Perracotta: Mining Temporal API Rules
from Imperfect Traces

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

Deepali Bhardwaj, Thirumalesh Bhat, Manuvir Das
Center for Software Excellence
Microsoft Corporation
{deepalib, thirub, manuvir}@microsoft.com

ABSTRACT

Dynamic inference techniques have been demonstrated to provide useful support for various software engineering tasks including bug finding, test suite evaluation and improvement, and specification generation. To date, however, dynamic inference has only been used effectively on small programs under controlled conditions. In this paper, we identify reasons why scaling dynamic inference techniques has proven difficult, and introduce solutions that enable a dynamic inference technique to scale to large programs and work effectively with the imperfect traces typically available in industrial scenarios. We describe our approximate inference algorithm, present and evaluate heuristics for winnowing the large number of inferred properties to a manageable set of interesting properties, and report on experiments using inferred properties. We evaluate our techniques on JBoss and the Windows kernel. Our tool is able to infer many of the properties checked by the Static Driver Verifier and leads us to discover a previously unknown bug in Windows.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]

General Terms

Reliability, Experimentation, Verification

Keywords

Dynamic analysis, temporal properties, specification inference.

1. INTRODUCTION

Many software tasks require specifications: verifying programs requires specifications of their intended behavior, testing programs requires specifications to determine the input domain and expected outputs, and maintaining programs requires specifications to understand what aspects of the behavior can be modified. Unfortunately, most programs do not come with precise specifications. Worse, those that do often fail to preserve the consistency of specifications and implementations. As the implementation changes, the specification becomes increasingly incorrect.

As a result, several researchers have been motivated to study the problem of specification inference [3, 4, 11, 16, 17, 19, 40, 48, 49]. Although early work in this area emphasized static analysis of the program text [3, 16, 19, 48], more recently several researchers have explored the possibility of using a program's dynamic

behavior on sample executions to infer a specification [4, 11, 17, 40, 49]. Dynamic specification inference has shown promising results in many areas, including bug detection [25, 36, 37, 39], test case selection [24, 26, 50], and program steering [35]. However, all of the results to date have been on small programs.

The scalability and applicability of dynamic inference techniques to industrial programs is limited by several issues:

1. The inference algorithms themselves often scale poorly with the size of the program and input trace.
2. Previous dynamic inference techniques only infer properties that are completely satisfied by the execution traces. This means they require perfect traces, and do not work well in situations where only imperfect traces are available.
3. Many of the properties that are inferred dynamically are often uninteresting. For small programs, it is feasible for the developer to manually winnow the set of inferred properties down to the interesting ones; for large programs, this winnowing must be mostly automated.

Our work seeks to address these challenges and find ways to usefully apply dynamic inference techniques to large programs in industrial scenarios. We focus on dynamic inference of temporal properties. Temporal properties constrain the order of occurrence of program events. For example, acquiring a lock should eventually be followed by releasing the lock. Such properties are very important in software development. Many programs have some inherent temporal behaviors. For example, network protocols can usually be described using a finite state machine. Satisfying temporal properties is essential for an implementation to be correct. Such temporal properties, however, are rarely available.

Manually specifying temporal properties is an expensive, error-prone, and tedious process [28]. In our previous work, we proposed a dynamic analysis approach for automatically inferring temporal properties from a program's execution traces [51] and demonstrated that the inferred properties are useful for supporting program evolution on some small examples [52]. Section 3 summarizes our inference approach and explains how our improved inference algorithm scales to large traces and programs. In attempting to apply our approach to larger programs, however, we encountered the problems identified above. The traces we were able to obtain were imperfect, which prevented many important properties from being inferred. Of the properties inferred, however, a large fraction was uninteresting and for larger programs it became increasingly impractical to manually separate the interesting and uninteresting properties.

The key contributions of this paper involve developing solutions to the problems encountered when dynamic inference techniques are used on industrial programs and evaluating their effectiveness experimentally:

- Addressing the problem of imperfect traces by developing an analysis technique for detecting dominant behaviors from a program's potentially imperfect traces (Section 4).
- Developing two techniques for incorporating contextual information into the inference algorithms (Section 5).
- Introducing two new heuristics for automatically identifying properties that are likely to be interesting and a method for logically combining inferred properties (Section 6).

We have implemented these techniques in our inference engine, Perracotta, and evaluated them in three scenarios, including experiments on inferring API rules for Daisy file system, Windows kernel, and JBoss core components (Section 7). We were able to find 56 interesting rules for Windows APIs, including many properties checked by the Static Driver Verifier [42]. One important use of the inferred properties is to validate the program satisfies those properties using static verifiers. This paper describes our initial effort to verify inferred temporal properties using a static analysis tool. We fed the inferred Windows kernel API rules to the ESP verifier which led us to find a significant bug in the Windows code.

2. RELATED WORK
This section provides background on temporal properties and program verification, and discusses related work in static and dynamic property inference.

2.1 Background
Pnueli developed the theory of temporal logic [38]. Dwyer et al. developed a set of temporal property patterns based on a case study of hundreds of property specifications [15]. Their Bandera specification language facilitates specification of those patterns [12]. Smith et al. developed the Propel approach to make writing and understanding specification easier [44]. Our work tries to infer properties which could be expressed using any of these logics. Since the properties we consider in this paper are simple, we can describe them with regular expressions.

Much research has focused on using specifications of temporal properties to verify systems, especially concurrent systems [2, 5, 9, 10, 13, 14, 18, 27, 33, 47]. All these works require a specification of properties to check, which is a big burden for users and limits the wider adoption of such tools. For our verification experiments, we use Java PathFinder [47] (Section 7.1.2) and ESP [13] (Section 7.3.2).

Java PathFinder (JPF) is an explicit-state model checker for Java programs [47]. It can check deadlock, race conditions, unhandled exceptions, and user-specified assertions. Upon finding a violation of a property, it produces an execution path illustrating the problem.

