In *Fifteenth IEEE International Symposium on Software Reliability Engineering* (ISSRE 2004). 2-5 November 2004, Saint-Malo, France.

Automatically Inferring Temporal Properties for Program Evolution

Jinlin Yang David Evans
Department of Computer Science
University of Virginia
{jinlin, evans}@cs.virginia.edu

Abstract

It is important that program maintainers understand important properties of the programs they modify and ensure that the changes they make do not alter essential properties in unintended ways. Manually documenting those properties, especially temporal ones that constrain the ordering of events, is difficult and rarely done in practice. We propose an automatic approach to inferring a target system's temporal properties based on analyzing its event traces. The core of our technique is a set of pre-defined property patterns among a few events. These patterns form a partial order in terms of their strictness. Our approach finds the strictest properties satisfied by a set of events based on the traces. We report results from experiments on two sets of programs: student solutions for a class assignment, and several recent versions of OpenSSL. Comparing properties inferred from different implementations led us to discover important behavioral differences which revealed flaws in the programs. Differences in automatically inferred temporal properties can provide useful information to programmers evolving complex, often unspecified, programs whose correctness depends on preservation of undocumented temporal properties.

1. Introduction

A common problem is ensuring that changes introduced in program maintenance do not change the program's behavior in unexpected ways. In particular, changes should not alter important properties of the previous version of the program on which clients may rely. Unfortunately, most programs do not have specifications, so programmers modifying programs often do not know what those important properties are. We present an approach for largely automating the task of discovering important property differences. We focus on temporal properties, since they provide good opportunities for automatic inference and analysis, and because satisfying certain temporal properties is essential for the correctness of many programs, yet they are particularly hard for programmers to document and test manually.

A temporal property defines the sequence in which events take place [27]. Temporal properties are especially important in concurrent programs in which threads interact through shared objects and messages. Writes from different threads to a shared object are mutually excluded using mechanisms such as locks to ensure that events are ordered consistently. While such properties are fundamental to program correctness, they are rarely documented or specified. Even when they are, it is extremely hard to assure them by inspection or testing due to the huge number of ways threads might interleave with each other.

Previously, we proposed automatically inferring interesting temporal properties from execution traces [30]. The key components of our approach are the development of a set of extensions to the response property pattern [11] and an algorithm that can automatically infer the strictest pattern a set of events satisfy. The key contribution of this paper is an experimental analysis of our approach that illustrates its effectiveness as an aid to program evolution. We report on experiments using our prototype tool on two sets of programs: student implementations from a software systems course assignment; and multiple versions of OpenSSL [25], a widely used open source implementation of the SSL protocol. Our tool was able to infer interesting temporal properties in both families of programs, and confirm that all implementations satisfied some important properties. By analyzing the differences between the temporal properties our tool inferred for each program, we were able to detect bugs in one of the student implementations, identify behavioral differences between different versions of OpenSSL, and identify security vulnerabilities in the OpenSSL implementation.

Section 2 provides a brief survey of related work. Section 3 presents our approach, which is a slight refinement of the approach presented in [30]. Section 4

provides an overview of our implementation and experiments. Sections 5 and 6 report on results from our two experiments, and Section 7 concludes.

2. Related work

The task of understanding differences between two versions of a program has been long recognized as an important software engineering problem. The majority of previous work has focused on static approaches, ranging from simple syntactic differencing [22] to more complex static approaches that consider program semantics using dependences [4, 18, 19, 21]. Static approaches provide guarantees that are not possible for dynamic approaches, since they can reason about all possible executions of the two programs. However, they are limited in the kinds of properties that can be considered since determining whether two programs are different for most interesting program properties is an undecidable problem. Our focus is on dynamic approaches, so we do not include a detailed survey of static differencing work here.

Our work is mainly inspired by Ernst's work on dynamically inferring program invariants [12, 23]. While their work focuses on the value relationships among variables which are more relevant to dataflow, ours focuses on a program's temporal properties such as the execution sequences of methods which are more relevant to control flow.

Specification mining [1, 2] discovers temporal specifications a program must satisfy when interacting with an application programming interface (API) or abstract data type (ADT) using machine learning techniques. It extracts scenarios from execution traces based on a dependency analysis and then uses a probabilistic finite automaton (PFSA) learner to infer specifications. Our approach differs from specification mining in several aspects. Our techniques target general types of events whereas specification mining is limited to API and ADT events. We use a template matching approach so that a large number of long execution traces can be analyzed, whereas their PFSA learner is limited to fairly short traces. Specification mining produces specifications that may be inconsistent with the execution traces, and they use a dynamic checker to verify the specification against those execution traces. Further, specification mining requires substantial guidance from an expert (e.g. to define which attributes of interactions may define objects, select seed events for dependency analysis, and to identify which attributes may use objects). Our goal is to develop techniques that are as automatic as possible.

Whaley et al. developed two techniques, one static and the other dynamic, for inferring sequencing models of methods of a component [29], and built a dynamic model checker to check if the code conforms to the models discovered. By slicing on methods accessing the same field of a class, they are able to discover a precise sub-model for such methods. They did not attempt to develop techniques to find the strictest pattern any two methods can have. Their dynamic approach adds a transition to the model upon finding one instance of such a transition. Ours only considers a temporal property to be valid if all the traces have it. Further, we focus on finding the precise relationship between a few (i.e. two or three) events by systematically examining all possible candidate patterns.

