9.7d. This was caused by a change to version 0.9.7c in order to make the implementation conform to the SSL 3.0 specification (documented in the change log of version 0.9.7c [99]). Starting from version 0.9.7c, a server did not process any certificate message it received from a client if it did not previously request authentication of that client. The condition of whether the server requested client authentication was recorded in a variable, which could be set using a command line option. The server checked this variable to determine its next state after entering the receiving certificate state (SR_CERT). If the server did not request client authentication, it directly advanced to receiving key exchange state (SR_KEY_EXCH). Otherwise, it first read and examined the client's certificate before changing to that state. Our faulty client may send its certificate even if the server did not request one. A server earlier than 0.9.7c noticed this as an error and stopped the handshake immediately: the SR_KEY_EXCH state was not entered at all. In contrast, a server of version 0.9.7c or 0.9.7d ignored the client's certificate, continued to change its state to SR_KEY_EXCH, and then stopped the handshake because of the wrong type of message the client sends (it expected the ClientKeyExchange message, but got the Certificate message).

In the third row, Perracotta inferred SW_SRVR_DONE → SR_CERT, only for version 0.9.7. Using the log messages we identified the cause to be a race condition present in all versions. When the client changed to a false state and got some unexpected message from the server, it tried to send an alert message to the server to stop the handshake process. After that, the client disconnected

BEFORE+ACCEPT → SR_CLNT_HELLO → SW_SRVR_HELLO → SW_CERT → SW_KEY_EXCH → SW_CERT_REQ → SW_SRVR_DONE → SR_CERT → SR_KEY_EXCH → SR_CERT_VRFY → SR_FINISHED → SW_CHANGE → SW_FINISHED → OK

FIGURE 8.29: Inferred alternating chains for correct OpenSSL clients.

SR_CLNT_HELLO → SW_SRVR_HELLO → SW_CERT → SW_KEY_EXCH → SW_CERT_REQ → SW_SRVR_DONE → SR_CERT

BEFORE+ACCEPT → SR_KEY_EXCH → SR_CERT_VRFY → SR_FINISHED → SW_CHANGE → SW_FINISHED → OK

FIGURE 8.30: Inferred alternating chains for non-error faulty OpenSSL clients.

the socket with the server. If the client disconnected the socket during the server sending messages, the server would get a sending error message. The server had not and would not get the alert message from the client now because the socket had been disconnected. The server would only be able to get the alert message if it had already finished sending messages to client and entered a receiving state. In the experiment, this receiving state was SR_CERT. If the server was able to enter this state before the client sent the alert message, this event and an alert message would both be printed out at the server. Therefore, there was no guarantee that an alert message would be received after sending. Perracotta discovered this important design decision that was not documented in the specification.

Correct clients. Perracotta inferred an Alternating Chain as shown in Figure 8.29 from the server traces in which the client behaves correctly (i.e., the 5% probability of switching to a random state is never selected). All versions agree on the same Alternating Chain. This result is desirable because the pattern in Figure 8.29 conforms exactly to the SSL specification as shown in Figure 8.27 (page 292). This demonstrates that the server implementation of the handshake protocol conforms to the specification as OpenSSL evolves.

Faulty clients without errors generated. Next, we considered the set of traces corresponding to faulty clients that, surprisingly, did not generate any error event on either the server or client. For all six versions, Perracotta inferred the two Alternating Chains shown in Figure 8.30. These two chains closely follow the normal handshake. However, there are a few key distinctions between the patterns in Figure 8.29 and Figure 8.30.

Two of the Alternating properties that are present in Figure 8.29 do not appear in Figure 8.30. Instead, those event pairs satisfy weaker patterns. SR_CERT and SR_KEY_EXCH satisfy the MultiCause pattern, and BEFORE+ACCEPT and SR_CLNT_HELLO satisfy the MultiEffect pattern. Figure 8.31(a) shows a trace that violates the expected Alternating properties. The key

BEFORE+ACCEPT, OK+ACCEPT,
SR_CLNT_HELLO, SW_SRVR_HELLO, SW_CERT, SW_KEY_EXCH,
SW_CERT_REQ, SW_SRVR_DONE, SW_FLUSH,SR_CERT,
SR_CLNT_HELLO, SW_SRVR_HELLO, SW_CERT, SW_KEY_EXCH,
SW_CERT_REQ, SW_SRVR_DONE, SW_FLUSH,SR_CERT,
SR_KEY_EXCH, SR_CERT_VRFY, SR_FINISHED, SW_CHANGE, SW_FINISHED,
SW_FLUSH, OK