ESP is a validation tool for typestate properties [46]. ESP allows a user to write a custom specification encoded in a finite state machine to describe typestate transitions. ESP employs an inter-procedural dataflow analysis algorithm [41] to compute the typestate behavior at every program point. ESP uses the property simulation method for combining dataflow analysis and symbolic evaluation. The algorithm computes two sets of information: (a) the property state (typestate according to the specified protocol) and (b) the path simulation state. At a merge point in the control flow, if two symbolic states have the same property state, ESP merges the path simulation states. Otherwise, ESP explores the two paths independently as in a full path-sensitive analysis.

2.2 Static Inference
Tools can attempt to automatically derive specifications from a static analysis of the program text [6, 7, 8] or through a trial and error approach with a static analyzer [19]. Alur et al. developed a static analysis for synthesizing Java class interfaces for a single class [3]. We limit the remainder of our survey to static inference work designed to deal with imperfect code, since this work is similar in spirit with our goal of handling imperfect traces for dynamic inference.

Engler et al. proposed a method for extracting properties by statically examining source code based on a set of pre-defined templates [16]. They can infer the kinds of alternating properties we focus on in this paper. They use a set of specific names to reduce the number of candidate events [16]. They select properties based on three program styles. Weimer et al. invented a static analysis that statistically analyzes incompletely satisfied static properties by examining a program's exception handling routines [48]. The main difference between these works and ours is that our approach uses dynamic traces instead of static analysis of the program text. This enables our techniques to be used in more scenarios such as when source code is not available. Their approaches mainly focus on local properties, while we can identify relationships among events that are far removed from each other in program text. Some of the techniques we develop in this paper could also be applied to possibly improve static inference techniques, as we discuss in Section 8.

2.3 Dynamic Inference
Another approach for inferring specifications uses program execution traces. This paper is the first to present results from applying dynamic inference on a real system whose size is comparable to Windows. Previous work has attempted to extract a complete finite state machine [4, 11, 49], which is historically called the grammar inference problem. Gold proved it is NP-hard [21, 22]. To achieve better scalability, our work only focuses on the relationships among a few events, which represent the majority of properties people care about most [15]. Our chaining heuristic enables us to compose more complex state machines out of the simpler ones. In addition, previous work on dynamic inference (except [11], described at the end of this section) assumes the test program executions are perfect and will not infer properties that are not completely satisfied by the traces. Our inference algorithm is designed for imperfect traces typically found in an industrial setting and can tolerate bugs in the trace as long as the majority of the trace is correct. Next, we survey dynamic inference techniques and explain other differences between previous work and this work.

Ammons et al. used an off-the-shelf probabilistic finite automaton learner to mine temporal and data-dependence specifications for APIs or ADTs from dynamic traces [4]. To handle traces containing bugs, their approach required human experts to decide whether a violation is actually a bug. In contrast, our techniques can automatically tolerate imperfect traces without guidance. Their machine learning algorithm has a high computational cost, whereas our algorithm scales better to larger traces than theirs. Daikon is a tool that automatically infers likely program invariants using statistical inference from a program's execution traces [17]. Daikon's effectiveness has been demonstrated for a variety of applications, such as assisting new programmers to maintain and improve a legacy system [17], test case selection [26], and generating annotations for use by static checkers [37]. Whaley et al. proposed a static and a dynamic approach for inferring what

protocols users of a Java class must follow [49]. The protocols their approach can find are mainly typestate properties and are limited to one class. Our approach is able to discover useful properties among methods from different classes. They sliced their dynamic traces based on a class's field, while we slice based on an object's identity and arguments.

Cook et al. invented a statistical dynamic analysis for extracting thread synchronization models from a program's execution traces [11]. Our work differs from theirs in that we focus on detecting API rules and assume the trace already has the thread information. Reiss et al. developed a technique to compact large volume of execution traces [40]. They use the sequencing properties on individual objects, while we detect rules across multiple objects. DynaMine extracts usage patterns from a system's CVS revision histories and dynamically validates inferred patterns [36]. Their approach is complementary to our work in that examining a CVS history is a way to select events to monitor at run-time. Their mining algorithm has to filter out a fixed set of frequent events to scale to large scenarios, which is not as general as our heuristics. The patterns they inferred tend to focus only on methods within a class, whereas we can infer properties spanning classes.

3. APPROACH

In previous work we proposed a dynamic analysis for automatically inferring simple finite state machines from a program's execution traces [51, 52]. Figure 1 depicts the steps in our approach. First, we instrument the target program to monitor events and states of interest. Then we run the instrumented program through a set of test cases collecting execution traces. Our inference engine then tries to match the traces against a set of pre-defined property templates. Our post-processing component selects and outputs the interesting properties out of the initial results.

For example, the *Alternating* template constrains two events to be in strict alternating order. We can define it in regular expression as $(PS)^*$, where P and S are placeholders and represent two different events. Suppose we monitor the *acquire* and *release* methods of a *lock* class and obtain this trace as follows: *lock.acq*, *lock.rel*, *lock.acq*, *lock.rel*. Our inference engine instantiates the $(PS)^*$ template with each possible pair of events: *(lock.acq lock.rel)** and *(lock.rel lock.acq)**. The first substitution accepts the trace as a valid string, so *Alternating* property is inferred. We denote it as *lock.acq→lock.rel*. The second substitution rejects the trace as a valid string, so no *Alternating* property is inferred.

We developed a hierarchy of eight property templates based on the *Response* pattern (whenever P happens, S must also eventually happen) [15], of which *Alternating* is the strictest. We have found they are useful for comparing the behaviors of several versions of an evolving system [52]. Our inference algorithm infers the strictest template that any two events satisfy [51]. In this paper, we only consider the *Alternating* property pattern. It is the strictest
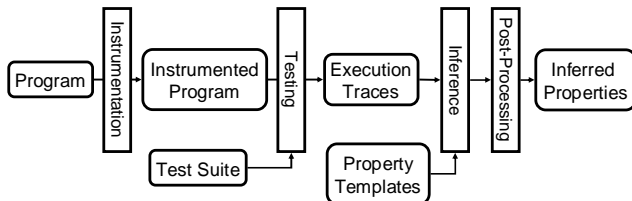
of the template patterns and has proven the most useful in practice.