Cook et al. developed statistical techniques to discover patterns of concurrent behavior from event traces [7]. Their techniques first extract a thread model out of the event traces, and then infer points of synchronization and mutual exclusion based on that model. Our approach is distinguished from theirs in that the temporal properties inferred by our method are more general, and their approach only uses a single event trace, while ours is based on many event traces.

Static analysis techniques like software model checking can verify temporal properties on a closed model of the system. They can examine a temporal property on all possible execution paths with certain constraints (e.g. the range of variables) to find faults in a system that are hard to detect using traditional methods. Software model checkers [3, 5, 8, 14, 16] have been successfully applied to check many real world systems. However, model checkers require specifications of properties to check such as assertions about valid states of the system and temporal properties. Temporal properties are represented using some formalism such as Linear Temporal Logic (LTL) [27]. The specification language is usually different from the language in which the system is written, and is often difficult to understand. Further, the specification is usually defined on the model, whereas it is best understood in terms of the implementation. Thus, to define a temporal property, one must be familiar with the formalism and be able to translate and redefine properties based on the structure of the model. This process can be very challenging and error-prone, even for experienced users. Holzmann showed how tricky and difficult it is to define a simple temporal property using LTL [17].

Dwyer et al. developed a set of temporal property patterns based on a case study of hundreds of real property specifications [11]. They integrated those patterns into their Bandera toolset [8] so that users can express a temporal property in the Bandera Specification Language [9]. That property is mapped into the underlying formalism the chosen model

checker accepts. Their patterns are too imprecise to describe some interesting properties. We derived a number of variations of their patterns by adding more constraints. To ease the task of formulating a property, we developed techniques to automatically search the strictest pattern matching the event traces. Inferred properties can then be subjected to validation by users or model checkers.

Havelund used information obtained from runtime analysis to guide model checking of Java programs [15]. Two dynamic analysis algorithms to detect race conditions and deadlocks run first. If those analyses report any warnings, the Java PathFinder model checker [14] is used to check the suspected threads specifically. Their approach showed that runtime analysis information can be used to pinpoint the problematic point in the program such that the state space for large program can be significantly pruned. Our approach is more systematic and general in that a broad category of temporal properties can be automatically derived and checked along a program's control flow.

3. Approach

The main components of our approach are shown in Figure 1 and described briefly here. For more details, see [30].

First, we instrument the target program to monitor a set of key events. This could be automated by instrumenting all program expressions of interest such as system calls or procedure calls.

Next, we execute a test suite on the instrumented program and collect execution traces. To generate execution traces, we need to execute the program. If the target program has a set of test cases, we could just use them. Otherwise, we can generate a test suite either by some automated test generator or manually.

Then, we instantiate candidate temporal property patterns. A property pattern [8] is an abstraction of a set of commonly used temporal properties. We are interested in the Response pattern describing the cause-effect relationships between two abstract events P and S: P's occurrence must be followed by the occurrence of S. For example, SPPSS is an event sequence that satisfies this pattern, but SPPSSP does not since no S responds to the last P. We use a Quantified Regular Expression (QRE) [24] to describe the Response pattern as: [-P]*(P[-S]*S[-P]*)*. If we assume all events other than P and S are filtered from the trace, this is equivalent to S*(PP*SS*)*. QREs are similar to regular expressions: [-] is the exclusion operator ([-P] specifies any event in the alphabet except P). The * (Kleene star) and () (grouping) operators have their normal meanings.

The Response pattern is very imprecise in that it allows several causing events (P) to share one effect event (S), one causing event to have multiple effect events, and effect events to happen before any causing event. As a result, knowing two events satisfying this property does not give us much useful insight into a program's temporal behaviors.

To solve this problem, we developed the variations on the original Response pattern shown in Table 1. Let $L(A)$ represent all event traces satisfying pattern A. Given two patterns A and B, if $L(A) < L(B)$ (that is, all event traces that satisfy B satisfy A, but at least one event trace that satisfies B does not satisfy A) we say A is stricter than B.

The eight patterns form a partial order in terms of their strictness as shown in Figure 2. To determine the strictest pattern satisfied by a pair of events, we first determine which of the CauseFirst, OneCause and OneEffect patterns they satisfy. Then we can use the relationships among the patterns to infer the strictest pattern. For example, if a pair of events satisfies OneCause and OneEffect, but not CauseFirst, we can infer that the strictest satisfied pattern is EffectFirst.

