Automatically Discovering Temporal Properties for Program Verification

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

Abstract

This paper reports on our experience using a dynamic analysis tool, Terracotta, to automatically infer temporal properties, and a model checker, Java PathFinder, to check the inferred properties. To our best knowledge, this is the first experiment using a model checker to check automatically inferred properties. We introduce two key ideas to make our approach effective. First, we develop techniques for handling context information in a program's execution traces so that some interesting properties can be discovered. Second, we infer properties in a single threaded environment and to check them in a multi-threaded one. We evaluate our approach on a prototype implementation of a UNIX-like file system, Daisy. Our techniques revealed a bug in Daisy and identified several interesting and subtle temporal behaviors.

1 Introduction

A temporal property constrains the occurring order of program states [25]. A program must satisfy certain temporal properties to be correct. Checking whether a program has a temporal property requires a specification of the property in first place. Most systems, however, are developed without such a specification. Manually specifying even simple temporal properties is difficult, time consuming, and error-prone [22]. This prevents a systematic exploration of all properties of interest and impedes wide adoption of tools for checking such properties.

To tackle this problem, we proposed a lightweight dynamic analysis technique for automatically discovering those desirable temporal properties from several executions of a target program [31]. We assume the target program is almost correct (i.e., we can have at least one correct execution of the program), but might still contain some transient bugs that violate certain desirable properties. Our approach works by observing the successful executions of the program and computing an abstraction of its temporal behaviors based on some pre-defined parameterized templates. The key observation that motivates us to develop this approach is that a transient bug usually lies on cold paths of the program that are not activated frequently. Hence, it is reasonable to assume that common executions such as successful test runs represent the desirable behavior of a program. Having inferred a temporal specification, we can check more extensively whether the target program satisfies it by leveraging some powerful verification tool. For example, a model checker is able to exhaustively explore all scenarios of thread interleaving, which is a very useful way for detecting those subtle bugs that are rarely detected by testing [9].

This paper explores the hypothesis that it is feasible and effective to obtain a temporal specification for a program so that checking this specification can help detect bugs in the

target program. To evaluate our approach, we developed a dynamic inference tool, called Terracotta, which takes a program's execution traces and infers a set of temporal properties. Section 2 describes our inference approach and implementation. We then applied it to Daisy, an implementation of a file system, and used Java PathFinder, a model checker for Java programs, to check the inferred properties [27]. This led us to detect a bug in Daisy and several subtle and interesting temporal behaviors. Section 3 provides details about our experiments and results. We discuss what lessons we have learned through the experiments and what future research needs to be done to make this approach more effective in Section 4. Section 5 summarizes related work and the paper concludes in Section 6.

We make the following contributions in this paper:

- To our best knowledge, this is the first experiment on using model checking to validate inferred temporal properties.
- We propose and evaluate an approach that infers properties by observing a program running in a single threaded environment and checking those properties in a multi-threaded environment.
- We introduce two types of contextual information (objects and threads) to improve property inference and systematically evaluate their impact on inferred properties.

2 Approach

Figure 1 gives an overview of our approach. First, we instrument a target program to monitor those events and states of interest. Then we execute the instrumented program against test cases to produce execution traces. Next our inference engine processes the traces offline. It starts by instantiating a set of pre-defined parameterized property templates based on the monitored events and states. Then it matches the execution traces against the instantiated properties and outputs properties satisfied by the execution traces as inferred properties.

The inferred temporal properties are a *temporal operational abstraction* of the target program (we borrow the term operational abstraction from [18]). One straightforward application of a temporal operational abstraction is to assist program evolution as shown in Figure 2. Suppose we have two versions of a program, both of which are expected to implement the same specification. In particular, one version is newer than the other. We can apply our approach to both versions by first running them against a same set of test cases and inferring a temporal operational abstraction for each. Comparing the two temporal operational abstractions can help us identify important properties that have been preserved and changed. In our previous

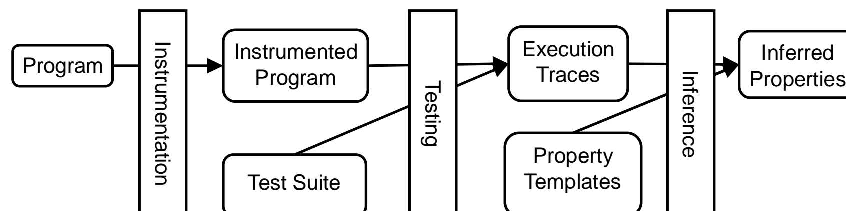

Figure 1. An overview of our approach

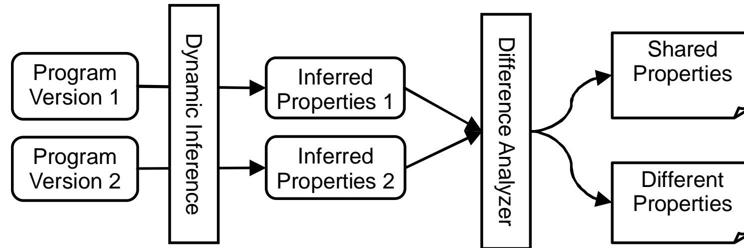

Figure 2. Use inferred properties in program evolution

work, we applied this approach to six versions of OpenSSL with encouraging results [2, 32].

This paper focuses on another use of a program's temporal operational abstraction: using it to verify properties of the target program. Figure 3 shows how this works. First our dynamic analysis infers a set of likely temporal properties for the target program. Then we employ a verification tool (in this case a model checker) to check whether the program satisfies those inferred properties, usually in a more exhaustive fashion (e.g., covering more thread interleaving scenarios or input sub-domains [29]).

If the verification tool does not find any violations of an inferred property, we have increased confidence that the property is satisfied by all possible executions of the program. If the verification tool does find a possible execution in which the inferred property does not hold, however, we may learn something useful about the program or our testing strategy.

There are two situations in which a verification tool can detect a violation. First, if the verifier tries to check the program on new input sub-domains, it is possible that an inferred property is violated since we do not observe those sub-domains in the test executions. Such cases represent either an inadequacy of our test execution or a bug in the program handling some unusual input. We could use this information to improve our test suite. In the second case, if the verifier tries to execute all possible scenarios of non-deterministic actions (such as thread scheduling), it is possible that an inferred property is violated under some very rare scenario. This usually indicates a bug in the target program whose behaviors should not depend on those non-deterministic actions. The verifier typically produces a counter-example demonstrating an execution path that leads to the violation. This information makes it easy to either add new test cases or debug the program.

