9-9-2006

Adaptive Online Program Analysis: Concepts, Infrastructure, and Applications

Matthew B. Dwyer
University of Nebraska - Lincoln, dwyer@cse.unl.edu

Alex Kinneer
University of Nebraska - Lincoln, akinneer@cse.unl.edu

Sebastian Elbaum
University of Nebraska-Lincoln, elbaum@cse.unl.edu

Technical Report TR-UNL-CSE-2006-0011, Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, Nebraska, USA, 9 September 2006

Adaptive Online Program Analysis: Concepts, Infrastructure, and Applications*

Matthew B. Dwyer, Alex Kinneer, Sebastian Elbaum
Department of Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, Nebraska 68588-0115, USA
{dwyer, akinneer, elbaum}@cse.unl.edu

9 September 2006

Abstract

Dynamic analysis of state-based properties is being applied to problems such as validation, intrusion detection, and program steering and reconfiguration. Dynamic analysis of such properties, however, is used rarely in practice due to its associated run-time overhead that causes multiple orders of magnitude slowdown of program execution. In this paper, we present an approach for exploiting the state-fullness of specifications to reduce the cost of dynamic program analysis. With our approach, the results of the analysis are guaranteed to be identical to those of the traditional, expensive dynamic analyses, yet with overheads between 23% and 33% relative to the un-instrumented application, for a range of non-trivial analyses. We describe the principles behind our adaptive online program analysis technique, extensions to our Java run-time analysis framework that support such analyses, and report on the performance and capabilities of two different families of adaptive online program analyses.

Dynamic analysis of state-based properties is being applied to problems such as validation, intrusion detection, and program steering and reconfiguration. Dynamic analysis of such properties, however, is used rarely in practice due to its associated run-time overhead that causes multiple orders of magnitude slowdown of program execution. In this paper, we present an approach for exploiting the state-fullness of specifications to reduce the cost of dynamic program analysis. With our approach, the results of the analysis are guaranteed to be identical to those of the traditional, expensive dynamic analyses, yet with overheads between 23% and 33% relative to the un-instrumented application, for a range of non-trivial analyses. We describe the principles behind our adaptive online program analysis technique, extensions to our Java run-time analysis framework that support such analyses, and report on the performance and capabilities of two different families of adaptive online program analyses.

1 Introduction

Run-time program monitoring has traditionally been used to analyze program performance to identify performance bottlenecks or memory usage anomalies. These techniques are well-understood and have been embodied in widely

*This work was supported in part by the National Science Foundation through awards 0429149, 0444167, 0454203, and 0541263. We would like to specially thank Heather Conboy for her support of our use of the Laser FSA package.

available tools that allow them to be regarded as part of normal engineering practice for the development of large software systems.

Researchers have sought to enrich the class of program properties that are amenable to run-time monitoring beyond performance monitoring, to treat stateful properties that were previously amenable only to static analysis or verification techniques. For example, a range of run-time monitoring approaches to check conformance with temporal sequencing constraints [9, 16, 21] and to find concurrency errors [14, 29, 32] have been proposed in recent years.

Existing monitoring functionality, however, is not entirely adequate to support such analyses. Techniques aimed at reducing the necessary instrumentation to monitor a program were designed for simpler properties related to program structures, such as basic blocks and paths [1, 4], and are not applicable to more complex properties. Sampling techniques on the other hand can effectively control the overhead, but their lossy nature makes them inappropriate for properties that depend on exact sequencing information, since skipping an action may result in either a false report of conformance or of error. The inability to drastically reduce instrumentation or utilize sampling makes these analyses expensive, with run-time overheads ranging from a factor of "20-40" [14] to "several orders of magnitude" [5][1]. As a consequence they have not been widely adopted by practitioners.

Reducing the interleaving of analysis and program execution can cut down on the overhead. A dynamic analysis that only records information during program execution and then analyzes that information after the program terminates is called an *offline* analysis. In contrast, an *online* analysis interleaves the analysis and recording of program information with program execution. Offline approaches are more common since they naturally decouple the recording and analysis tasks. One advantage of an online analysis is that it obviates the need to store potentially large trace files (an analysis such as the one reported by [36] deals with traces containing millions of events). The analysis consumes the trace on-the-fly during execution and simply produces the analysis result. In addition, rich dynamic program analyses are emerging as the trigger to drive program steering and reconfiguration, e.g., [6], which demands that the analysis be performed online.

In this paper, we present *adaptive online program analysis* (AOPA) as a means of significantly reducing the run-time overhead of performing dynamic analyses of stateful program properties. It may seem counter-intuitive to advocate an online approach to reducing analysis cost, but AOPA's performance advantage comes from using intermediate analysis results to reduce the number of instrumentation probes and the amount of program information that needs to be subsequently recorded. *AOPA builds on the observation that at any point during a stateful analysis only a small subset of program behavior is of interest.* Researchers have observed this to be the case for accumulating program coverage information [7, 26, 31]. In these approaches, the instrumentation for a basic block is removed once that block's coverage information has been recorded, and the analysis proceeds by monotonically decreasing the program instrumentation until complete coverage is achieved; the remainder of the program runs subsequently with no overhead.

AOPA generalizes this by allowing both the removal and the addition of instrumentation to detect program behavior relevant to a *specific state* of an analysis. Contrary to the pervasive sampling approaches, an AOPA analysis is guaranteed to produce the same results as a *non-adaptive analysis*, which maintains all relevant instrumentation throughout the program execution. While property preserving, the removal of instrumentation at points during analysis can lead

[1]Most published research in this area fails to even mention run-time overhead, much less provide clear performance measurements as was reported for Atomizer [14], so we assume that it is one of the better performing techniques.

```
public class File {
  public void open(String name) {..}
  public void close() {..}
  public char read() {..}
  public void write(char c) {..}
  public boolean eof() {..}
}
```

```
public static void main(String[] argv) {
  File f = new File();
  f.open(argv[0]);
  try { ..
    while (!f.eof()) {
      c = f.read(); ..
    }
  } catch (Exception e) { ..
  } finally { f.close(); }
}
```

Figure 1: File API (left) and Client Application (right)

to orders of magnitude reduction of the overhead of monitoring and analysis.

In the next section, we provide an overview of an AOPA applied to a toy program to illustrate the concepts introduced in the remainder of the paper. In addition to introducing the concept of AOPA, this paper makes several additional contributions. (1) In Section 3, we describe the implementation of an efficient infrastructure to support adaptive program analysis of Java programs using the `Sofya` [22] analysis framework; (2) We define correctness criteria for adaptive online conformance checking of programs against finite-state automata specifications, describe an implementation of a family of such analyses, and present performance results for those analyses over a small set of properties and applications in Section 4; and (3) We detail adaptive approaches to implementing another class of analyses that infer sequencing properties from program runs in Section 5. We discuss related work in Section 6, and outline several additional optimizations we plan to implement to further reduce the cost of AOPA. We address other directions for future work in Section 7.

2 Overview

We illustrate the principles of adaptive online analysis by way of an example. The top of Figure 1 sketches a simple `File` class with five methods in its API. The legal sequences of calls on this API are defined by the regular expression:

```
(open; (read | write | eof)*; close)*
```

This type of *object protocol* is commonly used in informal documentation to describe how clients may use an API. When formalized it can be used to analyze client code to determine conformance and detect API usage errors.

The bottom of Figure 1 sketches a simple client application of the `File` API. It instantiates an instance of the class, and then proceeds with a sequence of calls on the API to read the contents of a file. By inspection it is clear that this sequence of calls is consistent with the object protocol, whose finite state automaton description is shown in Figure 2. A traditional offline or online analysis to check the conformance of this program with the object protocol specification will consider a sequence of $3 + 2k$ calls where k is the length of the input file in characters; the sequence consists of single calls to `open` and `close`, a call to `eof` and `read` for each character, and an extra call to `eof` when the end of file is actually reached.