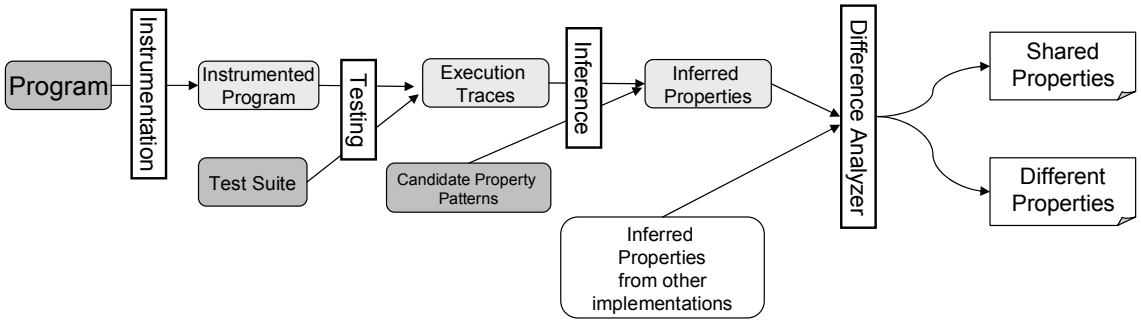

Figure 1. System overview

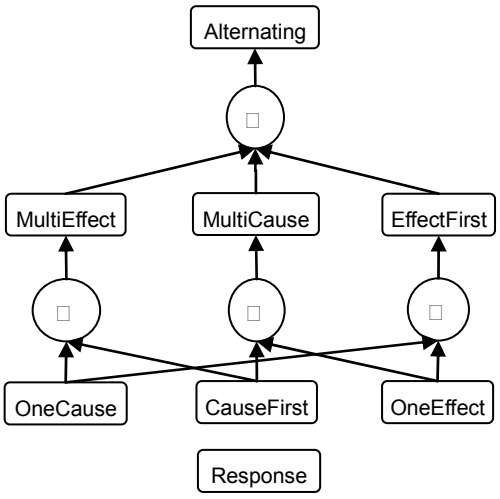

Figure 2. Partial order of properties

We obtain a concrete property by replacing each abstract event with a selected monitored event. We can get a set of concrete properties by replacing the abstract events with those monitored events of interest to us. If a pattern is parameterized by m abstract events and we monitor n events, there are n^m possible instance properties. For now, we only consider patterns where m is 2 so it is feasible to check all possible patterns as long as the number of monitored events is fairly low.

For each concrete property, we check if the collected event traces satisfy the property to determine the strongest temporal property satisfied by each pair of events. Then, we compare the inferred properties for this version with inferred properties for other versions, and produce a report describing the shared and different properties. The difference analyzer simply compares the strictest property inferred for each pair of events, and identifies events for which different properties are inferred for different program versions.

4. Experiments

To evaluate our approach, we built a prototype implementation and conducted experiments using it on two tasks related to program evolution. The next two subsections describe our prototype implementation and an overview of the experiments. Sections 5 and 6 describe each experiment and discuss our results.

4.1. Prototype implementation

Our current implementation automates all steps in the process except the instrumentation. Hence, to use our technique we must either instrument programs to generate event traces or provide a mapping from program output logs to event traces. In future work, we plan to build an instrumentation tool and event mapper that automates both processes.

We implemented our inference algorithm in a 900-line Perl script. To enable efficient pattern checking, we encode the event sequences into binary strings using the Huffman coding algorithm [10] implemented using the CPAN Algorithm::Huffman module [20]. Next an inference procedure processes the encoded traces to find the strictest pattern satisfied by any two events.

Finally a post-processing subroutine synthesizes the Alternating properties to present them in a more succinct and easier-to-understand form. When there are a large number of Alternating properties, it can be difficult for programmers to inspect and synthesize them manually. We developed an algorithm to automatically synthesize Alternating Chains. For example, if the alternating patterns A→B and B→C are inferred, they will be synthesized as an event chain A→B→C.

Programmers may use the results in several ways. They may compare the result with the informal specification by hand. If properties are inferred that are not expected according to the program specification, that reveals either a fault in the implementation, an inadequacy of the test suite, or a misunderstanding of the specification. If specified properties covered by the candidate property patterns are not inferred, it reveals a fault in the implementation. The inferred properties may also be used as input to a model checker. The model checker may validate the properties, or may find counterexamples to them. In cases where counterexamples are found, this may reveal a bug in the program or an important weakness in the test suite:

Table 1. Temporal property patterns

Name	QRE	Valid Examples	Invalid Examples
Response	*S*(PP*SS*)**	*SPPSS*	*SPPSSP*
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*

a property which is always true of the test executions is not actually guaranteed by the program. Both of these uses require substantial effort and expertise from the programmer, either in inspecting properties directly or using a model checker.

Our experiments focus on using our temporal property inference technique in a more automated way to support program evolution. Instead of validating or inspecting the inferred properties, we simply compare them to the properties inferred from some other instance. We compare properties inferred from different versions of a program using the same test suite. We could use similar techniques to compare test suites by keeping the program constant and comparing properties inferred from different test suites.

4.2. Hypotheses

To evaluate our approach, we did experiments on two families of programs: student implementations of a multithreaded programming assignment in a graduate systems software course, and archived versions of OpenSSL [25]. In both cases, all programs in the family were designed to implement the same informal specification. Hence, any differences in the temporal properties are likely to be important.

The first hypothesis we want to evaluate through these experiments is that our approach can recover a useful temporal specification from an implementation in the sense that most of the desirable properties can be inferred. If this is true, our tool can help new developers understand the temporal behaviors of a legacy system.