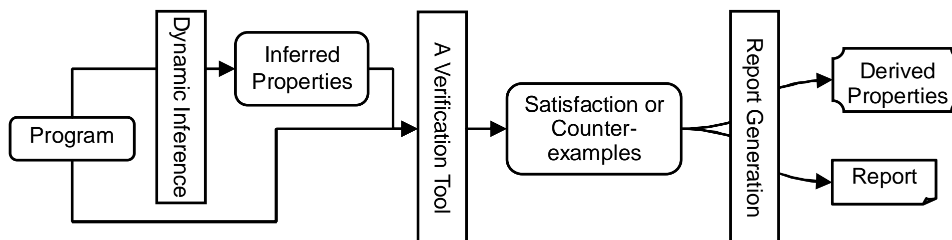

Figure 3. Use inferred properties in program verification

This approach can be largely automated. The main requirement is converting the inferred properties into a checkable form of a chosen verification tool, which can be automated. A significant impediment to using verification tools in practice is the effort and expertise required to develop properties worth checking. Identifying and manually expressing even simple temporal properties is time-consuming and error-prone [22]. Our approach infers properties to check automatically based on test executions. On the other hand, verification tools require significant resources to run, so it is not feasible to run a powerful verification tool on all possible properties directly. By using our inference approach, we are able to efficiently identify a relatively small number of interesting properties to verify.

Next we discuss each step in our dynamic analysis approach. We have implemented our dynamic analysis in a prototype tool called Terracotta, which are 8000 lines of Java code. It takes a program's execution traces and outputs a temporal operational abstraction for the program as a set of properties.

2.1 Instrumentation

Temporal properties provide constraints on the occurrences of program states and events, so our analysis depends on identifying which program states and events to monitor. A naïve approach would be to monitor all possible states and events; this simply does not scale to large systems. As a result, we need to selectively monitor only those events likely to be of interest. In our experience, selecting appropriate events to monitor can be easily automated. For example, we can choose to monitor the entrance and exit of all public methods in a class. Another question that is closely related to this one is what context information we should record. For example, when monitoring method invocations, in addition to the method signature, we can also record thread information, real parameters, and return values. As we will show later, such information is critical for inferring certain interesting properties.

We use JRat as our instrumentor for Java programs [1]. JRat is able to instrument Java bytecodes at any method entrance and exit point and the entrance of any exception handler. It allows users to develop their own event handler, which we implemented to record an event. JRat accepts specific policy files that can select which methods to be monitored based on name patterns of methods or classes.

2.2 Testing

Properties are inferred based on the test executions, so the value of the inferred properties will depend on the test cases we run. Many target system come with a set of regression test cases. Then we can simply select those test cases, which we assume represent desirable behaviors of the target system. There are, however, systems that might not come with a test suite or whose existing test suite is not suitable for our purpose. For example, if we are interested in inferring the usage rules of a library, we would like to have a client program that uses the interface of the library. So, the unit test cases of the library will not serve our purpose.

The problem underlying such scenarios is how to create a test harness to model the environment. In our experience, we have found that random testing is a very effective way to produce test cases or model the environment. For the library example, we can create a test

harness that tries to execute one or more operations up to certain number of times. Each time, it randomly selects the operations to be executed from the library's interface. We use a similar approach for testing Daisy. Section 3.3 presents results from our experiments on Daisy illustrating how different test harness configurations influence the inference results.

2.3 Property Templates

Properties are inferred by instantiating and checking property templates on the test executions. To extract the grammar that produces a set of strings is historically called the *grammar inference* problem, which has been proved to be NP hard [16, 17]. This limits the scalability of the earlier approaches that try to extract a complete state machine representation [3, 4, 5, 10, 30].

Dwyer et al. demonstrated that most of the real properties of interest involve only a few events [13]. This motivates us to focus on extracting properties only with a few events. In an earlier work, we proposed seven templates based on the Response pattern (we will use pattern and template interchangeably) [13, 31]. The Response pattern constrains the order of occurrence of a causing event, P, and an effect event, S, so that P's occurrence must be followed by S's occurrence. The seven templates shown in Table 1 simply add constraints on whether P or S can occur more than once and whether S can occur before P occurs. We use regular expressions to describe those templates. For example, the Response pattern has the form [-P]*(P[-S]*S[-P]*)*, which can be simplified to S*(PP*SS*)* if we filter all events other than P and S from the traces.

One important characteristic of this hierarchy of templates is that they form a partial order in terms of their strictness. Let A and B be two patterns. We say that A is stricter than B if the set of strings that A accepts is a strict subset of the set of strings that B accepts. Figure 4 shows their partial order relationship. In particular, OneCause, CauseFirst, and OneEffect are the three weakest primitive templates (but each is still stricter than Response). The Alternating pattern is the strictest pattern. We use the notation P→S to denote the Alternating pattern between P and S. In the middle, MultiEffect, MultiCause, and EffectFirst are three templates that are less strict than Alternating but stricter than any of the three primitive templates. A pattern is neither stricter nor less strict than another pattern on the same level.

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*

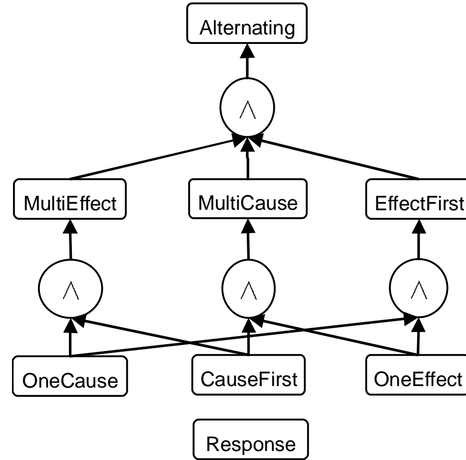

Figure 4. Partial order of properties

Another important feature of the patterns is that they have logical relationships (represented as "^" in Figure 4). For example, two events satisfy MultiEffect pattern if and only if they satisfy both OneCause and CauseFirst patterns. Our goal is to find the strictest pattern two events can satisfy. We can take advantage of the second characteristic of the patterns to solve this problem. First, we test which of the three primitive templates a pair of events satisfies. Then we can deduce their strictest pattern using their logic relationship. For example, if two events satisfy OneCause and OneEffect, but not CauseFirst, then their strictest pattern is EffectFirst.