An *adaptive* online analysis that proves conformance with this object protocol will only need to process 2 calls. The analysis calculates for each state of the FSA the set of symbols that label *self-loop* transitions, i.e., transitions whose source and destination is the same state, and *outgoing* transitions, i.e., transitions to different states including

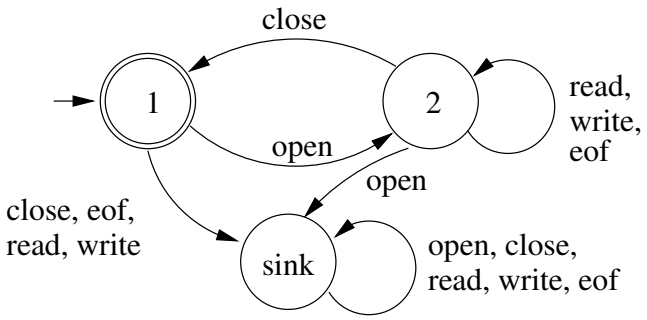

Figure 2: File API Protocol FSA

State	Self Symbols	Outgoing Symbols
1	{}	Σ
2	{read, write, eof}	{open, close}
sink	Σ	{}

Table 1: Self and Outgoing Symbols

the *sink* state. Let $\Sigma = \{$`open, close, read, write, eof`$\}$ denote the set of symbols for the FSA. Table 1 defines the self and outgoing symbols for the FSA. The adaptive analysis begins in the start state, i.e., state 1, and enables instrumentation for all outgoing symbols in that state, i.e., Σ. The first call on the API is `open` and the analysis transitions to state 2. In state 2, the analysis now disables instrumentation for {`read, write, eof`} since the occurrence of any of those symbols will not change the state of the analysis. From the perspective of state 2, those symbols are irrelevant. Obviously this has a dramatic effect on the run-time of the analysis since the `eof` and `read` calls in the loop are completely ignored by the analysis and the loop executes at the speed of the original program. When the `close` call executes, the analysis transitions back to state 1 and re-enables all instrumentation.

The adaptive analysis does incur some cost to calculate the instrumentation to add and remove. Self and outgoing symbol sets are easily calculated before analysis begins. During analysis, symbol sets are differenced each time a state transition is taken to update the enabled instrumentation. Our experience, which is discussed in detail in Sections 4 and 5, is that the reduction in instrumentation more than compensates for the costs of calculating self and outgoing symbol sets.

2.1 Breadth of Applicability

The simple example just presented illustrates that adaptive analysis *can* lead to non-trivial reductions in analysis cost. We are aware, however, that this approach may not always render such improvements. For example, the non-adaptive analysis described in Figure 1 would only consider 3 calls if $k = 0$, i.e., the file is empty. Or, for a property stating that `open` must precede `close`, i.e.,

```
(~[close]*) | (~[close]*; open; .*)*
```

where `~[ ]` means any symbol not inside the brackets, we could restrict the instrumentation to only `open` and `close` calls. Since our program only has 2 such calls, the adaptive and non-adaptive analyses will process the same number

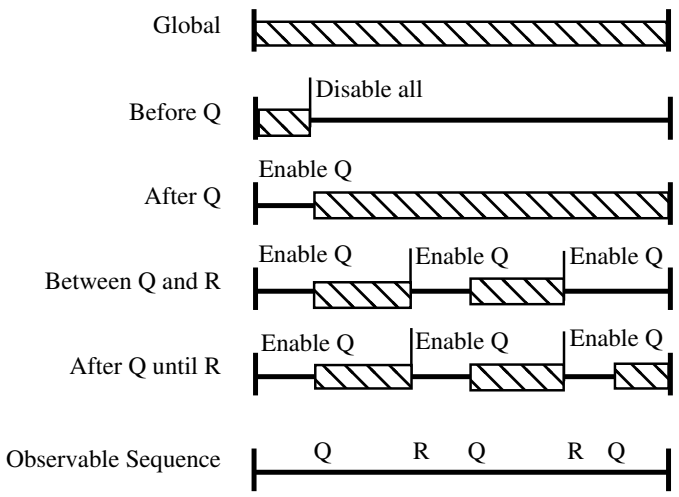

Figure 3: Specification Pattern Scopes

of calls. We are therefore interested in understanding the extent to which these benefits are observed over a range of different analysis problems, programs under analysis, program execution contexts, and properties analyzed.

While the benefits of adaptive analysis depend on the interplay between the property and program under analysis, we believe that two characteristics of program properties can be identified that may lend themselves to efficient adaptive analysis. (1) In [12] we defined property specifications using a concept called a *scope*. Figure 3 shows five kinds of scopes that delimit regions of program execution within which a property should be enforced – the hashed regions – outside of which a property need not hold. Consequently, when exiting a scope, all instrumentation can be disabled except for the instrumentation for the observable that defines the entry to that scope. (2) We found in [12], by studying existing temporal sequencing specifications, that properties like the cyclic `open-close` and precedence properties above occur quite commonly, and in more than 64% of the 550 specifications we studied there are significant opportunities for removing instrumentation. The remaining 36% of the specifications were invariants, which can be checked by predicates instrumented into the program and do not require the stateful correlation of multiple program observations.

Our preliminary findings, while admittedly limited, are very encouraging. We have discovered two broad classes of dynamic analysis problems that hold promise for significant performance improvement through the use of adaptive analysis techniques. These analyses exhibit low-overhead relative to the execution time of the un-instrumented program, which stands in marked contrast to the multiplicative factors, and orders of magnitude, overhead that have been reported for dynamic analysis of stateful properties by other researchers [5, 14].

The next section explains how we exploit recent enhancements to the virtual machine and Java Debug Interface (JDI) to achieve efficient re-instrumentation of a running program.

3 Adaptive Analysis Infrastructure

We have built adaptive online program analysis capabilities into the `Sofya` [22] framework. This framework enables the rapid development of dynamic analysis techniques by hiding behind a layer of abstraction the details of efficiently and correctly capturing and delivering required program observations. Observations captured by `Sofya` are delivered as events to *event listeners* registered with an *event dispatcher*. Clients of the framework request events using a specification written in a simple language.

`Sofya` also provides components at the level of the listener interface to manipulate streams of events via filtering, splitting, and routing. For the purposes of our discussion, we note especially that `Sofya` provides an *object based splitter* that sends events related to different object instances to different listeners. Such a splitter uses a factory to obtain a listener for each unique object observed in the program and direct events related to that object to that listener.

To capture observations efficiently and faithfully[2] in both single and multi-threaded programs, `Sofya` employs a novel combination of byte code instrumentation with the Java Debug Interface (JDI) [19] – an interface that enables a debugger in one virtual machine to monitor and manage the execution of a program in another virtual machine. Instrumentation is used to capture some events because the JDI does not provide all of the events that are potentially interesting to program analyses (such as acquisition and release of locks[3]), and because it cannot deliver some events efficiently (such as method entry and exit). This instrumentation operates by coordinating with the JDI, using breakpoints to insert events into the JDI event stream and deliver the information payloads associated with those events. Because the JDI provides a very efficient implementation of breakpoints, this introduces effectively zero overhead when breakpoints are not being triggered. Additions and enhancements to the virtual machine and debug interface in Java 1.5 have enabled us to implement features in `Sofya` to enable and disable the delivery of such observations as the program is running, including by addition and removal of byte code instrumentation during execution.