Our second hypothesis is that our approach is useful for supporting program evolution. Our goal is to help programmers ensure that desirable temporal properties are preserved and changes to temporal properties are noticed and investigated. Our hypothesis is that the temporal properties inferred for a program using our approach can be used as a signature of the program's temporal semantics to help programmers achieve those goals. So in each experiment, we compared the properties inferred for different versions of a program. Upon finding any discrepancy between the properties inferred for two programs intended to meet the same specification, we tried to identify its cause. Property differences may be benevolent (the programs behave differently, but in ways that are both consistent with the desired behavior) or may reveal variations in the way tests are run (such as operating system scheduling decisions) or faulty implementations. Finding any of these problems will then be useful for improving the test suite, the implementation, or the specification.

5. Tour bus simulator

An early assignment in a graduate software systems course taught at the UVa in fall 2003 asked students to write a multithreaded program to simulate the operation of city bus. Students were given an informal specification of the program, paraphrased below:

> Write a program that takes three inputs:
>
> - n, the number of people,
> - C, the maximum number of passengers the bus can hold (C must be ≤ 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 required specific input and output formats, which greatly facilitates automatic testing. Executing `bus -n <people> -C <passengers> -T <trips>` runs the program. Figure 3 shows a typical execution.

5.1. Properties

A correct solution must satisfy several temporal properties including:

1. The bus always rides with exactly C passengers.
2. No passenger will jump off or on the bus while it is running.
3. No passenger will request another trip before getting off the bus.
4. All passengers get off the bus before passengers for the next trip begin getting on.

We were given eight different submissions. All of these submissions had been previously evaluated by a grader both by looking at a design document and examining the implementation code, and by inspecting output from test executions. All of the submissions we used had been considered correct by the grader.

Because the existing outputs are already events of

```
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 3. Sample output from `bus -n 2 -C 1 -T 1`

interest to us, there is no need to instrument the programs. Instead, we mapped the output logs directly to event sequences. In our mapping, we considered these five events:

1. wait ("Bus waiting for trip n" log entries)
2. drives ("Bus drives around Charlottesville")
3. stops ("Bus stops for the day")
4. gets in ("Passenger *n* gets in")
5. gets off ("Passenger *n* gets off")

Note that the numbers of the trip and passenger are ignored in our event mapping. This makes the number of different types of events to consider small, but means certain temporal properties cannot be inferred from our event sequences (such as, "Passenger 3 gets in" must be followed by "Passenger 3 gets off"). We describe experiments using an alternate mapping that preserves passenger numbers at the end of this section.

5.2. Results

We ran each solution 100 times with randomly generated parameters ($20 < C \leq 40$, $C+1 \leq n \leq 2C$, and $1 \leq T \leq 10$). Our tool inferred exactly the same set of temporal properties for seven out of the eight submissions. Table 2 summarizes the results.

The Alternating pattern is inferred for wait and drives for seven of the programs, but not for the other program. The strongest property inferred for wait and drives is MultiEffect which means there can be multiple drives events for each wait event. Since seven out the eight programs satisfy the stronger property (as well as the sample output in Figure 3), we suspect that there is a bug in the other solution.

It was not obvious to us where the problem is by simply inspecting the code. So, we examined the analysis tool output which identified a specific trace in which wait and drives does not satisfy the Alternating pattern. This led us to find the bug in the go_for_drive code shown in Figure 4. At the end of that method, the bus thread releases the lock. This effectively allows

Table 2. Properties inferred

Pattern	Correct Versions	Faulty Version
Alternating	wait→drives	
MultiEffect	drives→gets off wait→gets off wait→gets in	wait→drives wait→gets in
MultiCause	drives→stops gets in→drives gets in→stops wait→stops gets off→stops	gets in→drives
CauseFirst	gets in→gets off	gets in→gets off drives→gets off wait→gets off