In this paper, we add two sets of new patterns by allowing alternate events. One is the alternate-effect-event pattern, P→S|Q, where the effect event can be either S or Q. The other is the alternate-cause-event pattern, P|Q→S, where the causing event can be either P or Q. The relationship between the causing events and the effect events can be any of the seven patterns mentioned earlier. We include them to infer such properties we have seen in real programs. For example, a file, after being successfully opened, can be either read or written (i.e., Multieffect (open, read|write)), and finally be closed (i.e., Multicause (read|write, close)).

Checking whether a trace satisfies a particular property can be done efficiently using a finite state machine. We instantiate all possible 2-event primitive properties and then determine the strictest property satisfied by each pair of events. Suppose that there are n distinct events, and the length of all traces is L. In our earlier work, we developed a straightforward algorithm that has $O(n^2L)$ time complexity and $O(n^2)$ space complexity [31]. By checking all properties during a single trace traversal, our current prototype has time complexity $O(nL)$ and space complexity $O(n^2)$. It is very efficient and is able to infer properties on traces with 10 million entries and 3000 distinct events in 10 hours. In this case, Daisy is a small program with few events, so none of the inference executions takes more than a few minutes. For the alternate-event properties, instead of needing to try all n^3 possible properties, we can use event counts to only try those properties that could be satisfied. That is, in order for the P→S|Q alternating property to be satisfied the number of P events must equal the sum of the number of S and Q events.

2.4 Techniques for Handling Context Information

A monitored event has two types of information: static information (e.g., the method entered) and context information (e.g., the runtime thread, this object, the real parameters passed to a method, and return values). The advantage of dynamic analysis over static analysis is the availability of precise context information. We can take advantage of this information to infer more useful properties.

In general there are two alternatives: context-neutral and context-sensitive. In context-neutral mode, we treat two events that have same static information but different context information as the same event, whereas context-sensitive mode considers them as two distinct events. For example, consider an example trace in Figure 5(a). There are only two events, A and B, in the context-neutral mode, but four events if we include the tread identity in the event context (indicated by the number in brackets after the event): A[1], A[2], B[1], and B[2].

Context-neutral analysis finds that the strictest pattern satisfied by A and B is CauseFirst as shown in Figure 5(c). Context-sensitive analysis finds the six Alternating properties shown in Figure 5(d), only the third and fourth of which are likely to be of interest. Neither context-sensitive nor context-neutral analysis, however, finds the alternating property between A and B events in the same thread. Examining the results in Figure 5(d), we can see that if we generalize the properties based on same thread identity, we will be able to infer the expected property. This is equivalent to slicing the original trace into separate traces based on thread identity. Figure 5(b) shows the two traces obtained by thread slicing. Our basic analysis will infer the A→B property from the two new traces.

Besides threads, we can also treat other types of context information in a similar way. For example, we can slice the trace based on the *this* object so that all entries in the trace relevant to the same object will be put into one trace. Similarly, we can also slice the trace based on the value of the ith argument to a method.

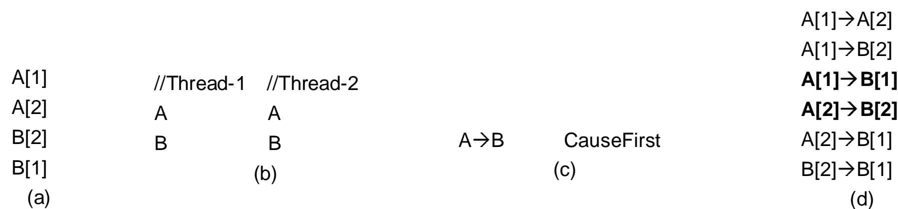

Figure 5. Different context handling techniques

(a) The original trace. (b) The two traces after slicing the original trace on thread identity. (c) Context-neutral mode found A and B satisfy the CauseFirst pattern on the original trace. (d) Context-sensitive mode found six Alternating properties. The two in boldface are most likely to be interesting.

The results of context-sensitive analyses are the most complete, but do not tend to be useful without generalization. Context-slicing can be viewed as a simple way to generalize the

results of context-sensitive analyses. Although it works well in this example, it is still an open question whether more complex techniques are necessary. An apparent problem is that properties that cross contexts are not inferred using context-slicing, such as an alternating pattern between an A event in one thread and a B event in a different thread. This means we will lose some properties, which might be discovered by context-neutral or some other generalization methods. We plan to study this in future work. For now, we analyze a trace using both context-neutral and context-slicing and union their results together. Although the above example seems to suggest that context-slicing is preferable than context-neutral, it is worth noticing that for some traces, context-neutral analyses would detect some properties that would be lost using context-slicing. We present our experimental results on evaluating different context handling methods in Section 3.3.

3 Inference Experiments

To evaluate our approach, we applied Terracotta to Daisy, a prototype implementation of a Unix-like file system. Daisy was designed as a layered hierarchy. Each layer provides an abstraction of lower level information, which is used by the layer immediately on top of it. Daisy is described in more detail in another article in this issue.

We instrument Daisy at all method entrance and exit points using JRat. At each entrance point, we record the method's signature, current thread, the hashcode of *this* object if applicable, and the hashcode (for objects) or value (for primitive variables) of all parameters. We also created a wrapper of the RandomAccessFile class so that we could collect traces of calls to its API.

For our testing, we adapted the test harness that comes with the Daisy distribution. Our test harness, DaisyTest, takes three parameters: F, the number of files to be created initially, T, the number of threads to be created, and N, the number of iterations each thread executes. The main thread of the test harness first creates F files and T threads. Each child thread executes up to N iterations. In each iteration, it randomly selects an operation from the APIs of DaisyDir (one of read, write, set_attr, or get_attr) to execute with arguments randomly selected within the valid range.

In this section, we show the results of experiments using different inference modes and different testing configurations. We focus on Alternating properties of entrance events of methods. So, Mutex.acq() represents the entrance event of the acq method of the Mutex class unless otherwise indicated. Unless there is ambiguity, we also omitted the parameters of a method in our discussion. Section 3.1 reports on how different inference modes impact the set of inferred properties.

We ran DaisyTest with a number of configurations. In each configuration, we fixed two of its parameters and varied the other one with several values. For each set of parameters, the test execution produces one trace. We will discuss how different configurations influence the inferred properties in Section 3.2.