Adaptive configuration of program observations requires a mechanism for correlating the desired observations with associated JDI event requests and the probes inserted into the byte code by the instrumentor. For this purpose, we have implemented components in `Sofya` that maintain the current mutable specification of requested program observations, a mapping from observables to the JDI requests to enable or disable the necessary events, and logs of currently active byte code probes. When a request to enable or disable a program observable is received, the mutable specification is updated. If the observable maps to events raised purely within the JDI, `Sofya` simply enables or disables the associated event requests. Otherwise, `Sofya` sends the byte codes for affected classes to the instrumentor, which uses the probe logs to remove probes for disabled observations, and the updated specification of requested observations to add probes for the new events. An observer is attached to the instrumentor to update the probe logs as the changes are made.[4]

The adaptive features are implemented within the `Sofya` framework by providing an online API, an excerpt of which is shown in Figure 4, to enable and disable the events delivered in the event stream at any time. Components of a client analysis that want to utilize the adaptive instrumentation functionality register to receive a reference to

[2] With respect to ordering, and with as little perturbation as possible.

[3] Java 1.6 will provide contended lock events, but this will still not address the need for observation of all lock events – information that is necessary for many analyses.

[4] The same observer also records initial probe logs for the observables requested when the program is instrumented prior to execution.

```
public final class InstrumentationManager {
  public void enableContructorEntryEvent(String key, String className,
        Type[] argTypes, boolean synchronous) { ... }

  public void disableConstructorEntryEvent(String key, String className,
        Type[] argTypes, boolean synchronous) { ... }

  public void enableContructorExitEvent(String key, String className,
        Type[] argTypes, boolean synchronous) { ... }

  public void disableConstructorExitEvent(String key, String className,
        Type[] argTypes, boolean synchronous) { ... }

  public void enableVirtualMethodEntryEvent(String key, String className,
      String methodName, Type returnType, Type[] argTypes,
      boolean synchronous) { ... }

  public void disableVirtualMethodEntryEvent(String key, String className,
      String methodName, Type returnType, Type[] argTypes,
      oolean synchronous) { ... }

  public void enableVirtualMethodExitEvent(String key, String className,
        String methodName, Type returnType, Type[] argTypes,
        boolean synchronous) { ... }

  public void disableVirtualMethodExitEvent(String key, String className,
        String methodName, Type returnType, Type[] argTypes,
        boolean synchronous) { ... }

  public boolean enableInstanceFieldAccessEvent(String key,
        String fieldName) { ... }

  public boolean disableInstanceFieldAccessEvent(String key,
        String fieldName) { ... }

  ...

  public void updateInstrumentation() { ... }
}
```

Figure 4: Sofya API (excerpt)

this `InstrumentationManager` API via a callback from the event dispatcher.[5] The JDI provides a function to redefine classes in a managed virtual machine, and as of Java 1.5 it is possible to to redefine classes from within the running virtual machine. The adaptive instrumentation API uses these features to add and remove instrumentation using the `Sofya` instrumentors, and the parts of the framework employed by the analyses discussed in this paper use the "redefineClasses" function of the JDI to swap in modified byte codes at runtime. A significant feature of Sofya's adaptive instrumentation API is that requests can be aggregated before redefinition occurs. This optimizes the use of the JDI class redefinition facility for groups of updates that affect the same class but different methods. Figure 6 illustrates the overall architecture of the adaptive analysis extension to the `Sofya` framework.

To illustrate how the API is used, in Figure 5 we sketch part of the implementation of an adaptive checker for the object protocol presented in Section 2; we abbreviate the names of API methods in our presentation. The analysis, which we refer to as A, is a factory (`FileProtocolMonitor.MonitorFactory`) attached to an object based splitter that produces an FSA checker, C (`FileProtocolMonitor`), for each instance of the `File` type allocated during

[5] A callback is employed because the adaptive event manager requires a connection to the virtual machine running the observed program. The callback is issued after the virtual machine for the target program has been launched and the connection has been established.

```
class FileProtocalChecker {
    public static void main(String[] args) {
        SemanticEventDispatcher dispatcher = new SemanticEventDispatcher();
        ObjectBasedSplitter objSplitter = new ObjectBasedSplitter(
            new FileProtocolMonitor.MonitorFactory());
        dispatcher.addEventListener(objSplitter);

        ...
    }
}

class FileProtocolMonitor {
    ...

    static class MonitorFactory implements ChainedEventListenerFactory {
        // Invoked when File constructor executes
        public EventListener createEventListener(ChainedEventListener parent,
                long streamId, String streamName) {
            return new FileProtocolMonitor();
        }
    }

    public void executionStarted() {
        iMgr.enableConstructorEntryEvent("fp-check", "File",
            new Type[]{ Type.STRING }, false);
        iMgr.updateInstrumentation();
    }

    public void virtualMethodEnterEvent(ThreadData threadData, ObjectData od,
            MethodData methodData) {
        String methodName = methodData.getSignature().getMethodName();
        if ("open".equals(methodName)) {
            if (state == CLOSED) {
                iMgr.disableVirtualMethodEntryEvent("fp-check", "File", "read",
                    Type.CHAR, new Type[0], false);
                iMgr.disableVirtualMethodEntryEvent("fp-check", "File", "write",
                    Type.VOID, new Type[]{ Type.CHAR }, false);
                iMgr.disableVirtualMethodEntryEvent("fp-check", "File", "eof",
                    Type.BOOLEAN, new Type[0], false);
                iMgr.updateInstrumentation();
            }
            else if (state == OPEN)
                error();
        }
        else if ("read".equals(methodName)) {
            if (state == CLOSED)
                error();
        }
        else if ("write".equals(methodName)) {
            if (state == CLOSED)
                error();
        }
        else if ("eof".equals(methodName)) {
            if (state == CLOSED)
                error();
        }
        else if ("close".equals(methodName)) {
            if (state == OPEN) {
                iMgr.enableVirtualMethodEntryEvent("fp-check", "File", "read",
                    Type.CHAR, new Type[0], false);
                // Enable 'write' and 'eof' events
                ...
                iMgr.updateInstrumentation();
            }
            else if (state == CLOSED)
                error();
        }
    }

    ...
}
```

