Dynamically Inferring Temporal Properties

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

ABSTRACT

Model checking requires a specification of the target system's desirable properties, some of which are temporal. Formulating a temporal property of the system based on either its abstract model or implementation requires a deep understanding of its behavior and sophisticated knowledge of the chosen formalism. This has been a major impediment to documenting and verifying temporal properties. We propose a dynamic approach to automatically infer a program's temporal properties based on a set of property pattern templates. We describe a preliminary implementation of this approach, and report on our experience using it to discover interesting temporal properties of a small program.

Categories and Subject Descriptors
D.2.4 [**Software/Program Verification**]

General Terms
Experimentation, Verification.

Keywords
Invariants, temporal properties, concurrent programming, dynamic analysis, property patterns.

1. INTRODUCTION

Satisfying certain temporal properties is essential for the correctness of many programs. A temporal property defines the sequence in which events take place [16]. For example, one of the key properties for a program manipulating pointers is that a pointer must be initialized before it can be dereferenced. 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. The progression of different threads is also synchronized using programming constructs. For example, the classic producer-consumer problem involves two threads sharing a finite data buffer. The producer creates new data and inserts them into the buffer, while the consumer reads and removes data from the buffer. If the buffer is full, the producer should wait before inserting a new element. Similarly, the consumer should block when reading from an empty buffer until the producer adds new data to it.

While such properties are fundamental to the correctness of an implementation, it is extremely hard to assure them by inspection or testing due to the huge number of ways threads might interleave with each other. Static analysis techniques like software model checking work 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, 4, 6, 10, 12] have been successfully applied to check many real world systems.

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) [16]. 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. As a result, 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 [11].

We propose to automatically infer interesting temporal properties from execution traces. The key contributions of our work are the development of a set of extensions to the response property pattern [8] and an algorithm that can automatically infer the strictest pattern a set of events satisfies. Section 3 describes the steps of our approach. Section 4 describes our results applying a preliminary implementation of our approach to a version of the producer-consumer problem and a faulty version of the same program. Our approach was able to effectively infer important and interesting temporal properties.

2. RELATED WORK

Our work is mainly inspired by Ernst's work on dynamically inferring program invariants [9, 14]. The distinction between their work and ours is that their work focuses on the value relationships among variables which are more relevant to dataflow, while 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] is a machine learning approach that discovers temporal specifications a program must satisfy when interacting with an application programming interface (API) or abstract data type (ADT). It extracts scenarios from execution traces based on a dependency analysis to make their approach tractable for realistic programs and then uses a probabilistic finite automaton (PFSA) learner to infer specifications from the scenarios. Our approach is distinguished from specification mining in several aspects: (1) Our techniques target general types of events whereas specification mining is limited to API and ADT events. Although these events are important and interesting, programs may have many interesting temporal properties not related to such events. (2) 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. (3) 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. (4) The temporal properties discovered by our approach are guaranteed to be consistent with the traces used to discover them. We plan to use a static checker to further verify them to gain more confidence. 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.

Whaley et al. developed two techniques, one static and the other dynamic, for inferring sequencing models of methods of a component [17]. They also 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 [5]. 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 the following aspects: (1) the temporal properties inferred by our method are more general, (2) our event traces include identification of the executing thread, whereas theirs does not; hence, we do not need to build a model to determine the threads; (3) the effectiveness of their thread discovering technique requires recurrence in a single event trace, whereas our technique does not have such a restriction; and (4) their approach only uses a single event trace, while ours is based on many event traces.

Dwyer et al. developed a set of temporal property patterns based on a case study of hundreds of real property specifications [8]. They integrated those patterns into their Bandera toolset [6] so that users can express a temporal property in the Bandera Specification Language [7]. 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 [13]. Two dynamic analysis algorithms to detect race conditions and deadlocks run first. If those analyses report any warnings, the Java PathFinder model checker [12] 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.

Our work is focused on automatically deriving properties to check. Another input to the model checker is a model of the system to be checked. Traditionally, the model is written manually in a specifically-designed language different from the one in which the system is implemented. Constructing the model requires programmers be very familiar with both the target system and the modeling language and limits the use of model checking in standard development processes. Much recent research has focused on automatically extracting models from code. For example, Bandera [6], Java PathFinder [12], and SLAM [3] can work directly on Java or C code. Such techniques greatly facilitate the adoption of model checking, and are complimentary to our work.