(a) Server trace

BEFORE+CONNECT, OK+CONNECT, CW_CLNT_HELLO,
CR_SRVR_HELLO,CR_CERT, CR_KEY_EXCH, CR_CERT_REQ,CR_SRVR_DONE,
RENEGOTIATE,
BEFORE, CONNECT, BEFORE+CONNECT, OK+CONNECT, CW_CLNT_HELLO,
CR_SRVR_HELLO, CR_CERT, CR_KEY_EXCH, CR_CERT_REQ, CR_SRVR_DONE,
CW_KEY_EXCH, CW_CHANGE, CW_FINISHED, CW_FLUSH, CR_FINISHED, OK

(b) Client trace

FIGURE 8.31: Traces generated with a faulty OpenSSL client.

distinction between this trace and the traces produced using correctly behaving clients is that the eight-event sequence appears twice (shown in boldface in Figure 8.31(a)). Figure 8.31(b) shows the corresponding client events. The faulty client falsely changes its state to renegotiate (an internal state in the implementation of the client, not shown in Figure 8.27 since it is not part of the normal handshake process) instead of CW_CERT after reading the five messages from server (ServerHello, Certificate, ServerKeyExchange, CertificateRequest, and ServerHelloDone). Then, the client starts the handshake again by sending the ClientHello message, which causes the server to repeat the hello stage of the handshake again. If a client always changes its state to renegotiate after receiving the ServerHelloDone message, the server and the client will enter an infinite loop.

Suspecting this could be exploited in a denial of service (DOS) attack, we contacted the OpenSSL developers. They argued that it did not indicate a serious DOS vulnerability because the server looped infinitely only when an ill-behaved client kept sending renegotiation requests. This was similar to too many clients attempting to connect to a server, which was a scenario that could not really be prevented at the server. Although this was not a real vulnerability, it did reveal an interesting aspect of OpenSSL, which was not documented in the SSL specification and was not obvious from code inspection.

Segmentation faults. There were three traces that included segmentation faults in all versions prior to 0.9.7d. These traces were from the faulty client that sent a change_cipher_spec instead of the normal client hello message at the beginning of the handshake process. We examined the change log for version 0.9.7d and found that this was due to a critical update [100], where an assignment to a null-pointer in the do_change_cipher_spec function caused the server to crash. Although this finding did not result from comparing the

inferred temporal properties, it shows that using randomly behaving clients to test a server is powerful enough to uncover important problems.

Faulty clients with other errors. For all versions, Perracotta inferred the same temporal properties for traces within this category. This demonstrated that the server handled misbehaving clients consistently with respect to the properties Perracotta inferred.

8.5 Related Work

This section surveys related work. Section 8.5.1 describes the grammar inference problem and its complexity, which provides the theoretical context of the specification inference problem. Section 8.5.2 classifies other inference work based on its underlying techniques and describes the representative work in each category. Finally, Section 8.5.3 presents previous work on using the inferred specifications.

8.5.1 Grammar Inference

A grammar G describes a language L if and only if $L(G)$ (the language generated by G) equals to L. The grammar inference problem can be informally stated as: "given some sample sentences in a language (positive samples), and perhaps some sentences specifically not in the language (negative samples), infer a grammar that describes the language." [22] Inferring temporal specifications from a program's execution traces is a concrete example of the grammar inference problem, where the specifications inferred comprise the grammar and the execution traces are the sample sentences. Gold developed the first theoretical framework for analyzing the grammar inference problem [54]. Gold proved that it is NP-hard to infer a deterministic finite-state automaton (DFA) with a minimum number of states from positive and negative sample sentences [55]. In addition, Gold showed that it is impossible to infer such a DFA given only positive samples because an algorithm has no way to determine whether it is overgeneralizing. Cook et al. surveys the grammar inference problem, its theoretical complexity, and several practical inference techniques [22].

Gold's results indicate that it is very difficult to infer the exact grammar (e.g., a DFA with minimal number of states) even for relatively simple languages such as regular languages. This explains why earlier specification inference approaches to extract a complete finite-state machine do not scale well.

8.5.2 Property Inference

We classify other property inference work into two main categories: *template-based inference techniques* that have a set of pre-defined templates and try to match either execution traces or static program paths against these templates [42, 43, 48, 59, 60, 104, 121] and *arbitrary model inference techniques* that do not have a set of pre-defined templates and can discover arbitrary models that execution traces or static program paths satisfy [4, 23, 50, 89, 107, 123]. The major difference between a template-based inference technique and an arbitrary model inference technique is that a template-based technique tries

to detect whether and how a program fits in a specific patterns, whereas an arbitrary model inference technique tries to create the best model that fit the target program or its execution traces.

We can further classify techniques in each of the two main categories into two subcategories: machine learning based techniques that use statistical machine learning to infer patterns [4, 23, 42, 43, 60, 89, 107, 121, 123] and dataflow analysis based techniques that infer properties by either symbolically executing a program [50,59,104] or use a verification tools as an oracle [48]. Following we compare our work with the work in each category.