3.1 Inference Modes

As discussed in Section 2, Terracotta supports several execution modes based on the use of

context information. Here, we consider how object and thread context information changes the inferred properties.

Table 2 shows the results of varying the use of object context information. We slice the original trace based on the recorded object identities including this object and all arguments. We found one new alternating property by slicing on *this* object: Mutex.acq()→Mutex.rel(). This indicates the acq and rel methods should alternate with each other when invoked on a same Mutex object. Because the point of a mutex is to provide exclusive access to a resource, the inferred property appears desirable. However, when we attempt to verify this property, we find a subtle counterexample as described in Section 4. The property was not inferred without object context information, since it is possible for an execution to call Mutex.acq on two different mutex objects before the first Mutex.rel call. We found another new property by slicing on the first argument: LockManager.acq(lockno)→LockManager.rel(lockno), where lockno is a long integer representing the sequence number of a lock. This property also seems desirable, which means the same lock is acquired and released in an alternating order. We did not find any new properties by slicing on other arguments.

Slicing can also miss some properties that involve more than one type of object, which can be detected in object-neutral mode. For example, eight out of the 56 properties were missing after slicing on *this* object. One of them is Petal.write→RandomAccessFile.writeByte. The first event is a static method whereas the second one is invoked from the RAF object member of the Petal class. As a result, slicing put all entries of the first event in a different trace from all entries of the second event.

We also observe that the number of traces first increases and then decreases as we move from the 0th argument (i.e. *this* object) to the 4th argument. This is because the number of traces is proportional to the number of different objects on which a particular slicing is based. For example, since most methods in Daisy have at least one argument, the number of traces sliced based on it is the most. Similarly, we only had two traces after slicing on the fourth argument because Daisy only has six methods with four arguments. There are only 139 sliced traces

Table 2. Slicing different objects

Sliced Object	Number of Traces	Alternating Properties	New Alternating Properties
No Object	1	56	0
this	139	49	Mutex.acq()→Mutex.rel()
1st arg	1749	15	LockManager.acq(lockno)→ LockManager.rel(lockno)
2nd arg	721	25	0
3rd arg	19	49	0
4th arg	2	56	0

The trace is from running DaisyTest with F=2, T=1, and N=10. We ran Terracotta in its thread-neutral mode and by limiting the frequency of events to be at least 3 (see Section 3.2). The “New Alternating Properties” column shows the new properties we obtained compared to the previous row. No method used in Daisy has more than four arguments.

<table>
<tr><th rowspan="3">Number of Threads</th><th colspan="6">Number of Alternating Properties</th></tr>
<tr><th colspan="3">No Object</th><th colspan="3">Slicing this</th></tr>
<tr><th>Neutral</th><th>Sensitive</th><th>Slicing</th><th>Neutral</th><th>Sensitive</th><th>Slicing</th></tr>
<tr><td>1</td><td>56</td><td>109</td><td rowspan="5">47</td><td>49</td><td>96</td><td rowspan="5">40</td></tr>
<tr><td>2</td><td>38</td><td>188</td><td>31</td><td>171</td></tr>
<tr><td>3</td><td>37</td><td>237</td><td>30</td><td>216</td></tr>
<tr><td>5</td><td>33</td><td>408</td><td>27</td><td>379</td></tr>
<tr><td>10</td><td>33</td><td>728</td><td>27</td><td>679</td></tr>
</table>

Table 3. Varying the number of threads

We ran Daisy with F = 2, T = 1, 2, 3, 5, or 10 and N = 10. From each trace, we derived two sets of traces: one without any object information and the other sliced on *this* object. Then we ran Terracotta in thread-neutral, thread-slicing, and thread-sensitive modes. DaisyTest's main thread created T Tester threads but did not overlap with them temporally.

based on *this* object, which are much less than one would have expected. This is because most of the methods in Daisy are static methods which have no *this* object. The only two types of objects that have interesting properties are Mutex and RandomAccessFile, while the latter only has one instance encapsulated in the Petal class. The number of alternating properties follows an opposite trend because slicing prevents us from detecting properties involving two different types of objects as explained above. So the more sliced traces there are, the fewer alternating properties we infer.

Terracotta supports three ways to handle thread information in a trace: thread-neutral, thread-sensitive, and thread slicing. Table 3 presents the results of increasing the number of concurrently running threads and how different thread handling techniques impact the inferred properties. With the thread-sensitive analysis, the same events occurring in different threads are considered different events for the inference algorithm, so more alternating properties are inferred.

Figure 6 shows the number of alternating properties as the number of threads increases. In thread-sensitive mode, the number of properties inferred increases linearly with the number of

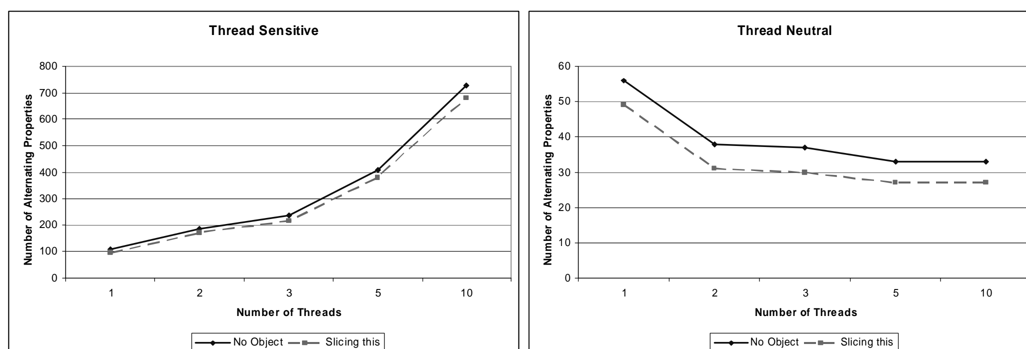

Figure 6. Properties inferred in thread-sensitive and thread-neutral modes

threads since invoking the same method counts as a different event for each thread. The number of distinct events increases as more threads are created due to lack of generalization as explained in Section 2, which also results in many more properties than thread-neutral

mode even when the number of threads is same. Thread-neutral mode decreases the number of properties. This is because more and more thread interleaving scenarios occur in the trace, which cause some properties that would otherwise be inferred to be violated. For example, two threads are enough to violate the Mutex.acq→Mutex.rel property, which, we expected should be true but actually can be violated in multi-threaded environment. This is consistent with our result of using JPF to check it.