Figure 5: Excerpt of File Protocol Checker Code

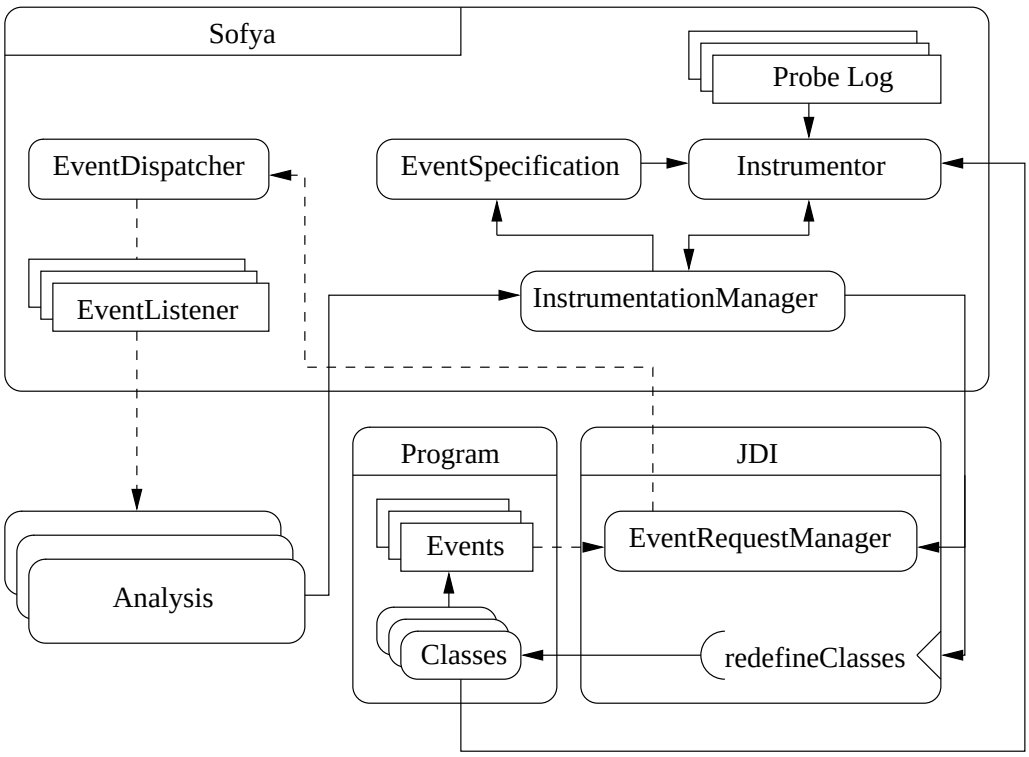

Figure 6: Sofya Adaptive Instrumentation Architecture

the program run. At the beginning of execution, A calls `enable("File", Type.STRING); update()` to enable instrumentation on `File` constructor calls. When that constructor is called, `Sofya` triggers the creation of an instance of C and attaches it to the event stream for newly allocated instance of `File`. The constructor of C will make calls (not shown) to enable the *outgoing* transition events out of the start state state, i.e., **1** of the protocol, specifically: `enable("File", "open"); enable("File", "read"); ... update()`. When `open` is called, C is triggered and disables the *self-loop* events for state **2** with calls: `disable("File", "read"); ... update()`. Finally, when `close` is called, C re-enables the events it disabled after the `open` was observed.

4 Adaptive Program Analyses

Researchers have investigated the use of a wide variety of formalisms for expressing properties of program executions that can be monitored at run-time. Assertions are now widely used during development [18, 28] and guidelines for safety critical software propose that assertions remain enabled during system operation [17]. Developers are clearly seeing the added error detection value of embedding non-functional validation code into their systems. In this section, we describe how adaptive analysis can reduce the overhead of *checking* stateful specifications of correct program behavior, to enrich online program validation techniques beyond the simple boolean expression evaluation supported by assertions.

4.1 FSA Checking

A variety of program properties can be expressed as finite-state machines or regular expressions, e.g., [12]. Developers define properties in terms of *observations* of a program's run-time behavior. In general, an observation may be defined in terms of a change in the data state of a program, the execution of a particular statement or class of statements, or some combination of the two. For simplicity, in our presentation we only consider observations that correspond to the entry and exit of a designated program method.

We define an *observable alphabet*, Σ, as a set of symbols that encode observations about the behavior of a program. A deterministic *finite state automaton* is a tuple $(S, \Sigma, \delta, s_0, A)$ where: S is a set of states, $s_0 \in S$ is the initial state, $A \subseteq S$ are the accepting states and $\delta | S \times \Sigma \rightarrow S$ is the state transition function. We use $\Delta | S \times \Sigma^+ \rightarrow S$ to define the composite state transition for a sequence of symbols from Σ.

Offline FSA checking involves instrumenting a program to detect each occurrence of an observable and to record the corresponding symbol in Σ, usually in a file. The sequence of symbols, $\sigma \in \Sigma^*$, describes a trace of a program execution that is relevant to the FSA property. An offline monitor will simply evaluate $\Delta(s_0, \sigma) \in A$ to determine whether the property is satisfied or violated by the program execution.

Online FSA checking involves instrumenting a program to detect each occurrence of an observable, but rather than record the corresponding symbol, $a \in \Sigma$, the analysis tracks the progress of the FSA in recognizing the sequence of symbols immediately. The analysis executes the algorithm sketched in Figure 7, where $s_{cur} \in S$ records the current state of the FSA. When the instrumentation for an observable executes, it triggers the execution of a handler, passing the id of the detected observable. An online monitor evaluates $s_{cur} \in A$ upon program exit to determine whether the property is satisfied or violated by the program execution. Online analysis has a clear space advantage since it need not record the program trace. In this simple setting, it appears as if the online approach also has a performance advantage, but for realistic situations where many FSAs may be monitored simultaneously the offline approach may be significantly faster.

4.2 Adaptive Online FSA Checking

Online FSA checking is made adaptive by defining precisely when observables can be safely ignored by the analysis. The instrumentation for those observables can be removed from from the program without affecting the ability of the checking algorithm to accept or reject the program run.

The *outgoing symbols* for a state, $s \in S$, are defined as

$$\mathit{out}(s) = \{a | a \in \Sigma \wedge \delta(s, a) \neq s\}$$

and the *self symbols* as

$$\mathit{self}(s) = \Sigma - \mathit{out}(s)$$

The intuition is that we wish to define a subset of the alphabet that forces the FSA to leave the given state, and the rest of the symbols can be ignored in that state. Consider a trace $\sigma = \sigma_0 + a + \sigma_1$ over the alphabet, where $+$ denotes concatenation. For an FSA with $s = \Delta(s_0, \sigma_0)$ and $a \in self(s)$, by definition, $\Delta(s, a) = s$, and thus, $\Delta(s_0, \sigma_0 + \sigma_1) = \Delta(s_0, \sigma)$.

Adaptive online FSA checking is an extension to online dynamic FSA checking. Figure 8 sketches the algorithm for the analysis. Initially, program instrumentation for outgoing observables in the FSA start state are enabled. When

```
INIT()                          HANDLER(oid a)
 1  s_cur = s_0                  2  s_cur = δ(s_cur, a)
end INIT()                      end HANDLER()
```

Figure 7: Online FSA monitor

```
INIT()                          HANDLER(oid a)
 1  s_cur = s_0                  3  s_next = δ(s_cur, a)
 2  enable(out(s_cur))           4  enable(out(s_next) − out(s_cur))
end INIT()                      5  disable(self(s_next) − self(s_cur))
                                6  s_cur = s_next
                                end HANDLER()
```

Figure 8: Adaptive online FSA monitor

an observable $a \in \Sigma$ occurs, the analysis updates the current state to be the next state, and enables and disables the appropriate instrumentation for that state. This guarantees that all instrumentation needed for transitioning out of the next state is enabled, which is all that is needed to assure equivalence with online dynamic FSA checking. Disabling or failing to disable instrumentation for self symbols does not impact correctness, only performance.

4.3 Checking Multiple Properties

Our adaptive analyses are capable of checking FSA conformance over the lifetimes of independent objects during program execution, i.e., they are object-sensitive. Consequently they are built using the *object based splitters*, discussed in Section 3, that trigger the creation of a copy of the FSA checker for each allocated object involved in the property. The handlers for those checkers are invoked only when an observable occurs on the specific object they are listening for.

Typical applications of FSA checking will check program conformance with a set of property specifications. Users may choose to sample from a large set of properties to mitigate run-time overhead, but even the presence of two FSA monitors active in the same adaptive analysis can cause interference when one monitor disables an observable that another requires. This would lead to incorrect analysis results relative to a non-adaptive analysis.

We illustrate a solution based on reference counting of instrumented observables by enhancing the algorithm in Figure 8 to obtain that of Figure 9. The concept is simple: reference counts are maintained that reflect the number of FSA monitors that require a given observable, and that observable is only disabled when the reference count reaches zero (lines 9-12 in Figure 9). An observable is enabled (lines 2-4 and 6-8 of Figure 9) when the first monitor requests it after it has been disabled. A more detailed excerpt from the actual implementation of the adaptive object-sensitive FSA checker is given in Figure 12.