3. APPROACH

Our approach begins by instrumenting either the whole program or its key aspects (e.g. shared objects). Next, we execute a test suite on the instrumented program and collect execution traces. Then, we instantiate candidate temporal property patterns. We compute the satisfaction ratio table for each candidate pattern based on the execution traces. From these ratios, we infer the strongest pattern satisfied by each set of events. Finally, we subject the inferred properties and the program to validation either by manual inspection or by using a model checker. Subsections 3.1 to 3.6 discuss each step in more detail.

To illustrate our approach, we will use a Java implementation of a simplified Producer-Consumer problem (adapted from the pipeline example in Bandera's distribution) shown in Figure 1. It has two threads: Producer and Consumer. These two threads share one object buf of type Buffer. In this simplified version, Buffer can only hold one value. The add method inserts a new element into the buffer and the take method removes an element and returns it to the caller. The Producer iteratively inserts the integers from 1 to *n* (as designated by user input) into buf, while the Consumer takes those numbers from buf and prints them out. After adding the final value, the Producer thread calls the stop method which writes 0 to buf. When the Consumer reads a 0, it exits the run loop and terminates.

One desirable temporal property is that adding an element to buf and removing an element from buf alternate. This ensures that a

```
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 ();
  }
}

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

  public final synchronized void stop () {
    while (queue!=-1)
       try { wait (); } catch (InterruptedException ex) {}
    queue = 0; notifyAll ();
  }
}

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

Figure 1. Example code.

new element will not be overwritten by the Producer before being taken by the Consumer and that the Consumer cannot take an element from an empty buffer. Another important property is that once the Producer calls the stop method, the Consumer must eventually stop. The implementation shown in Figure 1 satisfies both properties.

3.1 Instrumentation

We instrument the target program at all method entry and exit points. An event consists of the current thread's identifier, a method name, and a location: either *E* for entering a method or *X* for exiting the method (multiple exit points can be identified by line numbers). For the program in Figure 1, we use *W* (writer) to represent the Producer thread and *T* (taker) the Consumer thread. When the Producer calls the add method, we collect the event *W_add_E*. Running the program with input 3 produces this trace (which we will refer to as "Event Trace 1"):

W_main_E, *T_run_E*, *W_add_E*, *W_add_X*, *T_take_E*, *T_take_X*, *W_add_E*, *W_add_X*, *W_stop_E*, *T_take_E*, *T_take_X*, *W_stop_X*, *T_take_E*, *W_main_X*, *T_take_X*, *T_run_X*.

3.2 Testing

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. For our example problem, the only input is a single integer so automatically generating tests is easy.

3.3 Instantiating 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 a valid string satisfying this pattern, but *SPPSSP* is not because no *S* responds to the last *P*. We can use a Quantified Regular Expression (QRE) [15] to describe the *Response* pattern: [-*P*]*; (*P*; [-*S*]*; *S*; [-*P*]*)*. QREs are similar to regular expressions: **;** is the concatenation operator (*P*;*S* represents *P* followed by *S*), [-] is the exclusion operator ([-*P,S*] specifies any event in the alphabet except *P* and *S*). The ***** (Kleene star) and () (grouping) operators have their normal meanings.

We obtain a *concrete property* by replacing each abstract event with a monitored event. For example, if we select *P = W_add_X* and *S = T_take_X*, we get the property: [-*W_add_X*]*; (*W_add_X*; [-*T_take_X*]*; *T_take_X*; [-*W_add_X*]*)*. We can get a set of concrete properties by replacing the abstract events with those monitored events of interest to us (e.g. those events of the buf object). If a pattern is parameterized by *m* abstract events and we monitor *n* events, there are n^m possible instance properties.

3.4 Computing Property Satisfaction

For each concrete property, we determine if an event trace matches it. Table 1 shows the results for two execution traces (the first is Event Trace 1 introduced in Section 3.1; the second is another plausible trace of the example program). Note that *W_main_X* (as *P*) and *T_take_E* (as *S*) satisfies the *Response* pattern for the first trace but not for the second one.

For each event trace, we first compute its satisfaction table as in Table 1. Then we calculate the *satisfaction ratio* for each property by averaging the values for all traces. The result is a table whose cells represent the percent of event traces satisfying the corresponding properties. Table 2 shows the satisfaction ratios of the two traces. For instance, the *Response* pattern for *W_main_X* (as *P*) and *T_take_E* (as *S*) is satisfied in trace 2 but not in trace 1. Hence, the satisfaction ratio of this property is .5 for the two traces.