```
void go_for_drive() {
  pthread_mutex_lock (&mutex[mutex_lock]);
  if (num_riders < capacity) {
    printf ("Bus waiting for trip %d\n", num_trips);
    pthread_cond_wait (&cond[cond_shuttle_full],
                       &mutex[mutex_lock]);
  }
  printf ("Bus drives around Charlottesville\n");
  sleep (3);
  pthread_cond_broadcast (&cond[cond_ride_over]);
  num_riders = 0;
  num_trips--;
  pthread_mutex_unlock (&mutex[mutex_lock]);
}
```

Figure 4. Faulty code excerpt

passenger threads to compete for the lock and to possibly get in the bus before the bus starts waiting for passengers and output the Bus waiting for trip message that corresponds to the wait event. In most cases, the bus can successfully obtain the lock before it has been filled to capacity (the condition num_riders < capacity is true), so it can generate the wait event. However, the bus can be already full when the bus obtains the lock (if num_riders ≥ capacity), in which case it does not produce the wait event. In such situations, wait and drives do not alternate with each other and the Alternating pattern does not hold. This is a bug because it is possible for passengers to get in before the bus is waiting for trip. One way to fix it would be to use a conditional variable to synchronize the bus and the passengers and to make sure the bus generates the wait event before it broadcasts that condition.

The second property difference found in Table 2 is for the drives and gets off events. In seven of the implementations they satisfy the MultiEffect pattern, but in the other implementation they satisfy CauseFirst. The CauseFirst property means it is possible for the bus to drive around Charlottesville more than once without allowing passengers to get off between these trips. This is again a bug of missing synchronization between the bus and the passengers. As shown in Figure 4, the bus broadcasts that the ride is over to all passengers after it drives around the city. 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, wait and gets off satisfying CauseFirst instead MultiEffect, was caused by the same bug as discussed in the second one. Since the faulty program does not print out the stops event at all, none of the patterns related to the stops event, which appeared in the other seven versions, was discovered.

Our original event mapping lost all information about which passenger each gets in and gets off event concerned. We also ran our prototype with a different event mapping in which each "Passenger *n* gets in" log entry corresponds to a different event for each value of

n, and similarly for "Passenger *n* gets off". This enables us to detect the Alternating pattern between Passenger *i* gets in and Passenger *i* gets off (for all values of *i* corresponding to passengers) if this is true. We reran the analyzer and found that all solutions satisfy this property. That is, our tool correctly inferred the property that no passenger will request another trip before getting off the bus.

5.3. Discussion

Our prototype tool was able to automatically discover interesting differences in the temporal properties of the eight programs, revealing flaws in one of the programs. In addition, it was able to correctly infer patterns corresponding to three of the four desirable properties implied by the problem specification. It was not able to infer the property that the bus always rides with C passengers. To do this, we would need to analyze not just the ordering but also the count of events. Our current property patterns are not sufficiently expressive to detect this, since they only deal with pairs of events. We plan to include some event count analysis in our future work. We intend to develop a library of event count patterns can be developed similar to the event ordering patterns we use now.

6. OpenSSL

Our second experiment considered recent versions of OpenSSL. The Secure Socket Layer (SSL) protocol provides secure communication over TCP/UDP using public key cryptography [13]. We focus on the handshake protocol that performs authentication and establishes important cryptic parameters on both client and server sides before data are transmitted between them. OpenSSL, written in C, is a widely used open source implementation of SSL [25]. In our experiments, we used our tool to automatically infer the temporal properties of implementation of the handshake protocol in multiple versions of OpenSSL.

Chaki et. al used MAGIC, a C model checker that can automatically extract a model from a C program, to check OpenSSL's implementation of the handshake protocol [6]. They manually constructed the properties to check from the specification and they only checked version 0.9.6c of OpenSSL.

6.1. SSL handshake protocol

Figure 5 shows the client (left) and server (right) events in the SSL handshake protocol, which was derived from the SSL specification [13]. The three boxes with dashed outlines contain internal states created in the OpenSSL implementation but not

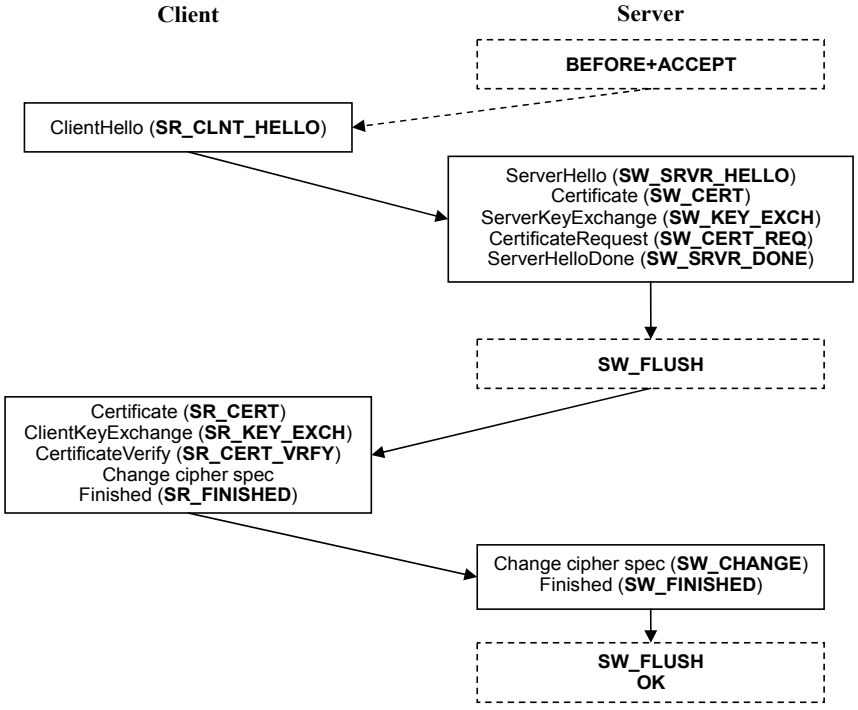

Figure 5. SSL handshake protocol states

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 6. A server event trace of normal handshake process

required in the SSL specification. The remaining boxes contain sequences of states corresponding to messages required by the SSL handshake protocol. The server state in the OpenSSL implementation corresponding to the beginning of either receiving the corresponding message from the client or sending the message to the client for each message is shown in the parentheses. For example, if the server is in the BEFORE+ACCEPT state and receives a ClientHello message from a client, the server enters the SR_CLNT_HELLO state. Receiving the change cipher spec message is not part of the handshake process, so that message has no corresponding state change. It does have a corresponding state for sending (SW_CHANGE) in the implementation that is monitored in our experiment.

The handshake begins when the server receives a ClientHello message from a client. Then the server sends out five messages consecutively (corresponding to the states SW_SRVR_HELLO, SW_CERT, SW_KEY_EXCH, SW_CERT_REQ and SW_SRVR_-DONE). Next the server enters the SR_CERT state in which it tries to read certificate from the client (whether the client sends its certificate or not depends on if the server requires one in its certificate request message). Then the server reads consecutively four messages from the client (certificate, key exchange, certificate verify, 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 box, the server implemented several additional internal states which we 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 the end, OK is another internal state indicating that the server cleans things up in its side and is ready for transmitting data with client. A typical event trace is shown in Figure 6.