Template-based inference

Template-based inference techniques have a set of pre-defined templates that can capture pre-condition, post-condition, and invariants [44,48,60], temporal constraints [42,121], or very specific program features such as locks [104] or buffers [59].

Daikon is a tool that automatically infers likely program invariants from a program's execution traces [43,44,103]. The technique uses a set of pre-defined invariant patterns that are matched against the values observed in the traces; invariants that are violated are dropped. Invariants that survive are ranked in order of confidence. Only invariants that are above a specified confidence threshold are reported. The original Daikon inference algorithm is limited by its scalability because the algorithm is cubic in the number of variables monitored [44]. Several new optimizations recently developed by Perkins and Ernst greatly improve Daikon's performance, but performance remains an impediment to applying Daikon to large-scale programs [103]. Our work is distinct from Daikon in that our approach infers specifications of the temporal behaviors of a system, which Daikon does not infer. Our technique also scales much better to large-scale real systems such as Windows. In addition, Daikon requires 100% satisfaction of templates and therefore is less robust to imperfect traces than our approximate inference algorithm.

Diduce is a run-time anomaly detection tool that is based on invariant inference [60]. Diduce infers data invariants as a program executes and generates alarms whenever the execution violates the current invariants. The invariants Diduce infers are bit field patterns (e.g., the last bit of a byte at certain address is always 0) and therefore are useful for detecting memory usage errors at runtime. Our work is different from Diduce in that our work focuses on API level invariants. In addition, the properties Perracotta infers can be used in several other purposes than run-time monitoring.

Engler et al. proposed a method for extracting properties by statically examining source code based on a set of pre-defined templates [42]. Like our approach, their technique scales well by targeting simple property templates. Our chaining method can build more complex properties that their technique does not infer. To deal with the imprecision of static analysis, they use statistics to prioritize their results. Similarly, our approximate inference algorithm

is also statistical. Furthermore, they use a set of specific names to reduce the number of candidate events [42]. In order to infer a property, Engler's technique requires an event (e.g., a function `foo`) occurs frequent enough in the code base (e.g., `foo` is called at many different places). In contrast, our dynamic technique can still infer a property as long as an event occurs frequent enough in the trace. The limitation of Engler's work is that it often produces too many false positives. To lower the rate of false positives, Weimer et al. improved Engler's work by developing an approach that examines a program's exception handling routines [121]. The intuition is that programs often make mistakes on exceptional paths, even when they are mostly correct on the normal paths. On one hand, Weimer's approach mainly focuses on local properties, while our technique can identify relationships among events that are far removed from each other in program text. On the other hand, Weimer's work and our work complement each other because it is usually very hard to generate test inputs for exception handling routines.

Houdini is an annotation inference tool for ESC [48, 49]. The annotations are in the standard Hoare-style logic that includes pre-condition, post-condition, and invariants [65]. Houdini first generates a set of candidate annotations and then feeds these candidates to ESC. ESC either proves or refutes a property. The main problem of Houdini is that it does not scale well because the underlying ESC is slow.

Other static inference techniques focus on certain program features and are very effective. LockSmith is a tool for automatically inferring locking discipline annotations that are later used to check for deadlocks and race conditions [104]. SALInfer is a tool for inferring annotations for function arguments that are related to buffer usages [59]. SALInfer has been successfully used to help annotate the whole Windows code-base. Both LockSmith and SALInfer use dataflow analysis. Compared to LockSmith, Perracotta infers similar type of property using a regular expression inference from execution traces. Compared to SALInfer, our work focuses on a different class of properties that SALInfer does not infer. Furthermore, our underlying technique is based on regular expression matching instead of dataflow analysis.

Arbitrary model inference

Arbitrary model inference techniques try to discover a model that fits the target program or its execution traces. Most of the related work in this category analyzes program execution traces [4, 23, 107, 123]. Other artifacts analyzed include revision history [89] and source code [50].

Ammons et al. used an off-the-shelf probabilistic finite state automaton learner to mine temporal and data-dependence specifications for APIs or abstract data structures from dynamic traces [4, 5]. In addition to handle traces containing bugs (i.e., imperfect traces as defined in our work), their approach required non-trivial human guidance. In contrast, our techniques can automatically tolerate imperfect traces without guidance. Their machine learning

algorithm has a high computational cost, whereas our algorithm scales better to larger traces than theirs.

Whaley et al. proposed a static and a dynamic approach for inferring what protocols clients of a Java class must follow [123]. The protocols their approach found are mainly typestate properties that involve one component and are small. In contrast, our approach is able to discover useful properties involving more than one component. In addition, our chaining method is able to construct large finite state machines efficiently.

Cook et al. developed a statistical dynamic analysis for extracting thread synchronization models from a program's execution traces [23]. Our work differs from theirs in that our approach focuses on detecting API rules and assumes the trace already has the thread information.

