

Automatically Inferring Temporal Properties

Jinlin Yang

Department of Computer Science

University of Virginia

jinlin@cs.virginia.edu

<http://www.cs.virginia.edu/jinlin>

1. RESEARCH AREA

Program analysis, software testing, temporal logic, formal methods.

2. BRIEF DESCRIPTION OF RESEARCH

Develop techniques for automatically extracting useful temporal properties from a program's execution traces to support the evolution, understanding, verification and debugging of software.

3. PROBLEM STATEMENT

Temporal properties specify constraints on the order in which a program's state changes. Satisfying certain temporal properties is essential for a program to be correct. Many formal verification tools have been developed to check certain temporal properties. Researchers have demonstrated the practical value of these tools in some domains (e.g. verifying the safety of device drivers) [8, 17].

Using these tools requires a specification of the temporal properties to be checked. Unfortunately, most systems are developed without a formal or informal specification. Formally specifying temporal properties by hand is a time-consuming, error-prone, and tedious task. Holzmann illustrated how difficult it is to specify a very simple temporal property in linear temporal logic [16]. Without an automatic way to discover those interesting temporal properties, it is extremely difficult if not impossible to check them. As a result, we still lack a systematic way to ensure that an implementation satisfies important temporal properties. This remains a big hurdle to the adoption of such verification tools.

4. PRIOR RESEARCH

There has been much research on the basic theory of temporal properties [19]. Temporal properties can be classified into two categories. The first one is called safety properties, violation of which could endanger a program's safety (e.g. no dereferencing of null pointers, no deadlocks, and no race conditions). Liveness properties belong to the second category, which indicates a program makes progress on the goal it tries to achieve, though violating them does not harm the system. Most research on checking temporal properties focused on safety properties, whereas how to effectively check liveness properties in large scale systems is still an open question. Another useful way to classify temporal properties is based on whether a property depends on knowledge of the program. Generic properties are those that don't depend on the implementation, whereas

application-specific properties are meaningful only to the particular implementation. For example, no program should dereference a pointer before it is initialized is a generic property, while the handshaking protocol in the OpenSSL specification is an example of application-specific property.

In the past decade, researchers have developed many tools to check temporal properties, especially safety properties. Some have been quite successful in specific domains. For example, Microsoft's PREfix/PREfast tool has been extensively used in its development process and caught many bugs that would, otherwise, be very hard to find. Microsoft also uses the Static Driver Verifier to check device drivers [17].

There are several works closely related to ours. Ernst developed a technique for dynamically inferring likely program invariants [12, 18]. Our work is distinct from his in that we focus on temporal invariants, which can be viewed as the likely sequence of occurrence of invariants discovered by Daikon. Ammons et al.'s work on mining specification [6, 7], Cook et al.'s work on discovering thread model [10], and Whaley et al.'s work on extracting component interfaces [22] all tried to solve the same problem as ours. The common characteristics of their approaches are that they are trying to extract a complete state machine representation, which is historically called the grammar inference problem and has been shown to be NP hard [13, 14]. This limits the scalability of these approaches to small programs. We are developing a lightweight approach that can scale well to large systems.

5. RESEARCH HYPOTHESES

This research aims to test four related hypotheses:

- We can automatically infer interesting temporal properties by analyzing a target program statically, dynamically, or a combination of both.
- Automatically inferring temporal property is scalable to large, real systems.
- Automatically checking those inferred properties using verification tools can lead to more reliable systems by detecting more bugs earlier.
- Automatically inferred properties can be used to assist program evolution.

6. PROPOSED SOLUTION

We propose a dynamic analysis that automatically extracts temporal properties from execution traces. Figure 1 shows how our approach works. First, we selectively instrument a target

program (P) to monitor the events and variables of interest. Second, we execute the instrumented program (P') against a set of test cases to produce execution traces. Third, our inference engine instantiates a number of parameterized temporal property templates based on the monitored events and states, tests what instance properties the traces match, and outputs those matched ones as properties the target program likely has.

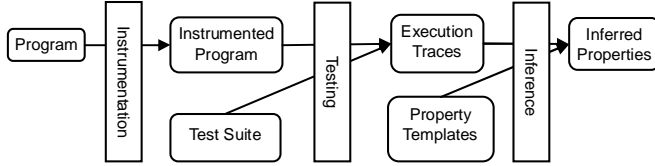


Figure 1. An overview of our approach

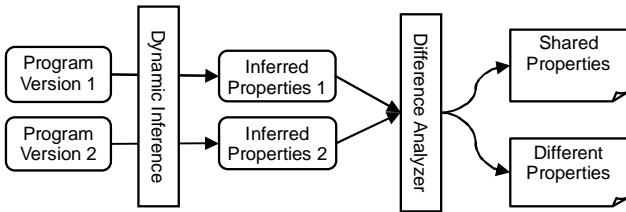


Figure 2. Use inferred properties in program evolution

One straightforward usage of the inferred properties is to facilitate program evolution. In Ernst’s terminology, all the dynamically inferred properties for a program are called its operational abstraction [15]. Assume we have two versions of a program, one of which is newer than the other. Figure 2 illustrates this scenario. We can run both versions against the same set of test cases, apply our inference tool to infer an operational abstraction for each version, and compare them. By checking the shared properties, we can decide whether some desirable properties have been preserved along the evolution of the program. By checking those properties where the two versions are different, we can conclude whether some intended improvement functions as expected.