Similarly, Daisy.iget(inodenum)→DaisyLock.acqi(inodenum) also disappears when we start three threads. Finally, when the number of threads was a big enough number (i.e. greater than 5), the number of alternating properties remained same. This is because those remaining properties involve events only occurring in the startup code of the test harness, which is a single thread.

Note that thread slicing infers the same set of properties no matter how many threads there were. It was still able to detect many interesting properties (Mutex.acq→Mutex.rel, DaisyLock.acqi→DaisyLock.reli, and DaisyLock.acqb→DaisyLock.relb) that thread-neutral did not infer when there were a large number of threads. This is because thread-slicing essentially mimics a single-threaded environment by putting all entries related to a thread to a separate sub-trace. In our Daisy experiment, applying thread-slicing to a trace containing multiple threads is effectively the same as running DaisyTest in a single thread with several different random number generator seeds or a large number of iterations, as discussed in the next subsection.

A key insight we drew from these experiments is that many interesting properties that would have been exposed in a single thread environment disappear in a multi-threaded one. As a result, we should either run the test harness with a single thread or use thread slicing if the trace has multiple threads. The other advantage of inferring properties from single threaded executions is there is less likelihood that important and necessary properties would be missed because they are not satisfied by the test executions but do not produce obviously flawed results when running on the test cases.

3.2 Test Parameters

In this section we consider how different configurations of the test harness affect the inferred properties. We ran Terracotta either with slicing on *this* object or without object information, but always enabled the thread-neutral mode. Since we already discussed the number of threads, we will focus on the other parameters of the test harness.

Table 5 shows what impact creating different numbers of files (F) can have on the trace, inferred properties, and running time of Terracotta. As the number of initially created files increases, the length of the trace increases accordingly. The number of sliced traces also increases because more Mutex objects appeared in the trace. Terracotta scales well to longer traces as its running time increases linearly to the length of the trace.

There is a big difference between one file and two files. This is because many events produced during creating new files only occurred once if there is only one file. This infrequency allows some uninteresting properties to be inferred. The number of alternating

Number of Files	Trace Length	Number of Traces	Number of Alternating Properties		Terracotta Running time (seconds)	
			No Object	Slicing *this*	No Object	Slicing *this*
1	9966	135	87	75	8	7
2	19188	139	56	49	13	13
3	30562	144	51	44	20	19
5	58980	157	51	44	35	34
10	163284	207	51	44	94	91

Table 5. Varying the number of files

We ran Daisy with F = 1, 2, 3, 5, or 10, T = 1, and N = 10. We ran Terracotta in thread-neutral mode with and without object slicing on *this* object.

properties drops and stabilizes after the number of files is at least three. However, the number of properties never decreased to 47 (without object slicing) or 40 (with object slicing), which is the stable point we observed in other experiments. This implies the number of files does not matter much as long as more than one file is created.

The final test parameters we consider are the event occurrence threshold and the effect of the random number seed used in generating the test cases. The event occurrence threshold can be used to filter out properties involving infrequent events since they are more likely to just be artifacts of the test executions.

Figure 5 shows how the number of alternating properties varied as we increased the frequency threshold. Slicing on *this* object resulted in 139 new traces. For two-event patterns, we ran Terracotta on both the trace without any object information and the sliced traces, while increasing the threshold from 1 to 12. When the threshold is 1, we analyzed all events and obtained 56 alternating properties for the trace without any object information and 49 for the sliced traces. The number of alternating properties keeps dropping when the threshold is increased, but it quickly becomes stable when the threshold is set to 4 or above. We also ran Terracotta with three-event patterns on the trace with no object information. When the threshold is 1, there are 49 alternate-effect-event alternating properties and 56 alternate-cause-event ones. But as long as we increased the threshold to 2, the number of two-effect-event Alternating properties decreased to and remained 1. The remaining property, RAF.seek→RAF.readByte|RAF.writeByte, is very interesting and desirable. Later we used Java PathFinder to find a violation to this property which revealed a bug in Daisy.

The number of alternate-cause-event properties kept dropping and became zero when the threshold was six. One alternate-cause-event property is DaisyDir.write|DaisyDir.writeLong→ Daisy.write(inodenum). In Daisy, Daisy.write(inodenum) is called once at two places: either at DaisyDir.write or DaisyDir.writeLong. That is why we would infer this property. We are investigating combining our dynamic inference with static analysis techniques to eliminate obviously satisfied (and uninteresting) properties.

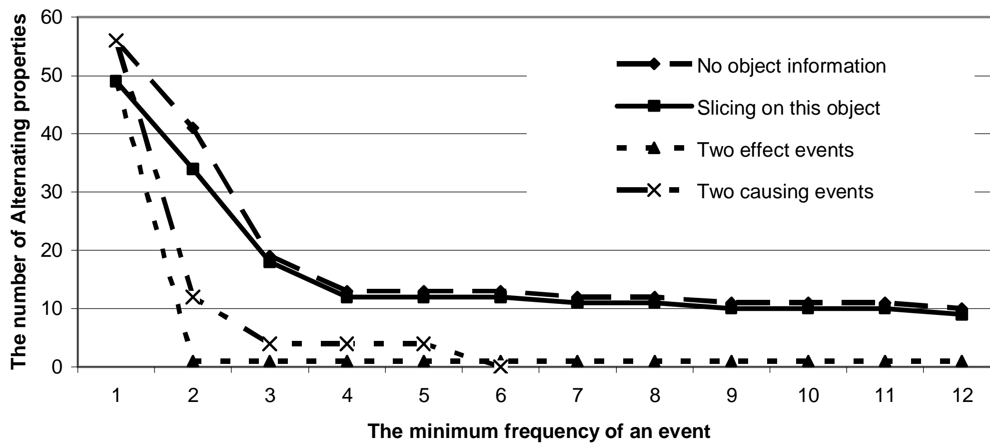

Figure 5. Impact of occurrence threshold

Dashed line represents the trace without any object context information. Solid line represents the trace being sliced based on this object. Without context information, dotted line shows the results of two-effect-event pattern and dash-dotted line shows results of two-causing-event pattern. We obtained the original trace by running DaisyTest with F = 2, T = 1 and N = 10. Its length is 19188. Before slicing on *this* object, there is only one trace. Slicing on *this* object increases the number of traces to 139.