3.1 Implementation

To enable our approach to scale to large traces, we developed the algorithm for inferring two-event properties with a time complexity $O(nL)$ and a space complexity $O(n^2)$, where n is the number of distinct events and L the length of the trace. Our algorithm first scans the trace and encodes each distinct event as a unique *index* number and then creates an n by n matrix, M. Each row corresponds to a P event, while each column an S event. Each cell of M stores the current state of a state machine that represents the alternating pattern between the corresponding pair of events. The key insight for achieving $O(nL)$ time complexity is that an event X from the trace can be treated as either a P event or an S event. When we read an X from the trace, we update both the row and column corresponding to *index(X)* of M. We hard-coded the templates as tables, so the updating can be done by looking up the tables and has $O(n)$ complexity for each event in the trace. Thus, the overall complexity is $O(nL)$. At the end of processing the trace, the inferred properties are those whose corresponding state machines are in accepting states.

3.2 Limitations

Although our original technique produced promising results when applied to small programs [52], it became ineffective when we attempted to use it on larger programs.

One major limitation of our previous inference algorithm is that it requires complete satisfaction [51]. For example, two events must always appear in alternating order for our technique to infer the *Alternating* property. Hence, our previous algorithm depends on the availability of perfect execution traces. The real world, however, is rarely perfect.

The execution trace may be incomplete because of partial profiling in cases where profiling must be turned on or off on a running system so it is possible that the trace misses some events. Another cause of trace imperfection is missing context information. For example, acquiring a lock and releasing a lock only alternate on the same lock object. If the trace does not include the identity of the lock object, our previous algorithm would not be able to distinguish calls to different lock objects and so would not be able to infer such properties.

The most insurmountable cause of traces being imperfect is buggy programs, even in test suites. Programmers often forget to release allocated resources, especially on rarely executed paths, even when traces are collected from test cases thought to exhibit correct behavior. As a result, allocation and release of the resource would not always alternate with each other in the trace. Our previous algorithm would not infer the *Alternating* property in such cases, since a single violation prevents the property from being satisfied. To address this limitation, we developed a new statistical algorithm that is able to discover the dominant behavior from imperfect traces (Section 4).

One big advantage of dynamic analysis over static analysis is the availability of precise context information (e.g. thread, pointers). Our previous work did not use such information. In this paper, we present two approaches for taking advantage of such information: context-neutral and context-sensitive (Section 5).

Another limitation of our previous dynamic inference techniques is noise. For traces with large number of distinct events, the

Figure 1. Approach overview.

inference results typically include many uninteresting properties. As a result, the relatively few interesting properties are very difficult to find manually. This problem becomes a substantial obstacle when we apply inference to systems with a large number of monitored events. To address this problem, we developed two heuristics for selecting interesting properties, and a method for combining inferred properties (Section 6).

4. APPROXIMATE INFERENCE

It is critical that some important system resources are freed after they are allocated, but sometimes even expert programmers forget to free resources. Similarly, proficient developers often neglect to release an acquired lock along exceptional paths. In order to infer properties from traces in which these buggy paths are executed, we need mechanisms for approximate inference that can infer properties that are not completely satisfied. Our technique works by identifying the dominant properties of an imperfect trace.

Consider a trace where P and S alternate n times but the last three P's do not have a corresponding S: $PSPS...PSPSPPP$. If P corresponds to a resource allocation and S to the corresponding deallocation, the last three allocations in the example trace are not properly deallocated. Intuitively we can see that the dominant behavior of P and S on this particular execution path is still *Alternating*. This becomes even more obvious if we partition the original trace into small subtraces as follows: $PS\ PS\ ...\ PS\ PPP$. The first n partitions all satisfy the *Alternating* property, and only the final partition does not.

We generalize the above observation and formally define what dominant behaviors are. We use the regular expression P^+S^+ to define a sub-trace because it intuitively corresponds to a satisfied or almost satisfied *Alternating* pattern. We can add the sub-trace partitioning as an extra state-machine to our original implementation. As a result the complexity of the algorithm is still $O(nL)$, where n is the number of distinct events, and L the length of the trace.

After partitioning the original trace into sub-traces, we run our inference algorithm on each sub-trace and compute the satisfaction rate of each template. The satisfaction rate of the *Alternating* property is $p_{AL}= n_{AL}/n$ where n_{AL} is the number of partitions that satisfy the *Alternating* template and n is the total number of partitions. For the $PS\ PS\ ...\ PS\ PPP$ example above, we would compute the *Alternating* satisfaction rate as $p_{AL}= n/(n+1)$, which for large n approaches 1.

Then we can rank all pairs of events based on p_{AL}. We can filter the results by setting a threshold for p_{AL} so that we only present those pairs of events whose p_{AL} is above the threshold.

This simple technique does not distinguish between the different kinds of imperfection that may be present in the trace (for example, it does not matter how many Ps are in the last group). Nevertheless, in our experiments it provides an adequate measure of approximate satisfaction for identifying useful properties. We plan to study other ways to partition a trace in the future.

The effectiveness of approximate inference depends on picking a good satisfaction threshold. If the threshold is too high, interesting properties may be missed. But if it is too low, too many false and uninteresting properties will be preserved. Our experiments in Section 7 reveal the importance of picking an appropriate value for p_{AL}.
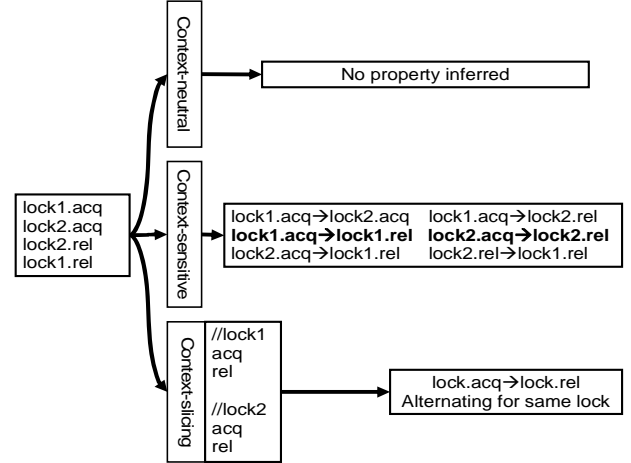

Figure 2. Context handling techniques

5. CONTEXTUAL PROPERTIES

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). A major advantage of dynamic analysis over static analysis is the ready availability of precise context information. We can take advantage of this information to infer more precise 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 the example trace in Figure 2. There are only two distinct events in context-neutral mode (*lock.acq* and *lock.rel*), but four events if we include the object identity in the event context.