In the OpenSSL server implementation, the handshake process is encapsulated in a method. A server starts the handshake process by calling the ssl3_accept method which 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 method to monitor 15 states shown in Figure 5.

6.2. Testing process

Our experiment considered only properties of the server implementation, but in order to generate test data we needed to execute it with sample clients. We are 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. The OpenSSL client implementation starts the handshake process by calling the ssl3_connect method, which implements the protocol state machine in a similar fashion as ssl3_accept. We modified ssl3_connect so that after every state it may either behave correctly or enter some randomly selected state. For example, suppose the current state is A, after finishing task X, the state should be changed to B. In our modified version, the state would correctly transition to B with 95% probability, but with 5% probability would transition into a randomly selected different state instead. Our purpose is to create a client that sometimes behaves abnormally. Note that in most SSL server deployments, it is important how the server behaves even when clients misbehave (usually this means reporting an error and terminating the handshake).

We used as test harness a simple OpenSSL-based implementation of HTTPS protocol: wclient and wserver (version 20020110) developed by Eric Rescoria [28]. We added one more command-line option for wclient so that users can seed the random number generator with a specific integer to enable reproducibility of the experiments. We also modified wserver so that it only accepts one connection and exits after that. We also added handler functions for SIGSEGV and SIGPIPE signals which we observed in both client and server in our initial tryout. Our handlers simply printed out the name of corresponding signals and exited.

We obtained traces for six versions of OpenSSL: 0.9.6, 0.9.7, 0.9.7a, 0.9.7b, 0.9.7c, and 0.9.7d (released 17 March 2004, the latest version available as of 25 April 2004). For each version of OpenSSL, we ran both wclient and wserver in tandem 1000 times on two Redhat Linux machines and obtained 1000 traces for the server. For each execution, the client random generator was seeded with an integer from 1 to 1000 to obtain different abnormal client behaviors. We used the default keys and certificates supplied in the example program's package and the default choice of cryptic algorithm. We wrote shell scripts to automate the testing process. In the early stages of our experiment,

we observed that the server and client may deadlock each other (i.e. both waiting for message from the other). Upon encountering such situations, our client shell script would just kill the client process.

6.3. Results

First, we applied our tool to all 1000 traces. Version 0.9.7 generates 38 different types of events. It took 197 seconds to analyze the 1000 traces for this version on a laptop with 1.3 GHz Pentium-M CPU. The number of different types of events in the rest versions varies slightly from 36 to 38. Then we partitioned the traces into four groups: 1) correct client (i.e. the client did not change to any unintended state); 2) faulty client (i.e. the client changed 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 our tool separately to each group of traces.

All traces

Table 3 highlights three key differences among the Alternating patterns detected for all versions. First, the Alternating pattern between SR_KEY_EXCH and SR_CERT_VRFY (receiving key exchange message, and certificate verify message) appeared in all versions up to 0.9.7b, but did not appear in 0.9.7c and 0.9.7d. This is a result of a change added since version 0.9.7c to make the implementation conform to the SSL 3.0 specification (documented in the change log of version 0.9.7c [25]). Starting from version 0.9.7c, a server does not process any certificate message it receives from a client if it has not requested authentication of client. The condition of whether the server requests client authentication is recorded in a variable, which can be set using command line option. The server checks this variable to decide what its next state is after entering the receiving certificate state (SR_CERT). If the server does not require client authentication as in our experiment, it directly advances to receiving key exchange state (SR_KEY_EXCH). Otherwise, it first reads and examines the client's certificate before changing to that state. Our faulty client may send its certificate even if the server has not requested one, OpenSSL server before 0.9.7c noticed this and stopped handshake immediately: the SR_KEY_EXCH state was not entered at all. Server versions 0.9.7c and 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 a key exchange message but got a certificate message). So, in versions 0.9.7c and 0.9.7d, such traces ended up with a SR_KEY_EXCH event without a following SR_CERT_VRFY event. In contrast, such pattern was preserved in earlier versions. Hence, our tool successfully exposed an important change made to the implementation of handshake protocol in version 0.9.7c.

In the second row, we found that SW_CERT and SW_KEY_EXCH satisfy the Alternating property for all versions except 0.9.6. Investigating the traces showed sometimes, the server crashed after entering SW_CERT state with a SIGPIPE signal. This is apparently a critical bug in earlier version of OpenSSL which has been fixed in later ones.

In the third row, we found that only version 0.9.7 has the Alternating property between the SW_SRVR_DONE and SR_CERT events. Finding the problem turned out to be more tricky than we expected since the result appeared to be non-deterministic when we ran the server on a single test. Sometimes the SR_CERT event was generated, but sometimes it was not. This happened in all versions of server. Using the program's log messages we were able to find the cause, which happened to be a race condition. When the client changes to a false state and receives an 'unexpected' message from the server, it tries to send an alert message to the server to stop the handshake process. After that, the client disconnects the socket with server. If the client disconnects the socket while the server is sending messages, the server will get a sending error message. The server has not and will not get the alert message from the client now because the socket has been disconnected. The server will only be able to get the alert message if it has already finished sending messages to client and entered a receiving state. In our experiment, this receiving state is SR_CERT. If the server is able to enter this state before the client alert message is sent, this event and an alert message will both be printed out at the server. So, there is no guarantee in these implementations that an alert message will be received after sending. This important design decision was not documented in the specification and our approach successfully discovered it. In addition, this experience reveals the importance in testing under a variety of conditions. The scheduling