Reiss et al. developed a technique to compact large volumes of execution traces [107]. Their tool uses the sequencing properties on individual objects, while Perracotta detects rules across multiple objects.

DynaMine extracts error patterns from a system's CVS revision histories and dynamically validates inferred patterns [89]. This approach is complementary to our work in that examining a CVS history is a way to select events to monitor at run-time. Their mining algorithm has to filter out a fixed set of frequent events to scale to large scenarios, which is not as general as our heuristics. The patterns their approach infers tend to focus only on methods within a class, whereas our approach can infer properties involving more than one class.

8.5.3 Use of Inferred Specifications

This section presents related work on using the inferred specifications to improve a variety of software development activities including defect detection [20,28,29,42,48,50,59,60,89,98,104,121,123], test case generation [29,61,125], program evolution [44], program understanding [23,90], theorem proving [124], bug localization [85], and data structure repairing at runtime [33].

Defect Detection

Defect detection is the application to which inferred specifications are most widely applied. We can further divide the related work into several groups: using a separate verification tool [48,59,89,98,123], statistical defect detection [20,42,60,121], context-free language reachability analysis [50,104], combining static analysis and test generation [28,29].

The work in the first group simply feeds the inferred properties to a separate program verification tool [48,59,89,98,123]. This is similar as our work on checking the inferred properties using Java PathFinder and ESP. The benefits of such an approach are that inferred specifications can be easily checked by different verification tools.

The work using statistical defect detection integrates specification infer-

ence tightly with defect detection [20, 42, 60, 121]. Engler et al. pioneered work in statistical defect detection [42]. Their technique examined a program's source code to observe common behavioral patterns (called *belief*) and detected violations of the common behavioral patterns based on statistics. Weimer et al. tried to reduce the false positive rate by focusing the analysis on exception handling routines [121]. The Diduce tool used a similar idea as Engler's work except that Diduce detected abnormal program behavioral at runtime [60]. Chilimbi et al. was inspired by Diduce and aimed to detect abnormal memory usage patterns [20]. Statistical defect detection works well in practice because real world programs, even though not perfectly correct, behave correctly most of the time and therefore only expose abnormal behaviors occasionally. Its drawback is that the defects that can be detected are hard-coded and therefore extending existing tool to check new defects requires significant efforts. Our approximate inference algorithm bears a similar spirit as the statistical defect detection work. Instead of detecting bugs, our approximate inference algorithm tries to use a statistical approach to tolerate imperfect program behaviors.

Foster's PhD dissertation on type qualifiers introduced the idea of using constraint-based context-free language reachability analysis to infer type annotations and detect bugs [50]. More recent work applied this idea to infer correlation between data and locks [104]. The inferred correlations were checked for race conditions. This work also tightly combined the inference of specifications (type qualifiers or data-lock correlations) with the detection of their violation.

Recent work by Csallner and Smaragdakis opened an interesting direction of combining dynamic inference, static analysis, and automated test generation for bug detection [28, 29]. Their work used dynamic inference to infer intended program behaviors so that users did not have to provide the specifications, static verification tool to detect violations of inferred invariants, and test generation to check the validity of the errors reported by the static analysis. One advantage of this approach compared to a purely static analysis was that the test generation eliminated the false positives in the results. Compared to this work, our work of checking the inferred specifications still produced too many false positives to be practical.

Other Uses

Inferred specifications were used to augment and minimize test suite [61]. The idea was to use inferred properties as a test coverage criterion. Test cases were selected until the set of inferred properties stabilized. Harder's work, however, did not automatically generate new test cases. Xie and Notkin used inferred properties to automatically generate unit-test cases [125]. In our future work, we plan to investigate how to use the inferred temporal specification to select and generate test cases.

Ernst et al. studied using the invariants inferred by Daikon to aid program-

mers in modifying programs [44]. In their experiments, programmers were supplied with the inferred invariants, as they added a new feature to an existing program. The programmers found the inferred invariants useful. Our work on using inferred invariants in differencing programs was inspired by Ernst's work. Compared to Ernst's work, our work focused more on automatically detecting the semantic difference of multiple versions of a program, whereas Ernst's work aimed to evaluate whether the inferred specification could be useful for programmers.

Naturally, inferred specifications can be used to help programmers gain more insights into the target program. Cook et al. developed a technique for inferring thread interaction models from execution traces [23]. Their technique used a statistical approach. Mandelin et al. developed a technique for discovering the sequence of APIs that were needed to accomplish some tasks [90]. They called their tool *specification prospector* and the inferred API sequence *jungloid*. Our work on inferring Alternating Chains bore a similar motivation as Mandelin's work.

Another representative use of inferred specifications is in guiding theorem prover to verify distributed programs [124].