Context-neutral analysis does not find an *Alternating* pattern between *lock.acq* and *lock.rel*, where context-sensitive analysis finds six *Alternating* properties shown in Figure 2, of which only the two in bold are useful. Neither context-sensitive nor context-neutral analysis, however, is able to detect that *lock.acq* and *lock.rel* alternate for a same *lock* object. We can precisely obtain this property if we generalize the results of context-sensitive analysis based on same object identity. This is equivalent to slicing the original trace into separate traces based on object identity. Figure 2 shows the two traces obtained by object slicing. Our basic analysis infers the *lock.acq→lock.rel* property from the two new traces. In addition to object identities, we can also treat other types of context information in a similar way. For example, we can slice the trace based on the thread context or based on the value of the ith argument to a method.

The results of context-sensitive analyses are the most complete, but are not useful without generalization. Context-slicing can be viewed as a simple way to generalize the results of context-sensitive analyses. A limitation of context-slicing is that it cannot detect properties that cross contexts, such as an *Alternating* pattern between event P in one thread and event Q in another thread. In future work, we plan to study more sophisticated generalization heuristics to handle this. For now, we analyze a trace using both context-neutral and context-slicing and union their results together. We present our experimental results on evaluating different context handling methods in Section 7.1.

6. SELECTION HEURISTICS

Our inference techniques infer a large number of properties on typical traces, most of which are not interesting to developers. This section presents two heuristics for selecting interesting properties from the inference results and a chaining method for combining properties so properties can be presented in a more useful way.

6.1 Reachability

In our preliminary experiments, we found many properties whose causing event is the wrapper of the effect event. For example, the $A \rightarrow B$ property in Figure 3 is less interesting than the $C \rightarrow D$ property whose two events don't have call relationship. The reason we think $C \rightarrow D$ is more interesting than $A \rightarrow B$ is because C and D are in some sense asynchronized calls whose relationship is not obvious from inspecting either C or D. Another reason is that the $C \rightarrow D$ scenario represents the two classes of interesting properties: resource allocation/deallocation and lock/unlock.

Our call-graph based heuristic marks a property $P \rightarrow S$ as probably uninteresting if S is reachable from P in the call graph. We apply this heuristic by performing a reachability analysis on the static call graph of the target for all the inferred properties. Then we only focus on those properties that involve pairs of events where the second event is not reachable from the first event.

If there is a function call made through a function pointer, we will stop searching on that path. Although a more sophisticated call graph construction algorithm would give us more precise results [23], in our experiments (Sections 7.2.1 and 7.3.1), we found this simple approach allows us to eliminate many uninteresting properties without missing interesting ones.

6.2 Name Similarity

Many non-trivial software systems are developed by strictly following a naming convention such as the Hungarian Naming System [43]. Even when such a naming convention is not strictly adhered to, developers tend to choose similar names for related functions. This makes the code more readable and easier to understand and maintain. Our second heuristic takes advantage of this practice – we expect a property to be more interesting if it involves similarly named events. For example, the event names ExAcquireFastMutexUnsafe and ExReleaseFastMutexUnsafe differ by only one word, and we would expect properties involving those two events to be interesting.

Our heuristic works by partitioning event names into words. We could do the partitioning based on the capitalized letters, underscores, or using a dictionary. Suppose we have a property $P \rightarrow S$. After partitioning, there are w_P words in P and w_S words in S. Suppose there are w common words between P and S. Then we can compute the word similarity of P and S as $2w/(w_P+w_S)$. For example, the word similarity of ExAcquireFastMutexUnsafe and ExReleaseFastMutexUnsafe is 0.8 since four out of five words are identical.

To identify properties that are likely to be interesting, we rank the inferred properties based on their word similarity scores. We can set a threshold (e.g., 0.5 means at least half of the words have to be the same) on the similarity score and only focus on those that are above the threshold. This works best for identifying properties relevant to resource allocation/deallocation and locking disciplines, which are some of the most useful properties whose two events only differ very little in name. The limitation of this

```
A() {               X() {
    ...                 ...
    B ();               C ();
    ...                 ...
                        D ();
                        ...
```

Figure 3. Reachable and unreachable events

heuristic is it can remove important properties whose event names are very different.

6.3 Chaining

In addition to using heuristics to select properties likely to be interesting, we can use logical methods to combine properties to reduce the number of inferred properties and present them in a more useful way. Our chaining technique combines properties by connecting related *Alternating* properties into *chains*. For example, suppose we infer three *Alternating* properties: $A \rightarrow B$, $B \rightarrow C$, and $A \rightarrow C$. Then we can remove $A \rightarrow C$ and say $A \rightarrow B \rightarrow C$ forms an *Alternating chain* (i.e. *(ABC)**). Our algorithm walks through all inferred properties and produces a list of *Alternating chains*.

The chaining method can significantly reduce the number of properties when there are many related properties, since a chain of length l replaces $\frac{1}{2}(l^2 + l)$ *Alternating* properties.

A typical finite state machine in a real system has more than just a few events. For performance reason, our inference technique is limited to inferring properties with only two events. The chaining method provides a way to compose more complex finite state machines out of many small state machines. This allows us to find more complex multi-event properties, without having to suffer from the high computational cost of previous approaches [21, 22].

7. RESULTS

To evaluate our approach, we have implemented our inference algorithm in a prototype tool called Perracotta (available from http://www.cs.virginia.edu/perracotta). It has 12,000 lines of Java code. Perracotta takes a program's function call sequence and produces a list of properties. Perracotta scales well to large traces. The largest example we have tried is a 10 million lines of trace with more than 3,000 distinct events. Perracotta was able to finish in 10 hours on a Sun Sparc workstation with 4GB memory. On most of the traces in our experiments, Perracotta finishes within a few minutes.

To instrument a Java program, we created a plug-in for the JRat bytecode instrumentor [32]. It can record the call sequence, thread information, and the object identity (i.e. the hashcode of *this* object). We used Vulcan-based profiling tool from Microsoft to instrument Windows kernel binaries [45]. It reports function calls with thread information but lacks support for obtaining object identity information.