S P	Event Trace 1		Event Trace 2	
	T_take_E	*T_take_X*	*T_take_E*	*T_take_X*
W_add_E	1	1	1	1
W_add_X	1	1	1	1
W_main_X	0	1	1	1

Table 1. Property satisfaction for two traces.

S P	*T_take_E*	*T_take_X*
W_add_E	1	1
W_add_X	1	1
W_main_X	0.5	1

Table 2. Property satisfaction ratios for Trace 1 and 2.

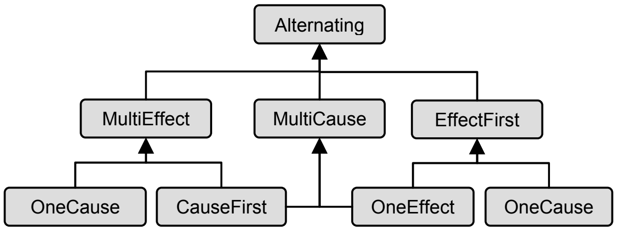

Figure 2. Partial order among patterns.

3.5 Inferring and Synthesizing Properties

We infer properties based on the satisfaction ratio table. We deem a property as true if its satisfaction ratio is greater than a pre-defined threshold. For now we simply set the threshold to 1 (meaning the property is true for all traces). All the properties shown in Table 2 are true except for the *W_main_X/T_take_E* property.

In future work, we intend to explore properties with thresholds less than 1. These could reveal bugs in the program, or properties that hold for only a subset of the input space.

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 3. 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. The following relationships among the above patterns hold:

(1) $L(Alternating) = L(CauseFirst) \cap L(OneCause) \cap L(OneEffect)$
(2) $L(MultiEffect) = L(CauseFirst) \cap L(OneCause)$
(3) $L(MultiCause) = L(CauseFirst) \cap L(OneEffect)$
(4) $L(EffectFirst) = L(OneCause) \cap L(OneEffect)$

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 above relationships to infer the strictest pattern. For example, for the events *W_stop_X* (as P) and *T_take_X* (as S) the Event Trace 1 satisfies *OneCause* and *OneEffect* but not *CauseFirst*. Using the above relationships, we find it satisfies *EffectFirst* but not *Alternating*, *MultiEffect*, or *MultiCause*. So, we conclude the strictest pattern between the two events this event trace satisfies is *EffectFirst* which gives us more information than *Response*. If we combine this with the knowledge that *W_stop_X* only appears once, we know that the execution ends with the Producer sending the stop signal followed by the Consumer exiting.

Another way to extend a pattern is to vary its scope [8]. Pattern scopes define the region where a pattern holds. The scope of all the above patterns is global since satisfying the pattern requires that the whole event trace matches it. We can modify patterns by adding scopes that limit when the pattern must hold. For example, the *before R* scope below specifies P's occurrence must be followed by the occurrence of S before R's occurrence:

[-R]* | [-P,R]*; (P; [-S,R]*; S; [-P,R]*)*; R; .*

In addition to the *before R* scope, [8] shows three other scopes of the *Response* pattern. All the patterns in Table 3 can be adjusted to the *before R* scope. For example, *CauseFirst* with *before R* scope is as follows.

$CauseFirst_{|R}$ = [-R]* | [-P,S,R]*; (P; [-S,R]*; S; [-P,R]*)$^{+}$; R; .*

(We use + (one or more instances) here to filter false results. For example, if R is an event always appears at the beginning of every trace, using * would result in a pattern that is satisfied by any two monitored events, which is not what we want.)

Fixing R, we can find the strictest pattern between two events P and S by following the same procedure we did for the global scope. If we replace R with all monitored events, we can find the strictest pattern before each event. The problem now is how to choose from them. First, since our goal is to find if two events can satisfy a stricter pattern under a different scope, we will only consider a pattern with a *before R* scope if we can find a stricter pattern by using the scope restriction. The later an R appears in the event traces, the larger the scope is. So, we always prefer a later R to make the scope as large as possible. Thus, our goal is to find the latest R before which the strictest pattern is still satisfied.

We sort all events according to their average relative positions. We calculate the average relative position of an event R as is a number between 0 and 1 defined as

$$\text{Average relative position of } R = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m_i}\frac{L_{i,j}}{m_i \cdot l_i}}{n}$$

Where $L_{i,j}$ is the position of the j^{th} occurrence of R in the i^{th} event trace (where positions just count the events), m_i is the total number of occurrences of R in the i^{th} event trace, l_i is the length of the i^{th} event trace, and n is the total number of event traces.

Name	QRE	Valid Examples	Invalid Examples
Response	[-P]*; (P; [-S]*; S; [-P]*)*	SPPSS	SPPSSP
Alternating	[-P,S]*; (P; [-P,S]*; S; [-P,S]*)*	PSPS	PSS, PPS, SPS
MultiEffect	[-P,S]*; (P; [-P,S]*; S; [-P]*)*	PSS	PPS, SPS
MultiCause	[-P,S]*; (P; [-S]*; S; [-P,S]*)*	PPS	PSS, SPS
EffectFirst	[-P]*; (P; [-P,S]*; S; [-P,S]*)*	SPS	PSS, PPS
CauseFirst	[-P,S]*; (P; [-S]*; S; [-P]*)*	PPSS	SPSS, SPPS
OneCause	[-P]*; (P; [-P,S]*; S; [-P]*)*	SPSS	PPSS, SPPS
OneEffect	[-P]*; (P; [-S]*; S; [-P,S]*)*	SPPS	PPSS, SPSS

Table 3. Temporal property patterns.