Liblit et al. showed the inferred specifications can be used to debug a program [85]. Their work monitored a set of program predicates and compared the observed properties of a successful execution against the ones of a failed execution. Their technique used a statistical approach to eliminate irrelevant predicates and rank inferred predicates. One big advantage of this work compared to previous work on bug localization is that it can effectively handle programs that have more than one bug. We also plan to investigate whether the temporal specifications inferred by Perracotta can be useful in debugging.

Much work has been on using inferred specifications to detect bugs, Demsky et al. showed that the inferred specification can also be used to let a program continue its execution in face of data structure corruption [33]. Their technique used Daikon to infer invariants for data structures and automatically detected and restored corrupted data structures to behave normally with regard to the invariants.

8.6 Conclusion

Software specifications are the foundation of many software development activities including maintenance, testing, and verification. We have presented Perracotta, a tool that automatically infers temporal specifications of programs by analyzing execution traces. Perfect, fully-automatic specification inference for industrial programs remains an elusive goal, well beyond the state-of-the-art. We have shown, however, that by targeting simple properties that can be efficiently discovered and by using approximation inference techniques along with heuristics for pruning the set of inferred properties, it is possible to obtain useful results even for programs as large and complex as JBoss and Windows.

Limitations. Our approach, being a dynamic analysis, shares the limitations of any dynamic analysis. In particular, most real systems have an infinite number of execution paths. It is impossible to execute such a system on all of its paths. Dynamic analysis only examines a subset of all paths of a target program and might produce results that are false for some paths. Therefore manual or machine validation is required before they can be used as specifications. In addition, dynamic analysis needs to instrument a target program to observe its behaviors. This instrumentation can affect the normal behavior of a target program. The extra computation introduced by the instrumentation might cause a thread in a real-time system to miss its deadline. The extra memory used by the instrumentation might affect cache locality. Hence, dynamic analysis is impractical to analyze properties that can be affected by the instrumentation.

Our approach uses a set of pre-defined property templates. This limits the properties that can be inferred to those that can be expressed using these templates. Even though we can introduce a new template to express new properties, a new template will also introduce many uninteresting properties that require new heuristics to filter. Developing a good heuristic for distinguishing interesting properties from uninteresting ones requires much effort and may not always be possible. Without a good heuristic, inferred properties can be useless if the density of interesting properties is very low. Furthermore, if a new template is complex (i.e., templates with many parameters), it might be very inefficient to infer properties that satisfy it.

To reduce the effort required to analyze the inference results, our approach relies heavily on the effectiveness of heuristics for selecting interesting properties. The effort required to analyze the remaining properties after applying the heuristics sometimes can still be quite high even for users familiar with the target system. In addition, heuristics, no matter how good they are, can mis-classify interesting properties as uninteresting ones. Therefore, interesting properties may be missing in the final results.

One of our assumptions is that a target program is well tested and ex-

hibits desirable specifications most of the time. To tolerate imperfection in the traces, our technique uses a very simple statistical approach. Even though our experimental results show that this assumption is valid for typical real systems, our approach can hardly be applicable to systems that do not satisfy this assumption.

Summary. We have presented a dynamic analysis approach for inferring interesting temporal properties. Through experiments on several real systems, we have shown that our approach is scalable, effective, and useful in aiding a variety of software development activities. There are many exciting opportunities for further work in both automatic inference of program properties and ways to use inferred program properties. Our current templates are very limited in the kinds of properties they can express. Developing richer templates with more parameters and conditional relationships can capture more important program properties, but poses challenges for scalability. Currently we only infer properties from a fixed set of test cases. We still don't understand how the inferred properties change as we vary the test cases, when the inferred properties stabilize as we add new test cases, and what size of test suite is required to produce a stable set of properties. Such knowledge can guide users in selecting test cases for inferring properties.

The experimental results of applying Perracotta to a diverse range of real systems provide strong evidence that our approach is useful in aiding several different software development tasks: program understanding, program differencing, and program verification. In particular, the experimental results demonstrate that our approach is able to automatically identify important temporal properties, identify interesting behavioral differences among multiple versions of programs, and help find bugs.

Bibliography

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press. October 1996.

[2] T. Andrews, S. Qadeer, J. Rehof, S.K. Rajamani, and Y. Xie. Zing: Exploiting Program Structure for Model Checking Concurrent Software. *International Conference on Concurrency Theory*, August/September 2004.

[3] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. *Symposium on Principles of Programming Languages*, January 2005.

[4] G. Ammons, R. Bodik, and J. R. Larus. Mining Specifications. *Symposium on Principles of Programming Languages*, January 2002.

[5] G. Ammons, D. Mandelin, R. Bodik, and J. R. Larus. Debugging Temporal Specifications with Concept Analysis. *Conference on Programming Language Design and Implementation*, June 2003.