Table 3. Alternating properties satisfied by six versions of OpenSSL

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

decisions made by the operating system influence the temporal properties of multithreaded programs. Ideally, the test suite would be run using a scheduler that could be configured to produce a variety of schedules.

Correct clients

We detected the event chain shown in Figure 6 by analyzing the traces of server in which the client behaved correctly (that is, the 5% probability of switching to a random state was never selected). Although the number of such traces varies slightly among different versions, all versions agreed on the same pattern. This result is desirable because the pattern in Figure 6 conforms exactly to the SSL specification as discussed earlier. This demonstrates that the server implementations of the handshake protocol conform to the specification at the state-level along the path of OpenSSL's evolution. The only discrepancies between behaviors from different versions of the server occur when the client does not follow the protocol correctly.

Faulty clients without errors generated

Next we consider the set of traces corresponding to faulty clients that (surprisingly) did not generate any error event on either the server or client. These traces recorded behavior from clients that jumped to a random state at some point during their execution, but did not lead to either the client or server reporting an error or failing to complete the handshake process.

Again all six versions agreed on the two event chains shown in Figure 7, though the number of such traces varies a little bit. These two chains closely follow the SSL specification about the normal handshake behavior of a server implementation. However, there are a few key distinctions between the patterns in Figure 6 and Figure 7.

We found that two Alternating properties that are present in Figure 6 do not appear in Figure 7. Instead, those event pairs had weaker patterns. First, SR_CERT and SR_KEY_EXCH satisfied the MultiCause pattern. Second, BEFORE+ACCEPT and SR_CLNT_HELLO satisfied the MultiEffect pattern. Figure 8 shows a trace that violated our expected Alternating properties. The key distinction between this trace and the traces produced using correctly behaving clients is that the

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 7. Inferred alternating chains for non-error faulty clients

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,)^2
SR_KEY_EXCH, SR_CERT_VRFY, SR_FINISHED,
SW_CHANGE, SW_FINISHED, SW_FLUSH, OK

Figure 8. Trace generated with a faulty client

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

Figure 9. Client trace corresponding to the server trace shown in Figure 7

eight-event sequence appears twice. Figure 9 shows the corresponding client events. The faulty client falsely changed its state to renegotiate (an internal state in the implementation of the client, not shown in Figure 4 since it is not part of the normal handshake process) instead of sending certificate (i.e. CW_CERT) after reading the five messages from server (ServerHello, Certificate, ServerKeyExchange, CertificateRequest, and ServerHelloDone). Then, the client started the handshake again by sending the client hello message which caused the server to repeat the hello stage of the handshake again. Although the handshake returned to normal and ended successfully after both parties repeated the hello stage twice, we found that the handshake can still be successful no matter how many times the hello stage is repeated. If a client always changes its state to renegotiate after receiving the server done message, the server and the client will enter an infinite loop.

Suspecting this could be exploited in a denial of service attack against an OpenSSL server, we reported it to the OpenSSL developers. They argued that it did not indicate a serious DOS vulnerability because the server loops infinitely only when an ill-behaved client keeps sending renegotiation requests. This is similar to letting too many clients attempt to connect to a server, which is a scenario that cannot really be prevented at the server.

Although the property is not a real vulnerability in OpenSSL server, it does reveal an interesting aspect of the implementation's behavior which is not documented in the SSL specification and certainly not obvious from the inspecting the code.

Segmentation fault

There are three traces that have segmentation fault in all servers prior to 0.9.7d. They resulted from the same faulty client, which sent out a change_cipher_spec instead of the normal client hello message at the very beginning of the handshake process. We examined the change log for 0.9.7d and found that this is due to a critical update [26], where a potential null-pointer assignment in the do_change_cipher_spec() function can cause the earlier versions of server to crash. Although this finding is not a result of comparing the temporal properties detected, it does show that using randomly behaving client to test server is powerful enough to uncover important problems.

Faulty client with other types of error

All servers agreed on the temporal properties inferred for traces within this category. This did not lead us to detect any interesting problems, but did confirm that the server versions handled misbehaving clients consistently.

6.4. Discussion

Our results support both hypotheses stated in Section 4.2. Using the temporal properties inferred for different program version we were able to recover the interesting temporal behaviors of the handshake protocol. Further, our approach successfully identified temporal property differences across OpenSSL versions that would be useful for understanding the behavioral differences between those implementations. In particular, we found that all implementations preserved the inferred temporal properties for well behaved clients, but that later versions of OpenSSL handled misbehaving clients in ways that preserved properties that were not preserved by earlier implementations when clients misbehaved. We believe knowledge of these properties, and the ability to automatically test them, will be useful to maintainers of OpenSSL.