One important limitation of our dynamic analysis is that the properties we infer might be violated by some other executions of a target program that we have not run. There are several contributing factors to this problem. First, our test cases do not cover all input sub-domains of the target program [21]. This problem can be solved by adding new test cases that belong to those uncovered sub-domains. Second, we might not execute all possible non-deterministic behaviors. For example, running a multi-threaded program once only executes one possible thread interleaving scenario.

To detect those false positives, we can leverage a powerful verification tool to further validate the inferred properties. For example, a software model checker can explore all possible thread interleaving scenarios under certain inputs, which can be employed to find if an inferred property still remains true in other unexecuted interleaving scenarios [9]. Figure 3 illustrates how this works. There are several benefits of such an approach. First, it can be largely automated. We only need to convert an inferred property into the format accepted by a model checker. Second, if a model checker finds that an inferred property is violated, it will produce a counter-example. If this is a desirable property, we find a bug in the program and the counter-example provides a useful hint for debugging. Third, this is a more economic way to take advantage of the power of a model checker. Although software model checking techniques have made significant progress, it is still impossible to use it to explore all possible properties. The inferred properties serve as a good starting point.

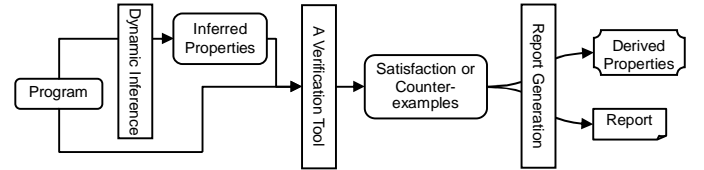


Figure 3. Use inferred properties in program verification

We summarize several key challenges that need to be addressed and our current progress on each of them as follows:

First, what events and states do we monitor? A naïve way is to monitor all possible events and states. Unfortunately, this won’t scale to large systems, so we need an automatic way to identify those interesting events and states. Currently we only observe the entrance and exit events of public methods of a class.

Second, what test inputs do we select? If the target program comes with a set of regression test cases, we can just use them. When there are no such test cases, or the existing test suite does not serve our purpose, we need to generate test inputs. In our experience so far, we found *Bounded Exhaustive Testing* (BET) to be very effective [20]. BET generates test inputs with some bounded value for each of its parameters. For example, if we want to analyze a library, we can create a test harness that create up to n threads, each of which executes a sequence of randomly chosen actions from the library’s interface.

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	$(PP^*SS^*)^*$	$PPSS$	$SPSS, SPSS$
OneCause	$S^*(PSS^*)^*$	$SPSS$	$PPSS, SPSS$
OneEffect	$S^*(PP^*S)^*$	$SPPS$	$PPSS, SPSS$

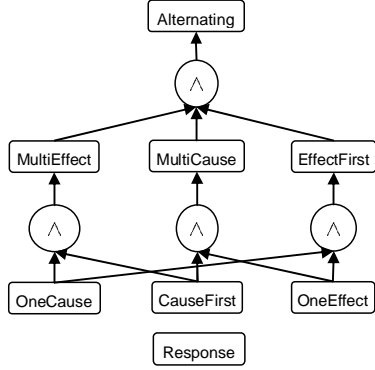


Figure 4. Partial order of properties

Third, what property templates do we use? It is crucial that these templates capture properties people care about, but otherwise would not have identified. In our preliminary work, we developed seven patterns based on 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 [11, 23]. We use regular expression to represent these patterns. For example, $[-P]^*(P[-S]^*S[-P]^*)^*$ is the regular expression for Response pattern. If we filter all events other than P and S from the traces, the Response pattern can be simplified as $S^*(PP^*SS^*)^*$. Table 1 shows the seven other patterns we developed. They form a partial order in terms of their strictness as shown in Figure 4 (we say a pattern A is stricter than another pattern B if $L(A) < L(B)$, where $L(A)$ means all the strings accepted by A). In particular, OneCause, CauseFirst, and OneEffect are the three *primitive* patterns that are the weakest among the seven. Above them come MultiEffect, MultiCause, and EffectFirst. Alternating is the strictest pattern among them. Our inference engine tries to find the *strictest* pattern any two events or states can satisfy. We take advantage of the partial order among the patterns by first testing which of the three primitive patterns they satisfy and then deducing the strictest pattern. For example, if two events satisfy all three primitive patterns, then Alternating is the strictest pattern they can satisfy.