[6] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. *International SPIN Workshop on Model Checking of Software*, May 2001.

[7] The Byte Code Engineering Library. http://jakarta.apache.org/bcel/

[8] K. Bennett, T. Bull, E. Younger, and Z. Luo. Bylands: Reverse Engineering Safety-Critical Systems. *International Conference on Software Maintenance*, October 1995.

[9] B. Beizer. *Software Testing Techniques, Second Edition*. Van Nostrand Rheinold, New York, 1990.

[10] S. Bhansali, W. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. Framework for Instruction-Level Tracing and Analysis of Program Executions. *International Conference on Virtual Execution Environments*, June 2006.

[11] D. Binkley, R. Capellini, L. Raszewski, and C. Smith. An Implementation of and Experiment with Semantic Differencing. *International Conference on Software Maintenance*, November 2001.

[12] J. Bowen, P. Breuer, and K. Lano. Formal Specifications in Software Maintenance: from Code to Z++ and Back Again. *Information and Software Technology*, November/December 1993.

[13] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. *International Symposium on Software Testing and Analysis*, July 2002.

[14] P. T. Breuer and K. Lano. Creating Specifications from Code: Reverse-Engineering Techniques. *Journal of Software Maintenance: Research and Practice*, Volume 3, 1991.

[15] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software - Practice and Experience*, Volume 20, 2000.

[16] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. *International Conference on Software Engineering*, May 2003.

[17] B. Chelf. Measuring Software Quality: A Study of Open Source Software. Coverity Inc. May 2006.

[18] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. *Conference on Computer and Communications Security*, November 2002.

[19] W. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang. Profile-Guided Proactive Garbage Collection for Locality Optimization. *Conference on Programming Language Design and Implementation*, June 2006.

[20] T. M. Chilimbi and V. Ganapathy. HeapMD: Identifying Heap-Based Bugs Using Anomaly Detection. In *Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[21] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[22] J. E. Cook and A. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, Volume 7, 1998.

[23] J. E. Cook, Z. Du, C. Liu, and A. L. Wolf. Discovering Models of Behavior for Concurrent Workflows. *Computers in Industry*, April 2004.

[24] D. Coppit, J. Yang, S. Khurshid, W. Le, and K. Sullivan. Software Assurance by Bounded Exhaustive Testing. *IEEE Transactions on Software Engineering*, Volume 31, April 2005.

[25] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. *International Conference on Software Engineering*, June 2000.

[26] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. A Language Framework for Expressing Checkable Properties of Dynamic Software. *International SPIN Workshop on Model Checking of Software*, August 2000.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.

[28] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining Static Checking and Testing. In *International Conference on Software Engineering*, May 2005.

[29] C. Csallner and Y. Smaragdakis. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. In *International Symposium on Software Testing and Analysis*, July 2006.

[30] Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software, July 2004.

[31] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. *Conference on Programming Language Design and Implementation*, June 2002.

[32] A. M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.

[33] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and Enforcement of Data Structure Consistency Specifications. In *International Symposium on Software Testing and Analysis*, July 2006.

[34] D. L. Detlefs. An Overview of the Extended Static Checking System. *Workshop on Formal Methods in Software Practice*, January 1996.

[35] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. *International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, April 1993.

[36] E. W. Dijkstra. Notes on Structured Programming. *Structured Programming*, Academic Press, 1972.

[37] N. Dor, S. Adams, M. Das, and Z. Yang. Software Validation via Scalable Path-Sensitive Value Flow Analysis. *International Symposium on Software Testing and Analysis*, July 2004.

[38] S. Ducasse and M. Lanza. The Class Blueprint: Visually Supporting the Understanding of Classes. *IEEE Transactions on Software Engineering*, Volume 31, January 2005.

[39] J. W. Duran and S. C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, July 1984.

[40] M. Duvall. Software Bugs Threaten Toyota Hybrids. *The Baseline Magazine*. August 4, 2005. http://www.baselinemag.com/article2/0,1540,1843934,00.asp

[41] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in Property Specifications for Finite-State Verification. *International Conference on Software Engineering*, May 1999.

[42] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: a General Approach to Inferring Errors in Systems Code. *Symposium on Operating Systems Principles*, October 2001.

[43] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D. dissertation, University of Washington Department of Computer Science and Engineering, August 2000.

[44] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, February 2001.

[45] D. Evans. Static Detection of Dynamic Memory Errors. *Conference on Programming Language Design and Implementation*, May 1996.

[46] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate Verification: Abstraction Techniques and Complexity Results. *International Symposium on Static Analysis*, June 2003.

[47] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective Typestate Verification in the Presence of Aliasing. *International Symposium on Software Testing and Analysis*, July 2006.