We also wanted to understand the stability of the inferred properties to nondeterminism in the text executions. Table 4 summarizes the results of five different random number generator seeds and four different numbers of iterations (N). When we increased N to greater or equal to 15, the number of alternating properties does node depending on the seed used. This implies the choice of the seed does not matter too much when there are enough iterations. If we have

Object slicing	Random Number Generator Seed	Number of Alternating Properties			
		N=10	N=15	N=30	N=50
No	1	56	47	47	47
	2	47			
	3	47			
	4	51			
	5	47			
this	1	49	40	40	40
	2	40			
	3	40			
	4	43			
	5	40			

Table 4. Varying the random number generator seed and the number of iterations

We ran Daisy with F = 2, T = 1, N = 10, 15, 30, 50 and five different random seeds. We ran Terracotta in thread-neutral mode both with and without object slicing on *this* object. Before slicing on *this* object, there was only one trace whose length ranged from 19000 to 27000. Slicing on *this* object increases the number of traces to 139.

enough iterations (in this case 15 is sufficient), then the same alternating properties are found regardless of what random number generator seed is chosen.

In summary, we found that slicing, on either object or thread, is a very useful technique for inferring some interesting properties that would not have been inferred otherwise. A thread-neutral analysis is useful for generalizing properties and detecting properties involving multiple threads.

4 Checking Inferred Properties

In this section, we discuss how we used Java PathFinder (Version 3.1.1) to check the inferred alternating properties [27]. We checked the 22 alternating properties obtained using our inference techniques shown in Table 6. We used the trace from running DaisyTest on F = 2, T = 1 and N = 10. We removed all object information from the trace and inferred properties 4 through 22 using 3 as the event threshold. We inferred the 19th property after slicing on this object, the 18th after slicing on the first argument, and the 22nd through the two-effect-event pattern. JPF found counterexamples for 10 properties. We will explain next how we used JPF to check these properties first before giving details about the 10 violations.

To check an inferred property, we need to express the property in a way that can be interpreted by JPF. We implemented a property template as a parameterized finite state machine. For example, Figure 6 shows the FSM of an alternating pattern. On the left side is the Java class implementing the FSM, whereas the right side shows its transition function. The static 4 by 2 byte array, *rule*, hard-codes the transition function. The row index represents the current state: 0 is the start state, 1 is the state after encountering a P event, 2 is the state after encountering an S event, and 3 is the error state. States 0 and 2 are the two accepting states. The column index represents the current event: 0 means an S event, and 1 means a P event.

The state of an alternating FSM can be changed by calling its update method with the corresponding event as a parameter. We instrumented the code of Daisy to call the update method whenever a relevant event occurs. We expressed the safety property to be checked by JPF as an assertion on the current state, which says the current state should never be in an error state. We inserted a call to checkExitState method to ensure that the current state is in an accepting state before the program terminates (i.e. there is no such case that a causing effect occurs without an effect event following it). To check properties inferred through object slicing, we have to differentiate same method events of different objects. Our solution is to create a hash table that maps an object to its FSM. Besides the event, the update method also takes *this* object as a parameter so it can find the correct FSM for an object-sliced property. Similarly, the checkExitState method checks that all FSMs are in an accepting state when the program terminates. Currently, we manually inserted calls to the update method, but this could be readily automated.

```
public class Alt {
      private final static byte[][] rule =
            { { 3, 1 }, { 2, 3 }, { 3, 1 }, { 3, 3 } };
      private byte currState = 0;
      public Alt() { currState = 0; }
      public synchronized void update(int event) {
            Verify.beginAtomic();
            currState = rule[currState][event];
            assert (currState != 3);
            Verify.endAtomic();
      }
      public synchronized void checkExitState() {
            assert (currState % 2 == 0);
      }
}
```

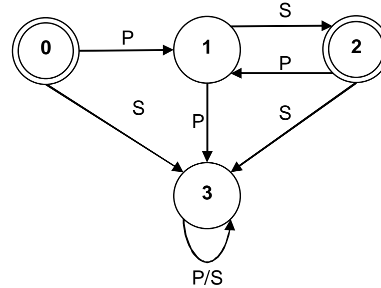

Figure 6. Monitoring Alternating properties and its FSM

0 is the start state. 0 and 2 are two accepting states. 3 is the error state. P represents a causing event. S represents an effect event

Since JPF does not support some Java native methods like the ones of the RandomAccessFile class, we implemented our own RAF class with an array to model the sequence of bytes of a file. In our initial experiment, we tried to run JPF on the same test harness we used to infer properties. Unfortunately, JPF did not terminate after 30 hours when checking a single alternating property. So we created a simpler test harness, DaisyTestSimple, which only creates one file and two threads. Each thread either reads from or writes to the created file once. We used JPF's Verify.random in place of Java's random number generator so that JPF will automatically explore all possible results of the random number generator. To improve performance, we also took advantage of some abstraction facilities provided by JPF. For example, Verify.beginAtomic and Verify.endAtomic can be used to indicate the enclosed statements are an atomic segment. This can significantly reduce the number of states JPF has to check because an atomic segment consisting of multiple statements is treated as a single step. We put the initialization code of DaisyTestSimple and the monitoring code into atomic segments.

JPF may find a counterexample either because of a bug in the implementation or some limitation in our inference approach. As an example of finding a counterexample that reveals a bug[1], consider the property RAF.seek→(RAF.read|RAF.write), which says whenever the seek method of RAF is called, the read or write method must be called (number 22 in Table 6). This is a desirable property inferred in a single threaded environment. JPF found a counterexample in a two-threaded environment, where it is possible that RAF's seek method is called twice without a call to either the read or write method in between. Diagnosing this problem led us to detect a race condition in Daisy's Petal class, which models the disk using

[1] This bug has been detected independently by several groups: Klaus Havelund using JPaX, Willem Visser using JPF, and ourselves through inspecting the traces during initial experiments in July 2004. Since then, we developed this new pattern trying to infer the corresponding property so that we can use JPF to find the bug.