When multiple properties are analyzed simultaneously, the *global* alphabet of all observable symbols is the union of the individual property alphabets. This requires each property FSA be transformed to be defined over this global alphabet and to introduce self-loop transitions in each state for all symbols in $\Sigma_{global} - \Sigma_{property}$, though if the FSA alphabets are disjoint this alphabet *inflation* is unnecessary. We note that this changes neither the meaning of the property, nor the outgoing symbol set definitions. Maintaining small outgoing symbol sets is a key factor in reducing the cost of adaptive analysis.

```
INIT()
 1  s_cur = s_0
 2  for each a ∈ out(s_cur do
 3      if (count[a]++) == 1 then
 4          enable(a)
end INIT()

 5  s_next = δ(s_cur, a)
 6  for each b ∈ out(s_next) − out(s_cur)) do
 7      if (count[b]++) == 1 then
 8          enable(b)
 9  for each b ∈ self(s_next) − self(s_cur)) do
10      count[b]−=(count[b] > 0)?1 : 0
11      if count[b] == 0 then
12          disable(b)
13  s_cur = s_next
end HANDLER()
```

Figure 9: Adaptive online multi-FSA monitor

4.4 Experience and Evaluation

We have implemented a family of online dynamic FSA checking analyses for Java using `Sofya`. The analyses vary in the degree of object and thread sensitivity they enforce, and in whether they are adaptive or not, but all accept properties specified as regular expressions and then convert those to deterministic FSAs using the Laser FSA toolkit from the University of Massachusetts.

Figures 10 and 11 show examples of the kind of regular expression that our analyses accept as input. On the top of Figure 10 is an instance of a *precedence* specification pattern [12] defined over calls on a Java interface where the `~` operator negates symbol classes, denoted by `[ ]`, `.` denotes all events, and `;` denotes concatenation. We refer to this property as **SetReader Before Parse** (sbp); to conserve space we have elided the JNI strings used to define calls based on signatures in all but this property.

On the top of Figure 11 is a more complicated *constrained-response* pattern instance that requires the cyclic occurrence of `setBuilder` and `getResult` calls; we refer to this property as **Parser Builder** (pb). These specifications have named parameters, e.g., `p` and `b`, and *wild cards*, $\star$, that are used to correlate calls on related objects. For instance, the unique object id of the `IXMLBuilder` instance passed to the `setBuilder` call is bound to `b`. Subsequently, the expression only matches calls on the API with `b` as the receiver object. Specifications with parameters are matched against the program trace in a manner that is very similar to parametric regular expressions [24] where special support in the checker is used to bind values to object instances.

We wrote two precedence properties and two constrained-response properties that captured expected usage patterns of the NanoXML library; we acquired this program from the SIR repository [10, 11]. NanoXML is an open-source library for parsing XML structures that is designed to have a simple API and be very light-weight. We checked the properties on two of the applications that come with the NanoXML release: `XML2HTML` converts an XML file written using a specific DTD to HTML, `JXML2SQL` translates an XML file conforming to a particular DTD into SQL commands to construct a database and populate it with tables.

Table 2 provides some basic static measures of the programs and of NanoXML itself. Since we focused on the use of the three primary interfaces in NanoXML, i.e., `IXMLParser`, `IXMLBuilder`, and `IXMLReader`, it will come as little surprise that we observed a similar pattern of library usage by the applications. There were, however, some interesting differences between the applications that demanded the accurate object tracking and correlation implemented in our analyses. For example, `XML2HTML` used a custom instance of an `IXMLBuilder` whereas `JXML2SQL` used a default builder, and the applications used different NanoXML factory methods to create and bind instances of the

```
for events {
  "IXMLParser:parse:()Ljava/lang/Object;",
  "IXMLParser:setReader(Lnet/n3/nanoxml/IXMLReader)V"
}

~["IXMLParser:parse:()Ljava/lang/Object;"]*
|
(
  ~["IXMLParser:parse:()Ljava/lang/Object;",
    "IXMLParser:setReader(Lnet/n3/nanoxml/IXMLReader)V"]*;
  "IXMLParser:setReader(Lnet/n3/nanoxml/IXMLReader)V";
  .*
)


for events {
  "IXMLBuilder:startBuilding", "IXMLBuilder:startElement",
  "IXMLBuilder:addAttribute", "IXMLBuilder:addPCData"
}

~["startElement", "addAttribute", "addPCData"]*
|
(
  ~["startElement", "addAttribute", "addPCData"]*
  "startBuilding";
  .*
)
```

Figure 10: API Call Precedence : sbp (top) and sbbsa (bottom)

Code base	Classes	Public Methods	SLOC
NanoXML	25	247	1908
XML2HTML	5	-	109
JXML2SQL	10	-	353

Table 2: Application code-base measures

three interfaces.

The NanoXML and application code bases are small, but the complexity of a dynamic analysis is dependent on the run-time behavior of the program, the inputs to the program, and of course the source code; even a small program can have complex and long-running behavior.

A few small XML sample input files were supplied with the applications, but based on an informal survey of XML files available on the Internet, which ranged from 10 to several hundred kilobytes in size, we felt that the sample inputs would not force the complexity and duration of run-time behavior that an XML parser would experience in real world conditions. To address this, we constructed a program that would generate XML files of increasing size that complied with the two applications' DTDs; the content of the files was generated by constructing XML structures around a sampling from the 311,142 words in the standard Linux dictionary and "extra words" files.

We ran several different analyses for each application and each property specification on increasing sizes of input. **noinst** is a completely un-instrumented version of the application; we use it to assess the overhead of our analyses. **jrat**

```
for events {
  "IXMLParser.parse", "IXMLParser.setBuilder", "IXMLBuilder.startElement",
  "IXMLBuilder.addAttribute", "IXMLBuilder.addPCData", "IXMLBuilder.getResult"
}

~["*.setBuilder(*)"]*;
( "p.setBuilder(b)";
  ~["p.setBuilder(*)", "b.startElement", "b.addAttribute", "b.addPCData", "b.getResult"]*;
  "p.parse";
  ~["p.setBuilder(*)", "p.parse"]*;
  "getResult";
  ~["p.setBuilder(*)", "b.startElement", "b.addAttribute", "b.addPCData", "b.getResult"]*
)*


for events {
  "IXMLParser.parse[call]", "IXMLParser.parse[return]", "IXMLParser.setReader",
  "IXMLReader.read"
}

~["*.setReader(*)"]*;
( "p.setReader(r)";
  ~["p.setReader(*)", "r.read"]*;
  "p.parse[call]";
  ~["p.setReader(*)", "p.parse[call]"]*;
  "p.parse[return]";
  ~["p.setReader(*)", "p.parse[call]", "r.read"]*
)*
```

Figure 11: API Constrained-response Properties : pb (top) and pr (bottom)

is an instrumentation framework that uses BCEL to capture trace data from Java programs that is used by a number of researchers; we implemented an optimized handler for recording just the set of observations present in a property as described in [36]. **adaptive** is our adaptive FSA checking analysis. We also ran a non-adaptive version of our FSA checking analysis, but we do not report on its performance since it was significantly slower than the others (or several examples, we observed that the cost increased at a rate that was more than twice that of **jrat**). Each combination of analysis, application and input size was run 10 times on a dual Opteron 252 (2.6Ghz) SMP system running Gentoo Linux 2006.0 and JDK 1.5_08; we instrumented the program under analysis to measure time spent between the start and end of execution of the analyzed application.