We report on results from experiments using Perracotta on three different programs: Daisy, a toy file system implementation; JBoss, a Java application server; and the Windows kernel APIs. For Daisy, we fed the inferred properties to the Java PathFinder model checker to verify [1, 47]. We found subtle behavioral differences in its locking discipline across multiple layers. For JBoss, we compared the inferred properties to the J2EE specification [29, 30, 31]. Approximate inference allowed us to detect properties in the specification that we failed to infer in earlier experiments. For Windows kernel, we compared the

inferred properties to those already documented in MSDN and the Static Driver Verifier (SDV for short) [42]. We found a previously unknown deadlock bug in the NTFS file system in Windows Vista using ESP [13].

7.1 Daisy File System

Daisy implements a model of a Unix-like file system in 2,000 lines of Java code [1]. Daisy's architecture has four layers. Daisy uses a RandomAccessFile object to emulate the hard drive, which is the bottom layer. Above it, the Disk layer abstracts the hard drive into byte stream. On top of it, there sits the layer that abstracts the byte stream into blocks. The top layer provides an abstraction of files and directories. We created a test harness that initially creates f files on the disk and starts t threads. Each thread makes a sequence of n calls to randomly selected APIs of the DaisyDir class. In our experiments, we used $f = 5$, $t = 5$ and $n = 15$. We created a wrapper for the RandomAccessFile class so that we could monitor its methods. We used JRat to instrument all methods (except for overridden Object class methods like toString) to record their invocations.

Because the number of properties we inferred for Daisy is small, it is possible for us to look at all properties manually before applying the heuristics for selecting properties. Hence, the main purpose of this experiment is to understand how many interesting properties our heuristics are likely to eliminate.

7.1.1 Inference

The trace has 70,000 events. We sliced it by thread and obtained six sub-traces (five for the child threads and one for the main thread). Without applying any object slicing, we ran approximate inference with a 0.70 threshold for p_{AL} and 10 for event frequency (that is, an event that occurs less than 10 times in the trace will be ignored). Perracotta inferred 70 properties for the 40 distinct events. Only 18 of the inferred properties have a 100% satisfaction rate ($p_{AL}=1$). Approximate inference enabled us to detect some useful properties that would not be found otherwise such as DaisyDisk.readAllocBit → DaisyLock.relb ($p_{AL}=0.97$), and LockManager.acq → LockManager.rel ($p_{AL}=0.86$).

Since there are many noise properties included in the inferred properties, we then applied our chaining method and found nine *Alternating* chains. This significantly reduced the number of properties we had to look at manually to nine chains with 30 properties. Because our chaining method is sound, it does not eliminate any interesting properties.

We then manually inspected the remaining 30 properties and determined that the six shortest chains (with length from one to three events) are not interesting because they are simple wrapper functions. In the remaining three chains, we found they all contain redundant edges due to wrapper functions. This left eight properties that are indeed interesting. This is consistent with the results of using Perracotta's call-graph heuristic.

Next, we applied contextual slicing on the *this* object and the first argument of a method. This led us to infer, with 100% satisfaction, the two properties Mutex.acq→Mutex.rel and LockManager.acq → LockManager.rel that approximate inference was able to detect. Thus, approximate inference allows us to detect useful properties even when important contextual information is unavailable. Object slicing, however, would prevent us from inferring some useful properties that involve more than one object such as LockManager.acq→Mutex.rel.

The call-graph and chaining heuristics were very helpful for reducing the number of properties to consider. Contextual slicing can be used when contextual information is readily available. Its limitation is that it might miss some useful properties, which could prevent us from finding complete chains. So combining the results from approximation and contextual slicing can give us both precision in important properties without missing useful properties. The name similarity heuristic had limited value in this experiment. It prioritizes properties relevant to locking discipline, but it does not work well for the other types of properties in this example.

7.1.2 Verification

We selected several properties including both those we deemed interesting and uninteresting (e.g., two events are reachable in call graph) and validated them using the Java PathFinder. A counter-example can result from imperfect inference. For example, we inferred DaisyLock.acqi→DaisyLock.reli, which acquires and releases the lock associated with an inode by calling the LockManager. The LockManager then tries to lock/unlock an inode by calling the relevant 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.

If JPF does not find any violation of a property, we have increased confidence the property is correct. One such interesting property is DaisyLock.acqb→DaisyLock.relb. We were surprised that JPF did not find a counterexample, because it found a violation of the similar property DaisyLock.acqi→DaisyLock.reli. 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). 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. Similarly, a lock on an inode would not be released until the lock on the corresponding block has been released.

7.2 JBoss

JBoss is currently the most widely used application server, middleware that provides APIs for important services like transactions, security, and caching for running web applications [31]. A Java application server is a particular class of application server that runs on a Java virtual machine. The J2EE specification published by Sun is the main document that defines what developers of web applications and developers of application server should do [30]. We selected the transaction module because a transaction typically involves multiple stages that need to occur in some constrained order. The Java Transaction API specification defines local Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the application, the resource manager, and the application server [29].

7.2.1 Inference Results

The relevant part of JBoss implementation comprises 100,000 lines of Java code. We used JRat to instrument all method invocations and ran the regression test suite that comes with the JBoss distribution. This produced a trace containing 2.5 million events with 91 distinct events (after dropping events that occur less than 10 times). Perracotta executes in 80 seconds.

Figure 4 shows what percent of properties remain as the threshold increases from 0 to 1. We used $p_{AL}=0.90$ to select our properties. The initial result has 490 properties, which is too many to
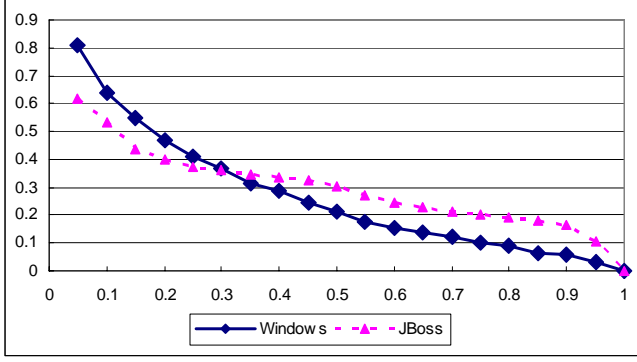
Figure 4. Inferred properties versus satisfaction threshold. The horizontal axis is the threshold varying from 0.0 to 1.0. The vertical axis is the fraction of properties that above the threshold.

reasonably inspect by hand. We first applied our chaining heuristic on them and produced 17 chains out of 61 properties. We then enhanced the result by applying the call-graph based heuristic. This brings the number of chains down to 16 with 41 properties. We did not find the edit distance heuristic useful in this case since being Java program, JBoss had few properties that deal with resource management and locking disciplines.