Number	Causing Event (P)	Effect Event (S or S|Q)	Terracotta Mode			JPF Found Violation
			Slice *this*	Slice 1st Arg	alternate events	
1	Daisy.alloc	DaisyDisk.writeAllocBit				
2	Daisy.creat	Daisy.alloc				
3	Daisy.creat	Daisy.ialloc				
4	Daisy.creat	DaisyDisk.writeAllocBit				
5	Daisy.get_attr(inodenum)	Daisy.get_attr(inode)				
6	Daisy.ialloc	Daisy.alloc				
7	Daisy.ialloc	DaisyDisk.writeAllocBit				
8	Daisy.iget	DaisyDisk.readi				✓
9	Daisy.iget	DaisyLock.acqi				✓
10	Daisy.iget	DaisyLock.reli				✓
11	Daisy.read(inodenum)	Daisy.read(inode)				✓
12	Daisy.write(inodenum)	Daisy.write(inode)				✓
13	DaisyDir.writeLong	Utility.longToBytes				
14	DaisyDisk.readi	DaisyLock.reli				
15	DaisyLock.acqb	DaisyLock.relb				
16	DaisyLock.acqi	DaisyDisk.readi				✓
17	DaisyLock.acqi	DaisyLock.reli				✓
18	LockManager.acq	LockManager.rel		✓		✓
19	Mutex.acq	Mutex.rel	✓			✓
20	Petal.read	RAF.length				
21	Petal.write	RAF.writeByte				
22	RAF.seek	RAF.readByte|RAF.writeByte			✓	✓

Table 6. The Alternating properties JPF checked

RandomAccessFile. In particular, both the read and write methods of the Petal class first call RAF.seek to move the file pointer into the intended point where the read or write starts. Then it calls either RAF.read or write. There is no guarantee that these two actions take place without another thread's calling the RAF.seek method in between. So, it is possible that thread 1 first moves the file pointer to the desirable position A, then thread 2 is scheduled to execute, which moves the file pointer to another position B. Next if thread 1 is scheduled to execute again, it will write to the incorrect position which could lead to corruption of the disk.

A counterexample can also be caused by the common limitation of any dynamic analysis technique: some execution scenario that we have not encountered when running the test harness. For example, in our experiment, we only ran the test harness with one thread. Then we used JPF to validate whether those properties we observed in a single thread environment are still true when there are multiple threads. Given these, it is not surprising that JPF found counterexamples because the implementation does not depend on (or try to ensure) such a property in multi-threaded environment.

For example, we inferred an alternating property between two static methods DaisyLock.acqi and DaisyLock.reli, which acquires and releases the lock associated with an inode by calling the LockManager (number 17 in Table 6). The LockManager then tries to lock/unlock an inode by calling its corresponding Mutex object's acq/rel method. As long as the implementation of Mutex's acq/rel method guarantees synchronized access to an inode, it is unnecessary for upper level methods to be accessed in a synchronized way. JPF found counterexamples to properties 8 to 12 and 16 to 19 all for this reason. Finding a violation of such a property still reveals some important design decisions which can serve as a useful document of the target system.

As another example of a "good" counterexample, consider the alternating property, Mutex.acq→Mutex.rel, on a same Mutex object (number 19 in Table 6). Figure 7 shows the implementation of the two methods. Recall that the Mutex.acq event corresponds to entry of the Mutex,acq method, so we inserted a call to the update method at the beginning of each method (i.e., between line 4 and 5, and between line 14 and 15). However JPF quickly found a counterexample where there are two consecutive calls on the same Mutex object's acq method. This was surprising, but does not mean there is a bug in the implementation.

We found that it is perfectly fine to have more than two entrances of the same Mutex object's acq method. The key is that it is impossible for these calls to reach line 12 at the same time, which is the critical section. This is ensured by the Boolean variable locked. Then we reinserted the call to update method between line 11 and 12 and reran JPF. This time JPF did not find any violation. Putting the update call at the very end of both methods did not raise any violation either. We actually also inferred the property that the exit events of the two methods alternate on the same Mutex object. We ended up with greater confidence in a more complete temporal specification of the Mutex class—the acq method can be entered by multiple threads but cannot be finished by multiple threads unless there is a rel method finished in between.

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

Figure 7. The Mutex class in Daisy

If JPF does not find any violation of a property, we have greater confidence in its correctness. One such interesting property is DaisyLock.acqb→DaisyLock.relb (number 15 in Table 6). We were surprised that JPF did not find a counterexample, because it found a violation of a similar property DaisyLock.acqi→DaisyLock.reli. We inspected the source code by hand. In particular, we looked at all call sites of the acqb/relb methods. We found these two methods are always enclosed between calls to acqi/reli methods no matter where the calls take place. In Daisy, a file is associated with a unique inode and block. The inode stores the block number which is used to locate the block that stores the data of the file. That is, one must first get the inode to be able to access the corresponding block (i.e., for read or write). Each inode is associated with a unique lock, and so is each block. As a result, the implementation enforces that a lock on a block would not be successfully acquired unless the lock on the corresponding inode has been acquired first, and similarly a lock on an inode would not be released until the lock on the corresponding block has been released. This explains why acqb and relb satisfy the alternating property even when there are multiple concurrently running threads.

The DaisyDisk.readi→DaisyLock.reli (number 14 in Table 6) property is another interesting property for which JPF did not find a counterexample. We were also surprised by this because JPF found violations to five other closely related properties (properties 8-10, 16, and 17 in Table 6). In Daisy, retrieving an inode consists of a sequence of calls. A method (i.e., Daisy.read or Daisy.write) that needs to access an inode first calls Daisy.iget, within which DaisyLock.acqi and DaisyDisk.readi are called in sequence. After Daisy.iget returns, the starting method first accesses the inode (i.e., reads from or writes to it) and finally calls DaisyLock.reli to release the lock associated with the inode. In a single threaded environment, we successfully inferred this calling chain as an alternating chain. JPF refuted the first five but found no counterexample to the sixth one because DaisyDisk.readi is also always enclosed between acqi and reli (very similar to the previous example).

5 Discussion

The effort required to use Terracotta is low. Most of the work can be automated. In particular, we fully automated the instrumentation and the inference with different modes. Our Java instrumentor works on bytecode, so the source code is not required. Creating a test harness is the only step that may require manual work. We found that developing a test harness based on random testing is fairly effective, which only requires knowledge about the target system's API and so can be created very easily. Since such a test harness, once developed, can be used many times later given that the system's API does not change, the cost of developing it is easily justified. In our experience, it took us about half an hour to create the test harness which we have been using since. All these make it very easy to carry out experiments on Daisy. The most resource-consuming, though automated, step is using a model checker to check the inferred properties (in our experiment, it took JPF about 40 minutes to finish if there was no violation). Additional efforts are required to interpret the inferred properties and counterexamples found. This usually requires users inspect the source code and error trace by hand. Such properties usually are interesting (i.e. revealing bugs or subtle temporal behaviors), so this effort can be easily justified compared to the efforts required to detect such things completely manually.