However, we have not yet been able to successfully extract useful differences from the traces produced with faulty clients that produced errors. Because there are a large number of different types of errors, each of which appeared relatively infrequently, the traces appeared to be very irregular. One possible way to handle such situations is first partitioning the traces according to different types of error events, then applying our tool to the corresponding subset of traces. In this experiment, we only compared the temporal properties of multiple versions inferred from the same type of traces. Another possibility is to compare the temporal properties of the same version inferred from different types of traces. In addition, we may need to expand our set of property patterns to include three-event patterns to deal with error cases by allowing disjunction patterns. We plan to explore this further in future work.

7. Conclusion

We presented a prototype tool that automatically infers temporal properties of programs by analyzing test execution traces, and argued that such a tool can be a useful asset in reliable program evolution. Our experimental results demonstrate that our approach is able to automatically determine important temporal properties and identify differences that reveal interesting properties of programs.

The results from our two experiments give us reason to be optimistic that our approach can be a useful tool to aid program evolution. However, further work needs to be done before the approach can scale to large programs with many events and long execution traces. We plan to evaluate our approach on more systems where temporal properties are especially important including reactive systems, and embedded systems. Many parts of our approach, which should be automated, are still done manually including program instrumentation and identification of interesting results. For large systems, we need to selectively monitor interesting events to make our approach scale. An open problem is developing heuristics for automatically identifying such events. We will investigate this important question in our future research. Our current patterns are not able to express certain properties that are important for software reliability. We plan to develop some more advanced ones including patterns involving event counts to capture program's temporal semantics more precisely.

The effectiveness of any dynamic analysis depends on the comprehensiveness of the executions of a target program it examines. It is very important to understand how the quality of the test suite could impact the result of our analysis and what testing approaches are most effective for our analysis. We plan to conduct research to answer these questions.

In these experiments, we focused on program evolution, but our tool has other applications. One possibility is using it to produce input to a model checker. Then, we can employ a model checker to automatically check the temporal properties discovered. This can not only give us more confidence in the inference results, but also help undetected subtle problems by leveraging the existing sophisticated verification techniques.

Acknowledgments

This work has been funded in part by the National Science Foundation through NSF CAREER (CCR-

0092945) and NSF ITR (EIA-0205327) grants. We thank Marty Humphrey for providing the student programs used in the first experiment. We also thank Chengdu Huang for help with OpenSSL.

References

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2002

[2] G. Ammons, D. Mandelin, R. Bodik, and J. R. Larus. Debugging temporal specifications with concept analysis. *SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *8th International SPIN Workshop on Model Checking of Software*, May 2001.

[4] D. Binkley, R. Capellini, L. Raszewski, C. Smith. An implementation of and experiment with semantic differencing. *2001 IEEE International Conference on Software Maintenance*, Nov 2001.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and Hwang. Symbolic model checking: 1020 states and beyond. *4th Annual Symposium on Logic in Computer Science*, June 1990.

[6] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *25th International Conference on Software Engineering*, May 2003.

[7] J. E. Cook, Z. Du, C. Liu, and A. L. Wolf. Discovering Models of Behavior for Concurrent Workflows. *Computers in Industry*, pp. 297-319, Vol. 53, No. 3, April 2004.

[8] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, H. Zheng. Bandera: extracting finite-state models from Java source code. *22nd International Conference on Software Engineering*, June 2000.

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

[10] T. Cormen, C. Leiserson and R. Rivest. *Introduction to Algorithm*, MIT Press, 1990.

[11] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. *21st International Conference on Software Engineering*, May 1999.

[12] 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.

[13] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol, version 3.0. http://wp.netscape.com/eng/ssl3/.

[14] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, September 1999.

[15] K. Havelund. Using runtime analysis to guide model checking of Java programs. *7th International SPIN Workshop on Model Checking of Software*, August/September 2000.

[16] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, May 1997.

[17] G. J. Holzmann. The logic of bugs. *10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2002.

[18] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *SIGPLAN Conference on Programming Language Design and Implementation*, June 1990.

[19] S. Horwitz and T. Reps. The Use of Program Dependence Graphs in Software Engineering. *14th International Conference on Software Engineering*, 1994.

[20] Algorithm::Huffman module on CPAN. http://search.cpan.org/~bigj/Algorithm-Huffman-0.09/Huffman.pm

[21] D. Jackson and D. Ladd. Semantic diff: a tool for summarizing the effects of modifications. *International Conference on Software Maintenance*, October 1994.

[22] W. Miller and E. W. Myers. A File Comparison Program. *Software – Practice and Experience*, Vol. 15 No. 11, 1985.

[23] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: an empirical evaluation. *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November 2002.

[24] K. M. Olender and L. J. Osterweil. Cecil: a sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, March 1990.

[25] OpenSSL. http://www.openssl.org/

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

[27] A. Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science*, October/November 1977.

[28] E. Rescorla. *An introduction to OpenSSL programming (part 1)*. http://www.rtfm.com/ openssl-examples/, October 2001.

[29] 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.

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