Fourth, can we develop an efficient inference algorithm that scales to large traces with a reasonably large number of events? This is essential for our approach to scale to real world applications. Suppose there are N events, and the total length of trace is L, our current algorithm has time complexity $O(NL)$ and space complexity $O(N^2)$. In our preliminary experiments, we found that only a very small fraction (less than 0.1) of the N^2 event pairs satisfies at least one of the three primitive patterns. We can take advantage of this sparseness by using a hash table representation.

Fifth, how do we prioritize inferred properties? It is likely that we will end up with a very large number of properties when applying our approach to any real system. We are implementing and testing two heuristics for ranking properties. First is the frequency of the occurrence of the events. The more frequently an event occurs in the trace, the less likely that a corresponding pattern occurs by chance. Second we rank those properties that can be easily identified by static analysis lower. The strength of dynamic analysis is that it can have more precise information

than static analysis. So those properties cannot be easily inferred by static analysis should be more interesting.

Sixth, how can we effectively use existing verification tools to check those inferred properties? Although verification tools have made significant progress tackling the state explosion problem, our preliminary experience shows that it is still very difficult to use them to check our properties without some abstractions. For example, many model checkers provide ways to mark those irrelevant sequences of statements as an atomic step to reduce the number of states. Currently we manually mark blocks of statements as atomic, which we plan to automate in future work.

7. EXPECTED CONTRIBUTIONS

We hope to make the following contributions:

- An approach to extracting a useful temporal specification of an implementation.
- A set of useful property patterns.
- An efficient algorithm for inferring temporal properties.
- Heuristics for identifying interesting temporal properties.
- An effective approach to help develop more reliable software by assisting program evolution and verification.

8. EVALUATION

To evaluate our approach, we are implementing a dynamic inference tool called Terracotta¹. It is written in Java and currently has 8K lines of code. We use JRat as our instrumentor for Java programs [3]. Terracotta can take program traces and automatically match them against the pre-defined property templates. It also implements the two heuristics discussed in previous section.

To test our first and second hypothesis, we plan to apply our tool to some real systems that have some form of specification of their temporal behaviors, and compare our inferred specifications against the provided ones. Candidate systems include OpenSSL, which implements the SSL specification, and JBoss, which implements the J2EE specification [1, 2, 4, 5]. The largest traces we have tried so far have tens of millions of entries, of which there are around 3,000 distinct events. Terracotta was able to analyze those traces in 10 hours.

To evaluate our third hypothesis, we plan to apply our tool to some domains that verification tools have been found useful. In particular, we plan to apply our tool to infer properties for device drivers or the APIs they call and use MOPS to check them [8].

For testing our fourth hypothesis, we have done experiments on six versions of OpenSSL, a widely used implementation of the SSL specification, and eight submissions by different groups to a student program assignment [24]. The results are promising. In particular, comparing the inferred properties of different versions led us to expose the fact that these versions all satisfy the desired temporal specification, detect some fixed bugs in the earlier

¹ Terracotta is available from <http://www.cs.virginia.edu/terracotta>

version of OpenSSL, and some intended improvements applied to an earlier version. The same approach also helped us detect a subtle bug in one submission of the program assignment that the grader did not discover.

REFERENCES

- [1] J2EE. <http://java.sun.com/j2ee/index.jsp>
- [2] JBoss. <http://www.jboss.org>
- [3] JRat. <http://jrat.sourceforge.net/>
- [4] OpenSSL. <http://www.openssl.org/>
- [5] SSL specification version 3. <http://wp.netscape.com/eng/ssl3/>
- [6] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2002
- [7] 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.
- [8] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. *ACM Conference on Computer and Communications Security (CCS)*, November 2002.
- [9] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
- [10] 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.
- [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] E. Gold. Language identification in the limit. *Information and Control*, 10, 447-474, 1967.
- [14] E. Gold. Complexity of automatic identification from given data. *Information and Control*, 37, 302-320, 1978.
- [15] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. *International Conference on Software Engineering*, May 2003.
- [16] G. J. Holzmann. The logic of bugs. *10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2002.
- [17] 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.
- [18] 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.
- [19] A. Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science*, October/November 1977.
- [20] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. *International Symposium on Software Testing and Analysis*, July 2004.
- [21] E. J. Weyuker, and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, May 1980.
- [22] 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.
- [23] J. Yang and D. Evans. Dynamically Inferring Temporal Properties. *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, June 2004.
- [24] J. Yang and D. Evans. Automatically Inferring Temporal Properties for Program Evolution. *15th IEEE International Symposium on Software Reliability Engineering*, November 2004.