```
rank = Rank all events;
foreach combination of two events {
  compute their globally strictest pattern;
}
foreach combination of two events {
  rank2 = rank;
  while (pattern != Alternating) and (rank2 is not empty){
    R = remove the latest event from rank2;
    new_pattern = compute the strictest pattern before R;
    if (new_pattern is stricter than pattern) {
      pattern = new_pattern$_{|R}$;
    }
  }
}
```

Figure 3. The prototype algorithm

Suppose that we only have Event Trace 1: *T_take_X* appears three times at positions 6, 11, and 15 respectively. The length of that trace is 16. So the average relative position of *T_take_X* is 0.6667. If we rank those events in the order of their average relative positions, we get:

(*W_main_E*, *T_run_E*, *W_add_E*, *W_add_X*, *T_take_E*, *T_take_X*, *W_stop_E*, *W_stop_X*, *W_main_X*, *T_run_X*).

Figure 3 shows the pseudo-code of our algorithm. To compute the satisfaction ratio table for the *CauseFirst*$_{|R}$, *OneCause*$_{|R}$, and *OneEffect*$_{|R}$ patterns, start from the latest event *R*, and compute the strictest pattern with *before R* scope. If the new pattern is stricter than the one computed for the larger scope, select the new pattern. Continue for the preceding event until all events have been tried or the *Alternating* pattern (strictest possible pattern) is reached. We keep the result the same as the global pattern if we cannot find a stricter pattern using any event as *R*. In this way, we find the strictest pattern *W_add_X* (as *P*) and *T_take_X* (as *S*) can satisfy is the *Alternating* pattern *before W_main_X* (as *R*), whereas the strictest pattern these two events satisfy in the global scope is *MultiEffect*. Both results are informative and useful. The first one reveals the alternating relationship between the two events before the *Producer* stops. The latter one indicates that there are multiple *T_take_X*'s responding to at least one *W_add_X*.

In our preliminary experiments, we have only investigated the *Response* pattern and *before R* scope. We plan to explore the usage of other patterns like *Precedence* and other scopes. We also need ways to prioritize properties. Currently, we show those properties relevant to the shared objects and between two events from different threads first, and give higher priority to the stricter properties.

3.6 Validation

To gain more confidence in the correctness of inferred properties, we subject them to validation manually by programmers. We plan to automate this by a model checker in future. The benefits of validation are twofold: first, disproving any desirable property could lead us detecting faults in a program; second, disproving any undesirable property could not only give us more confidence but also reveal the inadequacy of the test suite. In our experiments to date, we have only inspected properties by hand. For larger programs and more complex properties, it will be essential to use a model checker to automate the validation process.

4. RESULTS

We built a prototype implementation and evaluated our approach on the Java program shown in Figure 1. Our prototype implementation is 630 lines of Perl code, and automates all the steps in our approach except for instrumentation and validation which are done manually.

We executed an instrumented version of that program with 100 randomly generated inputs within the range from 1 to 10000. Table 4 shows the strictest pattern with *before R* scope. If a cell does not have "|*R*", it means this pattern is the strictest within global scope and cannot be improved by changing to any *before R* scope. The rank of events according to their average relative positions is same as the ordering in Section 3.5. An entry of **0** means that the corresponding two events satisfy only the *Response* pattern or do not satisfy any pattern listed in Table 3.

We found several interesting properties. Globally *W_stop_X* (as *P*) and *T_run_X* (as *S*) have an alternating pattern. Considering the fact that each event occurred only once, we concluded that after the Producer thread sent out the stop signal and the Consumer thread eventually stopped. This corresponds to one of the two properties we expect the program to have.