[48] C. Flanagan, R. Joshi, K. Rustan, and M. Leino. Annotation Inference for Modular Checkers. *Information Processing Letters*, February 2001.

[49] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. *Conference on Programming Language Design and Implementation*, June 2002.

[50] J. S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. Ph.D. thesis. University of California, Berkeley, December 2002.

[51] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*. Addison-Wesley Professional, September 2003.

[52] P. Godefroid. Model Checking for Programming Languages Using VeriSoft. *Symposium on Principles of Programming Languages*, January 1997.

[53] M. W. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. *International Conference on Software Maintenance*, October 2000.

[54] E. Gold. Language Identification in the Limit. *Information and Control*, Volume 10, 1967.

[55] E. Gold. Complexity of Automatic Identification from Given Data. *Information and Control*, Volume 37, 1978.

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

[57] D. Grove and C. Chambers. A Framework for Call Graph Construction Algorithms. *ACM Transactions on Programming Languages and Systems*. Volume 23, November 2001.

[58] N. Gupta. Generating Test Data for Dynamically Discovering Likely Program Invariants. *Workshop on Dynamic Analysis*, May 2003.

[59] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular Checking for Buffer Overflows in the Large. *International Conference on Software Engineering*, May 2006.

[60] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. *International Conference on Software Engineering*, May 2002.

[61] M. Harder, J. Mellen, and M. D. Ernst. Improving Test Suites via Operational Abstraction. *International Conference on Software Engineering*, May 2003.

[62] K. Havelund and G. Rosu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, Volume 24, 2004.

[63] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-Modular Abstraction Refinement. *International Conference on Computer-Aided Verification*, July 2003.

[64] J. Henkel. Discovering and Debugging Algebraic Specifications for Java Classes. PhD Dissertation, University of Colorado at Boulder, May 2004.

[65] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, October 1969.

[66] C. A. R. Hoare. The Verifying Compiler: a Grand Challenge for Computing Research. *Journal of the ACM*, Volume 50, January 2003.

[67] C. M. Holloway and R. W. Butler. Impediments to Industrial Use of Formal Methods. *IEEE Computer*, April 1996.

[68] G. J. Holzmann. The Model Checker Spin. In *IEEE Transactions on Software Engineering*, Volume 23, May 1997.

[69] G. J. Holzmann. The Logic of Bugs. *Symposium on Foundations of Software Engineering*, November 2002.

[70] S. Horwitz. Identifying the Semantic and Textual Differences between Two Versions of a Program. *Conference on Programming Language Design and Implementation*, June 1990.

[71] S. Horwitz and T. Reps. The Use of Program Dependence Graphs in Software Engineering. *International Conference on Software Engineering*, May 1994.

[72] J2EE. *http://java.sun.com/j2ee*

[73] D. Jackson and D. Ladd. Semantic Diff: a Tool for Summarizing the Effects of Modifications. *International Conference on Software Maintenance*, October 1994.

[74] JBoss. *http://www.jboss.org*

[75] JRat. *http://jrat.sourceforge.net*

[76] Java Transaction API specification. *http://java.sun.com/products/jta*

[77] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. 1972.

[78] S. Khurshid and D. Marinov. TestEra: A Novel Framework for Automated Testing of Java Programs. *Automated Software Engineering Journal*, December 2002.

[79] J. C. Knight, C. L. DeJong, M. S. Gibble, and L. G. Nakano. Why are Formal Methods Not Used More Widely? *NASA Langley Formal Methods Workshop*, September 1997.

[80] F. Kröger. *Temporal Logic of Programs*. Springer-Verlag New York, 1987.

[81] A. Lamsweerde. Formal Specification: a Roadmap. In *International Conference on Software Engineering*, May 2000.

[82] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting Software. *IEEE Software*, May/June 2004.

[83] S. Letovsky and E. Soloway. Delocalized Plans and Program Comprehension. *IEEE Software*, May 1986.

[84] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: a Tool for Finding Copy-Paste and Related Bugs in Operating System Code. *Symposium on Operating System Design and Implementation*, December 2004.

[85] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. *Conference on Programming Language Design and Implementation*, June 2005.

[86] L. Lin and M. D. Ernst. Improving Adaptability via Program Steering. *International Symposium on Software Testing and Analysis*, July 2004.

[87] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley Professional, April 1999.

[88] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical Model-Based Bug Localization. *Symposium on the Foundations of Software Engineering*, September 2005.

[89] B. Livshits and T. Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. *Symposium on the Foundations of Software Engineering*, September 2005.

[90] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Mining Jungloids: Helping to Navigate the API Jungle. *Conference on Programming Language Design and Implementation*, June 2005.

[91] D. Marinov, and S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. *International Conference on Automated Software Engineering*, November 2001.

[92] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI Test Case Generation Using Automated Planning. *IEEE Transactions on Software Engineering*. Volume 27, February 2001.