Our inference engine depends on the set of pre-defined patterns. In this experiment, we found the alternating pattern is a useful abstraction of a category of interesting properties, which can typically be expressed in the form of alternating patterns. A straightforward application of the alternating pattern, however, proves insufficient for detecting some of the interesting properties. Some properties, though in the form of Alternating, might require more than two events (e.g. seek→read|write). Other properties will only be found if contextual information include calling threads and object identities is used to distinguish events (e.g., Mutex.acq→Mutex.rel on a given object).

Some interesting properties, however, cannot be expressed in any of our patterns. For example, Visser used JPF to find a race condition of the allocation bit [28]. Our current patterns cannot express the desirable property that can lead JPF to find the violations. In order to express such a property, we would need to include the Precedence pattern [13]. All these demonstrate the need for some more complex patterns than what we currently use, which our future work will focus on.

We did not find other patterns than Alternating to be very useful in our experiments. In our previous work, we have found patterns like MultiEffect, MultiCause, and EffectFirst to be very useful in support program evolution. In particular, we have found that a pair of events satisfied the Alternating pattern in one version of OpenSSL but only the MultiEffect pattern in another version. This revealed some (previously detected) bugs in OpenSSL [2, 32].

The idea of inferring properties in a single threaded environment and checking them in a multi-threaded one seems to be useful. We have more confidence in those properties found to be consistent in both types of environment (e.g., DaisyLock.acqb→DaisyLock.relb). Those properties that are violated in the multi-threaded environment either gave us more insight into the target system (e.g., Mutex.acq→Mutex.rel) or led us to detect bugs (e.g., seek→read|write). The counterexample produced by a model checker is very helpful for diagnosing the problem.

Using an off-the-shelf model checker to check the inferred properties seems to be feasible, though it might still require some setup work before it can run smoothly. Creating a reasonable environment and some form of abstraction is still required to battle the state explosion problem. Automating this may not be feasible in most scenarios.

6 Related Work

Our work was initially inspired by Ernst's work on dynamically inferring program invariants [14]. Their work focused on inferring data invariants.

There are four other works that also try to infer temporal properties or class interfaces. Alur et al. developed a static analysis for inferring Java class interfaces based on predicate abstraction and symbolic model checking [3]. Ammons et al. developed a technique for learning temporal specifications leveraging a powerful off-the-shelf probabilistic finite automaton learner [4, 5]. Cook et al. invented a dynamic analysis for extracting thread synchronization models from a program's execution traces [10]. Whaley et al. proposed a static and a dynamic approach for inferring what protocols users of a Java class must follow [30]. They also proposed to slice based on the data members of a class. Ours is different in that we slice based on an object's

identity. Our approach also can handle thread information and is able to infer properties across contexts. One common characteristic of these four works is that they all try to extract a complete finite state machine representation either through static analysis or dynamic analysis. As mentioned in Section 2.3, this problem called grammar inference problem has been proved to be NP hard [16, 17]. To achieve better scalability, our light-weight technique only focuses on the relationships among a few events, which represent the majority of properties people care most [13].

Little previous work has been one on using automatic inference techniques to infer properties for use in program verification. Nimmer and Ernst used ESC/Java to check the inferred invariants [12, 24]. Our work is distinct from theirs in that we focus on temporal invariants especially in a multi-threaded system whereas theirs focus on value-based invariants. In addition, we used a model checker to verify inferred properties which makes more sense for temporal properties.

Our approach depends on the availability of efficient and scalable model checkers. Researchers in the model checking community have made significant progress in making model checker scale to fairly large programs and more user-friendly. Many model checkers have been developed [6, 7, 8, 11, 15, 19, 20, 21, 23, 26, 27]. Model checking has been found very useful in domains like device driver or network protocol [6, 26, 21]. All these works require a specification of properties to be checked, which is a big burden for users and limits the wider adoption of such tools. Ours is the first of its kind to automatically infer temporal properties and use a model checker to check them.

7 Conclusion

This paper evaluates the hypothesis that automatically inferring temporal properties and using a model checker can provide useful information about the target program. Our experimental results using this approach on Daisy give us reasons for being optimistic in this approach, which led us to uncover a known bug in the implementation and many important but subtle temporal design decisions. Our techniques for handling context information are very helpful for inferring some interesting properties. In our future work, we plan to develop some more complex property patterns and further evaluate our approach on some larger systems.

Acknowledgements

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 NASA Ames Research Center for providing Java PathFinder and Willem Visser for help run JPF.

References

[1] JRat. http://jrat.sourceforge.net/

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

[3] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. *32nd ACM Symposium on Principles of Programming Languages*, January 2005.

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

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

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

[7] 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.

[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*, 297-319, Volume 53, Number 3, April 2004.

[11] 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.

[12] D. L. Detlefs. An overview of the Extended Static Checking system. *First Workshop on Formal Methods in Software Practice*, January 1996.

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

[14] 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.

[15] P. Godefroid. Model checking for programming languages using VeriSoft. 24th ACM Symposium on Principles of Programming Languages, January 1997.

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

[17] E. Gold. Complexity of automatic identification from given data. *Information and Control*, 37, 302–320, 1978.

[18] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. *International Conference on Software Engineering*, May 2003.

[19] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*. Volume 24, page 189-215, March 2004.

[20] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. *International Conference on Programming Language Design and Implementation*, June 2004.

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

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

[23] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann and W. Grieskamp. Optimal strategies for testing nondeterministic systems. *International Symposium on Software Testing and Analysis*, July 2004.

[24] 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.

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

[26] S. Qadeer and D. Wu. KISS: keep it simple and sequential. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[27] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*. Volume 10, Number 2, April 2003.

[28] W. Visser. Java PathFinder 3.1: the Daisy killer. The talk given at the *Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software*. July 2004. http://research.microsoft.com/~qadeer/cav-issta-talks/jpf-daisy.ppt

[29] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, May 1980.

[30] 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.

[31] J. Yang and D. Evans. Dynamically inferring temporal properties. *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, June 2004.

[32] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. *15th IEEE International Symposium on Software Reliability Engineering*, November 2004.