In general, we observed very similar trends in performance across the two applications. This is not surprising, since they are both performing XML parsing using NanoXML, and then applying some additional custom computation on an internal representation of the parsed data. The performance of these applications is dominated by the time to perform the XML parsing, which causes the overhead of checking NanoXML APIs to appear larger than it would for applications that performed significant add:qitional computation.

Table 4 reports the time cost, at the 6th data point, of different analysis techniques for pairs of application and property. In addition to the "pb" and "sbp" properties described above, we check a precedence property for `IXMLBuilder` instances, called **SetBuilder Before StartElement AddAttribute** (sbbsa), and a constrained-response property relating `IXMLReader` and `IXMLParser`, called **Parser Reader** (pr). These two properties are shown in Figure 10 and

```
class AdaptiveObjectSensitiveFSAMonitor {
  ...

  public AdaptiveObjectSensitiveFSAMonitor(
      RunnableFSAInterface<StringLabel> fsa,
      HashMap<FSAStateInterface<StringLabel>,
      HashSet<StringLabel>> selfLoopSymbolsMap,
      HashMap<FSAStateInterface<StringLabel>,
      HashSet<StringLabel>> progressSymbolsMap,
      HashMap<StringLabel, String[]> eventStringMap,
      HashMap<StringLabel, Integer> eventIndexMap,
      ResultCollector results) {
    this.fsa = fsa;
    ...
    this.currentState = fsa.getStartState();
    this.alphabet = fsa.getAlphabet();

    if (initialize) {
      initialize = false;
      activeSymbolCounts = new int[alphabet.size()];
    }

    for (StringLabel pSym :
           progressSymbolsMap.get(currentState)) {
      int pSymCount = activeSymbolCounts[
            eventIndexMap.get(pSym).intValue()]++;
      if (pSymCount == 0) {
        String[] es = eventStringMap.get(pSym);
        instMgr.enableVirtualMethodEntryEvent(
                  null, es[0], es[1], es[2], true);
      }
    }
    instMgr.updateInstrumentation();
  }

  public void virtualMethodEnterEvent(
      ThreadData threadData, ObjectData od,
      MethodData methodData) {
    String className =
        methodData.getSignature().getClassName();
    String methodName =
        methodData.getSignature().getMethodName();
    String signatureString =
        methodData.getSignature().getTypeSignature();

    StringLabel sl = alphabet.createLabelInterface(
       className + ":" + methodName + ":" +
       signatureString);

    if (!alphabet.contains(sl)) return;

    SortedSet<FSAStateInterface<StringLabel>> succs =
        fsa.getSuccessorStates(currentState, sl);

    FSAStateInterface<StringLabel> prevState =
        currentState;
    currentState = succs.first();

    if (prevState != currentState) {
      HashSet<StringLabel> newProgressSymbols =
          (HashSet<StringLabel>) (progressSymbolsMap
              .get(currentState).clone());
      newProgressSymbols.removeAll(
          progressSymbolsMap.get(prevState));

      for (StringLabel pSym : newProgressSymbols) {
        int pSymCount = activeSymbolCounts[
            eventIndexMap.get(pSym).intValue()]++;
        if (pSymCount == 0) {
          String[] es = eventStringMap.get(pSym);
          instMgr.enableVirtualMethodEntryEvent(
              null, es[0], es[1], es[2], true);
        }
      }

      HashSet<StringLabel> newSelfLoopSymbols =
          (HashSet<StringLabel>) (selfLoopSymbolsMap
              .get(currentState).clone());
      newSelfLoopSymbols.removeAll(
          selfLoopSymbolsMap.get(prevState));

      for (StringLabel slSym : newSelfLoopSymbols) {
        if (activeSymbolCounts[eventIndexMap.get(
              slSym).intValue()] > 0) {
          int slSymCount = activeSymbolCounts[
                eventIndexMap.get(slSym)
                    .intValue()]--;
          if (slSymCount == 1) {
            String[] es = eventStringMap.get(slSym);
            instMgr.disableVirtualMethodEntryEvent(
                null, es[0], es[1], es[2], true);
          }
        }
      }

      instMgr.updateInstrumentation();
    }
  }
  ...
}
```

Figure 12: Excerpt of Adaptive Object-sensitive FSA Checker Code

11, respectively. These data clearly show that adaptive FSA checking can be performed with relatively low overhead compared to the un-instrumented application.

Measurements of overhead are useful, but they only characterize analysis performance at single points in the range of behaviors of the program under analysis. To get a more complete picture of analysis behavior, in Figure 13 we plot the rates of growth of the analysis costs, as input size increases, for each of the properties analyzed on one of the applications; the curves for the other applications are similar. Two prominent trends are apparent in the data. (1) Adaptive analysis almost never performed worse than **jrat**. For a few small input sizes of the *setReader Before parse* precedence property **jrat** is faster, but this is a property that observes two API calls, each of which occurs a single time in each application, so the overall burden of checking is limited to processing two observations; it is noteworthy that because of the structure of the precedence FSA, the adaptive analysis need only process a single observation. The performance advantage of the adaptive analysis is an underestimate, since an offline dynamic analysis would incur