[93] W. Miller and E. W. Myers. A File Comparison Program. *Software - Practice and Experience*, Volume 15, 1985.

[94] N. Mitchell and G. Sevitsky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. *European Conference on Object-Oriented Programming*, July 2003.

[95] M. Musuvathi and D. Engler. Model-Checking Large Network Protocol Implementations. *Conference on Network System Design and Implementation*, March 2004.

[96] B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall, 1997.

[97] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. *International Workshop on Mining Software Repositories*, May 2005.

[98] J. W. Nimmer and M. D. Ernst. Invariant Inference for Static Checking: an Empirical Evaluation. *Symposium on the Foundations of Software Engineering*, November 2002.

[99] OpenSSL. *http://www.openssl.org*

[100] OpenSSL security advisory, 17 March 2004. http://www.openssl.org/news/secadv_20040317.txt

[101] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, July 1982.

[102] D. L. Parnas. On the Criteria to be Used in Decomposing System into Modules. *Communications of the ACM*, Volume 15, December 1972.

[103] J. Perkins and M. Ernst. Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. *International Symposium on Foundations of Software Engineering*, November 2004.

[104] P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive Correlation Analysis for Detecting Races. In ACM Conference on Programming Language Design and Implementation, June 2006.

[105] A. Pnueli. The Temporal Logic of Programs. *Symposium on Foundations of Computer Science*, October/November 1977.

[106] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss. Automated Fault Localization Using Potential Invariants. *International Symposium on Automated and Analysis-Driven Debugging*. September 2003.

[107] S. P. Reiss and M. Renieris. Encoding Program Executions. *International Conference on Software Engineering*, May 2001.

[108] T. Reps, S. Horwitz, and M. Sagiv. Precise Inter-Procedural Dataflow Analysis via Graph Reachability. *Symposium on Principles of Programming Languages*, January 1995.

[109] E. Rescorla. An Introduction to OpenSSL Programming, Part One. http://www.rtfm.com/ openssl-examples/, October 2001.

[110] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. Transactions of the American Mathematics Society, Volume 74, 1953.

[111] J. Rose, N. Swamy, and M. Hicks. Dynamic Inference of Polymorphic Lock Types. *Science of Computer Programming*, Volume 58, 2005.

[112] I. Sommerville. *Software Engineering, Sixth Edition*. Addison Wesley, 2000.

[113] C. Simonyi. Hungarian notation. MSDN library. *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs600/html/HungaNotat.asp*

[114] J. M. Spivey. *The Z Notation: A Reference Manual, Second Edition*. Prentice-Hall, 1992.

[115] SSL Specification, Third Version. *http://wp.netscape.com/eng/ssl3*

[116] Static Driver Verifier: Finding Bugs in Device Drivers at Compile-Time. *Windows Hardware Engineering Conference*, April 2004.

[117] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary Transformation in a Distributed Environment. *Microsoft Research Technical Report*, MSR-TR-2001-50, April 2001.

[118] R. E. Strom and S. Yemini. Typestate: a Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*, Volume 12, January 1986.

[119] M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. *Symposium on Operating Systems Principles*, October 2003.

[120] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, April 2003.

[121] W. Weimer and G. Necula. Mining Temporal Specifications for Error Detection. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 2005.

[122] E. J. Weyuker and T. J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, May 1980.

[123] J. Whaley, M. C. Martin, and M. S. Lam. Automatic Extraction of Object-Oriented Component Interfaces. *International Symposium on Software Testing and Analysis*, July 2002.

[124] T. N. Win, M. D. Ernst, S. J. Garland, D. Kirli, and N. Lynch. Using Simulated Execution in Verifying Distributed Algorithms. *Software Tools for Technology Transfer*, July 2004.

[125] T. Xie and D. Notkin. Tool-Assisted Unit-Test Generation and Selection Based on Operational Abstractions. *Automated Software Engineering Journal*, Volume 13, July 2006.

[126] J. Yang and D. Evans. Dynamically Inferring Temporal Properties. *Workshop on Program Analysis for Software Tools and Engineering*, June 2004.

[127] J. Yang and D. Evans. Automatically Inferring Temporal Properties for Program Evolution. *International Symposium on Software Reliability Engineering*, November 2004.

[128] J. Yang and D. Evans. Automatically Discovering Temporal Properties for Program Verification. Technical Report, Department of Computer Science, University of Virginia, 2005.

[129] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. *International Conference on Software Engineering*, May 2006.

[130] J. Yang. *Automatic Inference and Effective Application of Temporal Specifications*. Ph.D. dissertation, University of Virginia, Department of Computer Science, May 2007.

[131] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. *Symposium on Operating System Design and Implementation*, December 2004.

[132] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants. *International Symposium on Microarchitecture*, December 2004.