7.2.2 Comparison with JTA specification

One of the most interesting results is that the longest chain we inferred is almost identical to an object interaction diagram shown in the Java Transaction API (JTA) specification. Our longest chain, shown in Figure 5, has 22 properties, which includes not only the JTA APIs, but also the internal implementation of JBoss. For comparison purposes, we removed those properties about internal implementation and present the resulted chain in Figure 6. The TxManager and TransactionImpl classes implement the JTA TransactionManager and Transaction interfaces respectively.

The JTA specification has a diagram to illustrate how an application server may handle a transactional connection request from an application (as clearly indicated in the JTA specification, this is just one typical scenario but not prescriptive) [29]. Our inferred alternating chain captures most of the scenario. An application server starts a transaction by calling the *begin* method of the
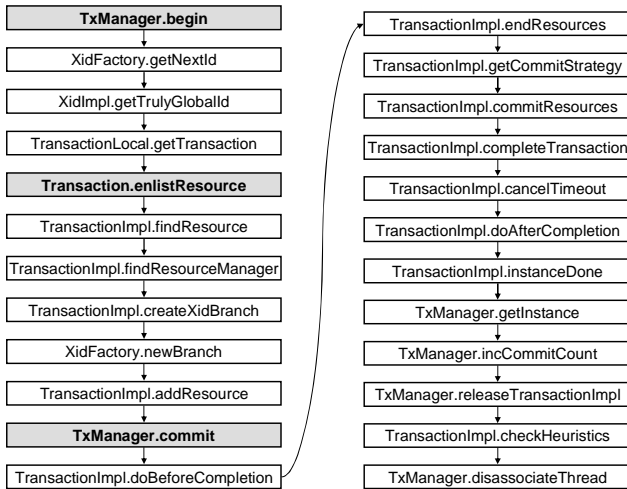
transaction manager. After getting a transactional resource from the resource manager, the application server calls the enlistResource method. Then the application does its work. The application server calls the delistResource method to release a resource and then commit the transaction. Here, Perracotta missed the important edge between enlistResource and delistResource because whenever enlistResource is called, either delistResource or commitResources must be called. In other words, a Resource does not have to be delisted. As shown in Figure 6, we successfully inferred the alternating relationship between enlistResource and commitResource because it is the *dominant* behavior exposed in the trace.

Our results reveal more than just how the APIs interact. It also has information on how internal implementation works, which would be useful for new developers to understand how JBoss works. For example, we successfully inferred a chain shown in Figure 5 that reveals how starting and committing a transaction is implemented respectively in JBoss. We also found useful properties like TransactionImpl.lock→TransactionImpl.unlock.

7.3 Windows Kernel

Our final experiment is to infer API rules for the latest kernel (ntoskrnl.exe) and core components (hal.dll and ntdll.dll) of Windows Vista. Because of limitations of the tracing tool, we only have calling sequence and thread information in our traces and cannot distinguish between objects. To our surprise, Perracotta's approximation algorithm was still able to infer many useful properties such as locking disciplines that typically only alternate for the same object. We compared our inferred properties against the list of properties checked by the Static Driver Verifier [42]. We found we inferred not only many of those properties, but also properties they did not document. We also selected 10 properties and fed them to the ESP verifier. We found a serious bug in one of the Windows binaries that calls the kernel APIs. Section 7.3.1 presents our inference results using approximation and the heuristics. Section 7.3.2 discusses our ESP verification results.

7.3.1 Inference results

We instrumented the APIs of the Windows kernel and core components and obtained 17 execution traces by running some typical windows applications (e.g., Windows MediaPlayer, Windows MovieMaker). The lengths of the traces range from 300,000 to 750,000 events, for 5.8 million total trace events. The number of distinct events in each trace varies from 30 to 1,300. On average each execution trace has 500 distinct events. Perracotta analyzes all traces in 14 minutes.

Figure 4 shows what percent of properties remain as the satisfaction threshold increases from 0 to 1. As with JBoss, we set p_{AL}=0.90, which identified 7,611 properties. This is too many to manually inspect, so we applied the call-graph and edit distance
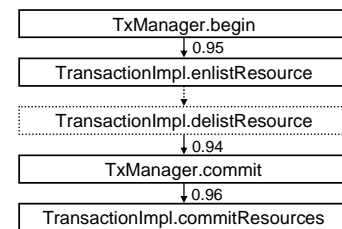

Figure 5. Inferred implementation chain.
(Grayed events are interface events.)

Figure 6. Inferred interface chain

	Properties	Name Similarity (>0.5)		Call Graph Only				Both	
		Properties	Reduction	Unreachable	Unknown	Total	Reduction	Properties	Reduction
Kernel	436	33	92.4%	331	16	347	21.2%	32	92.66%
Non-Kernel	7175	152	97.9%	2949	3310	6259	23.7%	110	98.47%
Total	7611	185	97.6%	3280	3326	6606	23.5%	142	98.13%

Table 1. Impact of selection heuristics.

heuristics to identify the interesting ones. This sharply reduced the number of properties to 142.

Table 1 summarizes the impact of the two heuristics we developed for identifying useful properties. Name similarity is very effective on the Windows code. For example, for Kernel API only, without any heuristics, we inferred 436 properties. Name similarity alone cut down the number to 33, which is a 92% reduction. Although the call-graph based heuristic did not have a reduction rate as large as editing distance, it is still very helpful for reducing the number of properties. For example, we used the call-graph of ntoskrnl.exe generated by ESP on the 436 properties. We found there are 89 properties whose second event is reachable from the first one.