app-property-technique	size (kb)									
	8.2	16.4	24.5	32.7	40.8	49.0	57.1	65.2	73.4	81.4
	89.5	97.6	105.7	113.8	121.9	130.1	138.3	146.5	154.6	162.8
XML2HTML-noinst	0.06	0.10	0.13	0.17	0.19	0.24	0.28	0.34	0.40	0.40
	0.41	0.44	0.47	0.48	0.53	0.50	0.54	0.57	0.58	0.57
XML2HTML-pb	2.67	4.34	5.71	6.78	7.94	9.20	10.31	11.38	12.58	13.64
	14.59	15.70	16.90	17.83	18.78	19.94	21.02	22.07	23.02	24.10
XML2HTML-apb	1.39	1.45	1.52	1.59	1.63	1.62	1.65	1.71	1.73	1.74
	1.81	1.78	1.83	1.89	1.87	1.91	1.90	1.95	1.91	1.98
	1.0	1.9	2.8	3.6	4.5	5.3	6.2	7.0	7.9	8.7
	9.6	10.5	11.3	12.2	13.0	13.9	14.8	15.7	16.6	17.4
XML2HTML-noinst	0.02	0.03	0.04	0.04	0.05	0.05	0.06	0.06	0.06	0.07
	0.07	0.08	0.08	0.08	0.09	0.09	0.10	0.10	0.10	0.10
XML2HTML-pr	3.28	4.73	6.01	7.31	8.52	9.54	10.68	12.07	12.96	13.86
	15.13	16.33	17.21	18.34	19.44	20.60	21.45	22.64	23.68	24.48
XML2HTML-apr	1.31	1.34	1.33	1.35	1.37	1.36	1.40	1.38	1.42	1.40
	1.36	1.42	1.40	1.49	1.43	1.44	1.53	1.50	1.52	1.49
	874.8	1745.8	2623.9	3499.1	4382.0	5260.5	6137.6	7015.6	7899.4	8785.5
	9666.7	10550.1	11429.1	12310.3	13188.3	14065.0	14949.3	15839.6	16712.3	17590.3
XML2HTML-noinst	1.43	2.02	2.86	3.53	4.20	5.50	6.17	6.70	7.57	8.03
	9.64	10.33	10.91	11.53	12.26	13.03	13.27	16.11	16.74	17.50
XML2HTML-sbp	2.20	3.32	4.25	5.15	6.04	7.17	7.83	8.68	9.83	10.75
	11.57	12.32	13.12	13.95	15.16	16.52	17.38	18.04	18.89	19.59
XML2HTML-asbp	2.54	3.56	4.43	5.43	6.25	7.41	8.18	8.94	10.28	11.15
	11.86	12.69	13.50	14.26	15.59	16.58	17.45	18.27	19.10	19.83
	10.9	21.8	32.7	43.5	54.4	65.2	76.0	86.8	97.6	108.4
	119.2	130.1	141.0	151.9	162.8	174.0	184.8	196.0	207.4	218.3
XML2HTML-noinst	0.08	0.12	0.16	0.21	0.28	0.34	0.38	0.40	0.47	0.51
	0.48	0.54	0.54	0.54	0.60	0.60	0.60	0.66	0.67	0.70
XML2HTML-sbbsa	2.75	4.42	5.84	7.05	8.24	9.42	10.57	11.62	12.89	13.88
	15.20	16.15	17.29	18.44	19.27	20.54	21.50	22.64	23.96	24.85
XML2HTML-asbbsa	0.93	1.01	1.08	1.14	1.15	1.19	1.25	1.24	1.34	1.37
	1.37	1.44	1.44	1.51	1.48	1.53	1.58	1.60	1.61	1.60
	13.0	25.4	37.6	50.0	62.4	74.8	87.3	99.7	112.5	125.3
	137.7	150.4	163.0	175.6	188.3	200.8	213.2	225.8	238.5	251.1
JXML2SQL-noinst	0.16	0.26	0.38	0.46	0.59	0.72	0.95	1.13	1.30	1.42
	1.54	1.96	2.08	2.42	2.82	3.12	3.27	3.40	4.19	4.40
JXML2SQL-pb	2.23	3.56	4.71	5.73	6.83	7.85	8.84	9.99	11.37	12.35
	13.60	15.03	16.28	17.23	18.86	20.12	21.77	22.81	24.49	25.70
JXML2SQL-apb	1.66	1.76	1.90	1.97	2.17	2.39	2.58	2.87	3.37	3.58
	3.86	4.50	4.88	5.36	5.95	6.52	6.75	8.10	8.40	9.22
	2.2	3.8	5.5	7.2	8.9	10.5	12.2	13.8	15.5	17.2
	18.9	20.5	22.1	23.8	25.4					
JXML2SQL-noinst	0.07	0.07	0.08	0.09	0.12	0.13	0.13	0.17	0.16	0.17
	0.18	0.20	0.25	0.25	0.27					
JXML2SQL-pr	4.56	6.87	8.80	10.95	12.68	14.69	16.46	18.00	20.05	21.47
	23.47	25.30	27.21	28.92	30.49					
JXML2SQL-apr	1.40	1.44	1.49	1.51	1.47	1.45	1.56	1.53	1.60	1.63
	1.63	1.64	1.72	1.66	1.68					
	21.3	41.8	62.4	83.2	103.9	125.3	146.3	167.2	188.3	209.0
	230.0	251.1	272.3	293.1	313.9	334.8	355.5	376.5	397.5	418.3
JXML2SQL-noinst	0.21	0.40	0.61	0.89	1.12	1.47	1.83	2.18	2.66	3.02
	3.71	4.36	4.79	5.91	6.44	8.43	8.79	9.53	12.40	12.12
JXML2SQL-sbp	0.86	1.04	1.33	1.83	2.18	2.79	3.55	4.17	5.13	6.07
	7.02	8.10	9.40	10.63	11.57	13.72	14.99	17.15	18.36	21.46
JXML2SQL-asbp	1.42	1.37	1.63	2.06	2.47	2.97	3.82	4.50	5.34	6.29
	7.01	8.36	9.88	11.08	12.09	13.87	14.84	17.06	19.26	20.55
	15.5	30.3	45.1	59.9	74.8	89.8	104.8	120.3	135.2	150.4
	165.5	180.7	195.8	210.7	225.8	241.0	256.3	271.5	286.5	301.4
JXML2SQL-noinst	0.16	0.30	0.42	0.54	0.75	1.00	1.22	1.39	1.61	1.91
	2.19	2.54	3.04	3.34	3.50	4.00	4.48	4.92	5.68	5.76
JXML2SQL-sbbsa	1.78	2.85	3.87	4.78	5.88	6.90	7.79	8.88	10.03	11.11
	12.32	13.49	14.81	16.05	17.51	18.78	20.33	22.30	23.58	25.92
JXML2SQL-asbbsa	1.11	1.21	1.35	1.59	1.79	2.19	2.40	2.95	3.45	3.91
	4.41	5.26	5.70	6.28	7.08	7.72	8.56	9.67	10.64	11.64

Table 3: Scaled Timings (seconds)

app-property-technique	1746 (kb)	3499	5261	7016	8786	10550	12310	14065	15840	17590
XML2HTML-noinst	1.9	3.5	5.4	6.7	8.1	10.5	11.8	12.9	16.1	17.4
XML2HTML-pb-jrat	5.9	11.8	18.8	22.9	30.7	36.6	43.2	46.7	55.4	59.5
XML2HTML-pb-adaptive	4.1	5.9	7.9	9.3	11.6	13.2	14.7	17.3	18.7	20.4
XML2HTML-pr-jrat	48.8	93.7	142.1	185.2	234.5	284.5	322.0	377.0	414.0	468.8
XML2HTML-pr-adaptive	4.1	5.8	7.9	9.4	11.6	13.1	14.6	17.2	18.7	20.3
XML2HTML-sbp-jrat	3.0	4.3	6.9	9.6	10.5	13.7	16.4	17.2	22.1	22.6
XML2HTML-sbp-adaptive	3.9	5.6	7.7	9.1	11.5	13.0	14.5	17.0	18.5	20.1
XML2HTML-sbbsa-jrat	5.3	9.7	15.4	19.2	23.7	28.7	35.0	40.1	41.5	48.5
XML2HTML-sbbsa-adaptive	3.8	5.5	7.7	9.2	11.4	12.9	14.4	17.0	18.7	20.1
JXML2SQL-pb-noinst	2.6	5.1	6.5	8.3	11.3	12.6	16.8	18.7	20.9	23.5
JXML2SQL-pb-jrat	5.6	10.0	15.6	19.0	25.0	29.4	33.3	42.8	45.1	49.7
JXML2SQL-pb-adaptive	4.9	7.3	9.7	12.1	14.4	16.8	19.3	21.7	25.5	31.2
JXML2SQL-pr-jrat	50.0	96.1	152.7	197.6	250.6	298.5	339.0	400.9	442.9	500.4
JXML2SQL-pr-adaptive	4.7	7.2	9.7	12.1	14.4	16.5	19.5	21.6	25.5	31.1
JXML2SQL-sbp-jrat	3.4	6.5	7.7	10.3	14.3	15.9	17.6	21.9	23.9	24.4
JXML2SQL-sbp-adaptive	4.5	7.0	9.3	11.9	14.1	16.3	19.0	21.3	24.6	30.8
JXML2SQL-sbbsa-jrat	4.8	8.3	11.9	15.4	20.0	24.9	25.2	32.8	36.1	40.4
JXML2SQL-sbbsa-adaptive	4.6	7.2	9.5	11.8	14.3	16.6	19.5	21.3	24.9	31.2

Table 4: Uniform Incremental Timings (seconds)

the cost of the **jrat** execution, to record a trace file, and then additional cost to process the trace file. Furthermore, for some of the larger input sizes, the **jrat** analysis generates trace files of several gigabytes, whereas the adaptive analysis, as an online checker, simply delivers the boolean analysis verdict. (2) Adaptive analysis appears to have a similar rate of growth as the un-instrumented program. Clearly there is some initial startup overhead incurred by adaptive analysis, but the gap in performance does not widen as the program input size increases. This bodes well for considering adaptive analyses as candidates to be deployed in fielded systems, since their overhead appears negligible once the system is initialized.