Globally every *W_add_X* has at least one response event *T_take_X*, and *W_add_X* starts the chain (indicated by the pattern *MultiEffect*). This results from the Consumer calling the take method in response to the Producer calling the stop method. *Before W_main_X* (as *R*), these two events have a one-to-one correspondence. This corresponds to another property we expected.

The second experiment we did is applying our approach to a faulty program. We removed the synchronization lines (the while statements in add and stop in Figure 1) in the add and stop methods of the Buffer class. In the new program, it is possible for the Producer thread to add multiple times before the Consumer thread reads the previously inserted values, thus losing some values.

We instrumented the new program and executed it with another 100 randomly generated inputs within the range from 1 to 10000. Table 5 shows the strictest patterns with *before R* scope. We

S *P*	*T_take_E*	*T_take_X*	*T_run_E*	*T_run_X*
W_main_E	Alternating$_{\|R1}$	Alternating$_{\|R1}$	Alternating	Alternating
W_main_X	0	EffectFirst	0	Alternating
W_add_E	CauseFirst	CauseFirst	0	MultiCause
W_add_X	Alternating$_{\|R1}$	Alternating$_{\|R2}$	0	MultiCause
W_stop_E	EffectFirst	EffectFirst	0	Alternating
W_stop_X	EffectFirst	EffectFirst	0	Alternating

Table 4. The strictest pattern with *before R* scope.
R$_1$ = *T_take_X*, R$_2$ = *W_main_X*

S *P*	*T_take_E*	*T_take_X*	*T_run_E*	*T_run_X*
W_main_E	Alternating$_{\|R1}$	Alternating$_{\|R1}$	Alternating	Alternating
W_main_X	**0**	EffectFirst	**0**	Alternating
W_add_E	**0**	CauseFirst	**0**	MultiCause
W_add_X	**0**	CauseFirst	**0**	MultiCause
W_stop_E	**0**	EffectFirst	**0**	Alternating
W_stop_X	**0**	EffectFirst	**0**	Alternating

Table 5. The strictest pattern with *before R* scope for faulty program. R$_1$ = *T_take_X*.

found that the strictest property that holds for *W_add_X* (as *P*) and *T_take_X* (as *S*) is only that the first *W_add_X* always appeared before any *T_take_X*. This clearly indicates the lack of an alternating pattern resulted from the inadequate synchronization. We found that the faulty program still has one of the desirable properties: as long as the Producer called the stop method, the Consumer eventually stopped itself.

5. CONCLUSION

One of the biggest challenges in effective adopting of model checking is determining useful properties to check. We have described an approach for automatically inferring interesting temporal properties of programs by analyzing execution traces. We built a prototype implementation, and demonstrated its effectiveness on a simple program. The results are promising, but a number of challenges remain before our technique can be applied to realistic programs including extracting interesting properties with a large number of possible events, developing good strategies for testing that produce useful event traces for longer executions, and handling more complex thread interactions.

ACKNOWLEDGMENTS

This work has been funded in part by the National Science Foundation through NSF CAREER (CCR-0092945) and NSF ITR (EIA-0205327) grants.

REFERENCES

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, January 2002

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

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

[4] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *15th IEEE International Conference on Automated Software Engineering*, September 2000.

[5] J. E. Cook, Z. Du, C. Liu, and A. L. Wolf. Discovering models of behavior for concurrent systems. *New Mexico State University Technical Report*, NMSU-CSTR-2002-010.

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

[7] 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.

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

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *IEEE Transactions on Software Engineering*, February 2001.

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

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

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

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

[14] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: an empirical evaluation. In *ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering*, November 2002.

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

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

[17] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium on Software Testing and Analysis*, July 2002.