To demonstrate the need for our inference technique also, we conducted an experiment where we applied the name similarity heuristics on all possible event pairs. We found there are 656 pairs of events that have a similarity score greater than 0.5. We then used the call-graph heuristic, which resulted in 572 pairs of events. As a result, Perracotta's inference algorithm was able to cut down the properties by 75% even if we apply the two heuristics first. In this case, nearly all of the eliminated properties are undesirable since they were not satisfied more often than our approximation threshold of 0.90. This shows our dynamic inference is necessary to find meaningful properties.

We manually inspected the 142 properties remaining after our selecting heuristics and identified 56 apparently useful ones, which is 40% of the 142 properties. The properties we deemed interesting are relevant to either resource allocation/deallocation or locking discipline. Table 2 shows 20 sample properties. We found the approximation algorithm is essential for detecting useful properties that otherwise would be missing. A surprising result is we were still able to infer those type-state properties that only alternate for the same object (such as ObpCreateHandle →ObpCloseHandle), even though there is no object information in the trace we had.

It was not feasible to examine all the 7,611 properties initially inferred to see how many interesting were accidentally eliminated. Based on our experiences, however, only about 1% of the initially inferred properties appear to be interesting. Our selection heuristics are very effective for increasing the density of interesting properties in the result to 40%. At this density, it is reasonable for a developer to manually inspect the remaining properties.

We compared the properties we inferred to those checked by the SDV, and found that Perracotta had inferred four out of the 16 properties that the SDV checks [42]. For example, KeAcquireQueuedSpinLock→KeReleaseQueuedSpinLock.

Terracotta also inferred many properties that SDV does not check. For example,

    MmSecureVirtualMemory→MmUnsecureVirtualMemory
is documented as required in MSDN but the SDV does not check this because it is not related to device drivers. We missed seven properties the SDV checks because our trace does not cover those events. For example, KeAcquireSpinLock→ KeReleaseSpinLock. We missed the other five properties SDV checks because of the limitations of our current property templates.

More interestingly, we inferred some properties SDV could have included such as KiAcquireSpinLock→KiReleaseSpinLock and KfAcquireSpinLock →KfReleaseSpinLock. The SDV developers did not to include these critical properties because they are Windows internal functions that are not visible to driver developers.

7.3.2 Verification
We manually selected ten of the inferred properties and fed them into the ESP verifier, which checks temporal properties on Windows binaries [13]. This led us to find one previously unknown serious bug in the NTFS file system. The property is a typical locking discipline property that constraints acquiring kernel Mutex must be followed by releasing the same Mutex. The bug is a double-acquire fault on an exceptional path, where a fastmutex is acquired twice without being released in between. This bug can cause the system to deadlock. ESP clearly showed us an execution path that can activate the double acquire bug. The

p_{AL}	Property	p_{AL}	Property
1.0	ExAcquireFastMutex->ExReleaseFastMutex	0.993	ObpCreateHandle→ObpCloseHandle
1.0	**IoAcquireVpbSpinLock→IoReleaseVpbSpinLock**	0.988	**GreLockDisplay→GreUnlockDisplay**
1.0	**ExAcquireRundownProtectionCacheAwareEx→ ExReleaseRundownProtectionCacheAwareEx**	0.985	RtlActivateActivationContextUnsafeFast→ RtlDeactivateActivationContextUnsafeFast
1.0	KefAcquireSpinLockAtDpcLevel→ KefReleaseSpinLockFromDpcLevel	0.982	KeAcquireInStackQueuedSpinLock→ KeReleaseInStackQueuedSpinLock
1.0	KeAcquireQueuedSpinLock→KeReleaseQueuedSpinLock	0.977	SeCreateAccessState→SeDeleteAccessState
1.0	KfAcquireSpinLock->KfReleaseSpinLock	0.972	IoAllocateIrp→IoFreeIrp
1.0	KiAcquireSpinLock->KiReleaseSpinLock	0.961	**CmpLockRegistry→CmpUnlockRegistry**
1.0	MmSecureVirtualMemory→MmUnsecureVirtualMemory	0.959	**ObAssignSecurity→ObDeassignSecurity**
1.0	**ObpAllocateObjectNameBuffer→ObpFreeObjectNameBuffer**	0.954	ExCreateHandle→ExDestroyHandle
1.0	SeLockSubjectContext→SeUnlockSubjectContext	0.954	**ExpAllocateHandleTableEntry→ExpFreeHandleTableEntry**

Table 2. Selected Properties Inferred for Windows. Properties in bold are not documented in either MSDN or checked by SDV.

Windows development team confirmed it is a real bug and subsequently fixed the problem.

8. DISCUSSION

In our experience with industrial programs, we found that it was not usually possible to obtain perfect traces. Very often some important information (e.g., object identity) is missing or incomplete in the trace due to limited tracing techniques. Another important reason for imperfect traces was exposed in the JBoss experiment – tests can fail or be designed to deliberately test improper API usage, thus introducing imperfections in the traces. For example, good test suites include intentionally bad call sequences to test how the application handles exceptions. Instrumentation might also interfere with the normal run and cause the test to fail. The last and most fundamental reason for the trace to be imperfect is that the program can have bugs. If the buggy path is executed, the trace may contain some false behaviors. Ammons et al.'s earlier work made the same observation [4]. The imperfect trace they had prevented their approach from being fully automated. In their case, they were dealing with small programs and able to deal with the imperfections with substantial human guidance. Our approximate inference is able to tolerate small amounts of noise in the trace automatically, so it can be applied even when perfect tracing tools or immaculate test programs are not available. It works because most realistic programs have been through non-trivial testing, which ensures its main paths to be correct. The latent bugs that escape from testing typically reside on infrequent paths, which do not represent the program's dominant behavior.

The approximate inference increases the likelihood that important properties are found, but those properties are still hidden within the excessive number of mostly uninteresting properties that are typically inferred for large programs. Our chaining method is a simple and effective way to synthesize larger state machines from small ones to reduce the number of properties.

Our call-graph heuristic is also a more systematic way to select properties than previous work [16]. For the alternating properties, Engler et al. used three very specific programming styles to select the traces [16]. Our approach does not have that restriction. We have found it very effective in both C and Java programs. It works because it captures properties for which developers are more likely to make mistakes. For example, acquiring a lock and releasing a lock is very often performed by two different functions and hence even an experienced developer might forget to release a lock or accidentally acquire a lock twice.

Engler's earlier work used naming conventions to select properties in an ad-hoc way by looking for specific keywords [16], while our name similarity heuristic is more systematic and general. We found this approach works best when there are many properties relevant to locking disciplines and resource management since the event pairs in these properties have similar names. This was the case for the Windows code, but not for Daisy or JBoss since Java programs tend not to have such properties. We plan to further investigate other selection heuristics that account for class hierarchies and multiple events in alternating chains.

We found the inferred properties can be used effectively to understand and check a program. Feeding properties to a verifier like ESP or JPF is a promising way to detect behavioral defects. We were able to find a serious bug in Windows using this approach. Current program verification tools, however, are not at the point where this can be done in a fully automated way. A programmer has to manually inspect the counter-example traces to determine whether a reported violation is a real defect. Analyzing the ESP results for an inferred property consumed about a day of human effort. We believe there is a promising future for closer integration between static and dynamic techniques. Perhaps it will be possible to reduce the false positive rate of static analysis tools by using data from dynamic analyses.

9. CONCLUSION

Perfect, fully-automatic specification inference for industrial programs remains an elusive goal, well beyond the state-of-the-art. We have shown, however, that by targeting simple properties that can be efficiently discovered and by using approximation inference techniques along with heuristics for pruning the set of inferred properties, it is possible to obtain useful results even on large, complex programs. Some manual effort is still required, but after the heuristics are applied the task is fairly manageable even for programs as large and complex as JBoss and Windows.

Our results so far are limited to simple alternating properties involving only two events. Despite this, we are encouraged by how many useful properties we find. In future work, we plan to extend the techniques to deal with more complex properties involving three or more events, as well as properties that combine data constraints with temporal ones. Our approximation technique and selection heuristics are general enough to be applied to other types of properties, although it remains to be seen if they will be effective in these situations.

ACKNOWLEDGEMENTS

REFERENCES

[1] *Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software*. Jul. 2004.

[2] T. Andrews, S. Qadeer, J. Rehof, S. K. Rajamani, and Y. Xie. Zing: exploiting program structure for model checking concurrent software. *International Conference on Concurrency Theory*, Aug./Sep. 2004.

[3] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. *POPL*, Jan. 2005.

[4] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. *POPL*, Jan. 2002.

[5] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *SPIN*, May 2001.

[6] K. Bennett, T. Bull, E. Younger, and Z. Luo. Bylands: reverse engineering safety-critical systems. *ICSM*, Oct. 1995.

[7] J. Bowen, P. Breuer, and K. Lano. Formal specifications in software maintenance: from code to Z++ and back again. *Information and Software Technology*. Nov./Dec. 1993.

[8] P. T. Breuer and K. Lano. Creating specifications from code: reverse-engineering techniques. *Journal of Software Maintenance: Research and Practice*. Vol 3. 1991.

[9] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *ICSE*, May 2003.

[10] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. *ACM Conference on Computer and Communications Security*, Nov. 2002.

[11] J. E. Cook, Z. Du, C. Liu, and A. L. Wolf. Discovering models of behavior for concurrent workflows. *Computers in Industry*, Apr. 2004.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the Bandera specification language. *International Journal on Software Tools for Technology Transfer* 4(1): 34-56. 2002.

[13] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. *PLDI*, Jun. 2002.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. PLDI, Jun. 2002.

[15] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. *ICSE*, May 1999.

[16] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *ACM Symposium on Operating Systems Principles*, Oct. 2001.

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

[18] D. Evans. Static detection of dynamic memory errors. *PLDI*, May 1996.

[19] C. Flanagan, R. Joshi, K. Rustan, and M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, Feb. 2001.

[20] P. Godefroid. Model checking for programming languages using VeriSoft. *POPL*, Jan. 1997.

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

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

[23] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6), 685-746, Nov. 2001.

[24] N. Gupta. Generating test data for dynamically discovering likely program invariants. *Workshop on Dynamic Analysis*, May 2003.

[25] S. Hagnal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. *ICSE*, May 2002.

[26] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. *ICSE*, May 2003.

[27] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design.*. Mar. 2004.

[28] G. Holzmann. The logic of bugs. *SIGSOFT FSE*, Nov. 2002.

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

[30] J2EE. http://java.sun.com/j2ee/index.jsp

[31] JBoss. http://www.jboss.org

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

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

[34] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. *PLDI*, Jun. 2005.

[35] L. Lin and M. D. Ernst. Improving adaptability via program steering. *ISSTA*, Jul. 2004.

[36] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. *SIGSOFT FSE*, Sep. 2005.

[37] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: an empirical evaluation. *SIGSOFT FSE*, Nov. 2002.

[38] A. Pnueli. The temporal logic of programs. *Annual Symposium on Foundations of Computer Science*, Oct./Nov. 1977.

[39] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. *International Symposium on Automated and Analysis-Driven Debugging.* Sep. 2003.

[40] S. P. Reiss and M. Renieris. Encoding program executions. *ICSE*, May 2001.

[41] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. *POPL*, Jan. 1995.

[42] Static Driver Verifier: Finding bugs in device drivers at compile-time. *WinHEC*, Apr. 2004.

[43] C. Simonyi. Hungarian notation. MSDN library.

[44] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. PROPEL: an approach supporting property elucidation. *ICSE*, May 2002.

[45] A. Edwards, A. Srivastava, and H. Vo. Vulcan: binary transformation in a distributed environment. *Research Technical Report, MSR-TR-2001-50*, Apr. 2001.

[46] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, Vol 12(1), Jan. 1986.

[47] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, Apr. 2003.

[48] W. Weimer and G. Necula. Mining temporal specifications for error detection. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Apr. 2005.

[49] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *ISSTA*, Jul. 2002.

[50] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal,* 2006.

[51] J. Yang and D. Evans. Dynamically inferring temporal properties. *ACM Workshop on Program Analysis for Software Tools and Engineering*, Jun. 2004.

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