These results can only be considered preliminary evidence on the cost-effectiveness of adaptive online program analysis, but we believe they are a strong indicator that low-overhead dynamic analysis of stateful properties can be achieved.

5 Adaptive Online Property Inference

In recent years a number of researchers have proposed analyses to *infer* sequencing properties from program traces [2, 34, 33, 37, 36]. For example, [2] presents a general approach for learning arbitrary FSAs that characterize the sequencing of program observations; specifically, they infer calling sequences on APIs. This approach has proven problematic in part because it produces FSA specifications that are quite complicated; in essence it produces the product of all of the sequencing constraints that are realized by the client-API interactions on a specific program run. API documenters tend to express calling constraints in terms of smaller, more focused specifications. For these reasons, followup research has looked at the sequencing property inference problem as one of matching program behavior to instances of a families of specification patterns. For example, Weimer et. al [33] considers simple variants of an alternating pattern, i.e., $(AB)^*$ where A and B are bound to distinct program observations. Yang et. al consider the alternating pattern in addition to several other patterns [37, 36].

These analyses all operate offline; they collect a trace file over a set of observations and then process the trace file to

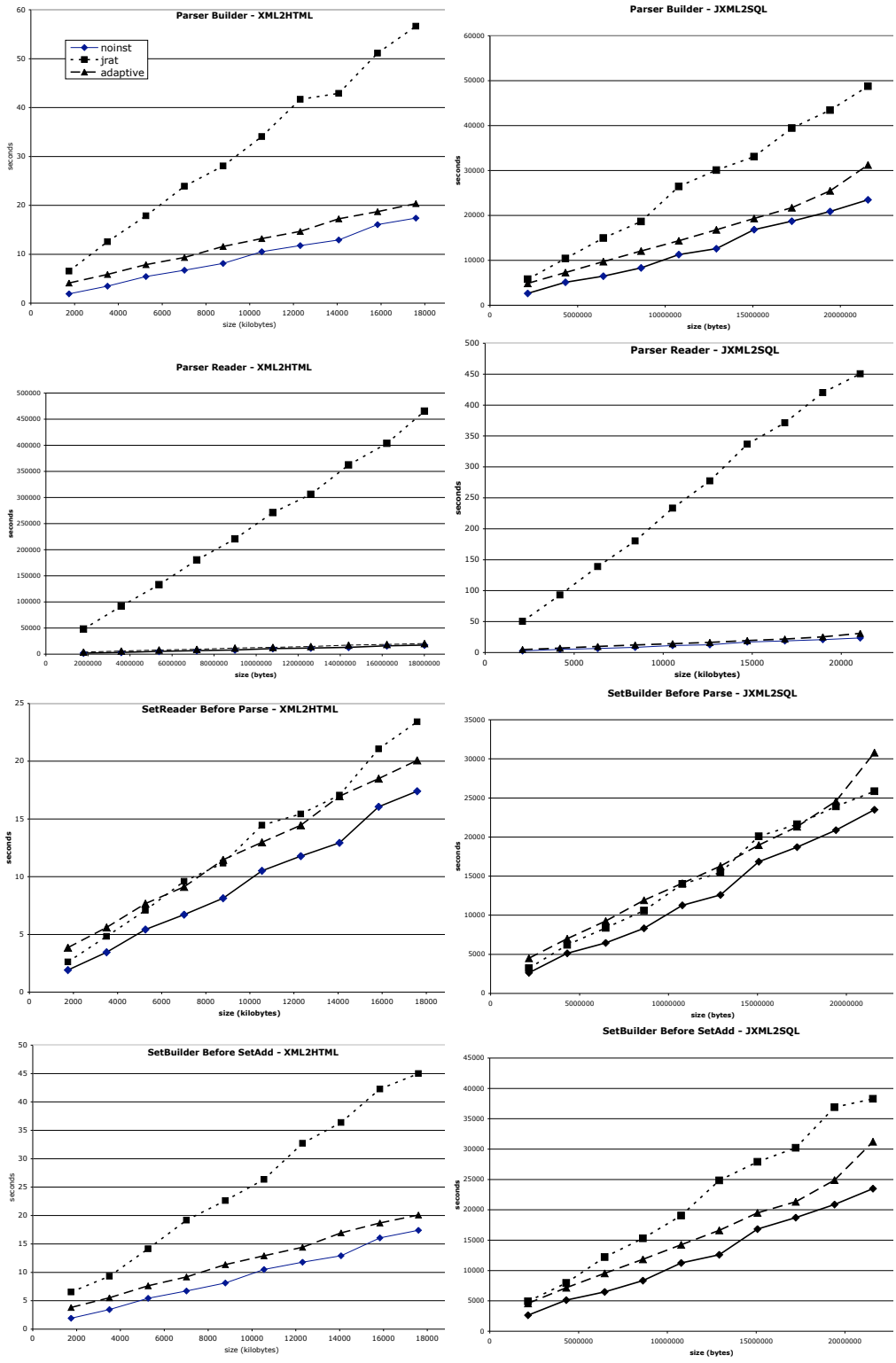

Figure 13: Growth of FSA checking analysis cost with XML file size

detect pattern matches. We are interested in exploring the potential of adaptive analysis in improving the performance of these analyses by making them online. Whereas for property checking we had access to reasonable implementations of analyses that define best-practice for offline analyses, for property inference we were unable to gain access to any of the above analyses. Consequently, our performance comparison is less mature than for property checking analyse.

In this section, we describe two adaptive property inference analyses. The analyses were designed and implemented independently by two of the authors of this paper. The first approach, which we call *eager* inference, generates a set of candidate pattern instances and collects evidence from program runs that either invalidate candidates during the run or confirm them at the end of the run. The second approach, which we call *lazy* inference, only accumulates positive evidence for the presence of a pattern instance; it invalidates pattern instances that have been detected as potential candidates earlier in the program run. These very different strategies for inference of alternating patterns independently have allowed us to make qualitative observations about the potential performance improvements that can be achieved through adaptive property inference.

5.1 Eager Adaptive Property Inference

Conceptually the analysis is very simple: it generates the set of all possible $(AB)^*$ regular expressions over the public calls in an API and launches simultaneous FSAs (as in Section 4.3) to perform online checks for those expressions. Figure 14 gives the generic structure of an FSA for this pattern, where A and B are bound to each pair of calls. On the face of it, it seems hopelessly inefficient to have so many online checkers running simultaneously. However, most of the FSAs are violated very early in processing a program trace and transition to their *sink* state. Recall that once an FSA reaches its *sink* state all transitions are self-loops. This results in a rapid convergence of observable reference counts towards zero, at which point instrumentation for the observable is turned off for the remainder of the analysis run.

Table 5 illustrates this process for the example in Figure 1 with a file of length 3, which produces a sequence of six observable events; we restrict the alphabet to `open` (o), `close` (c), and `eof` (e) to keep the example small. Six instances of the FSA from Figure 14 are operating simultaneously, making independent transitions into different states (represented in each cell) based on the sequence of observables; the AB bindings for the FSA are given in the first column of the table. When the program exits, the analysis produces the set of patterns, i.e., alternating pattern instances, that were not violated, which for this example is $(\texttt{open};\texttt{close})^*$. We note that after the third observable has occurred, all FSAs checking properties involving `eof` have transitioned to their sink states and the instrumentation for that observable is removed. Thus, property inference over this alphabet for this program will require 4 observable events, regardless of the size of the program input.

One significant advantage of this approach is that it is simple to adapt to mining other specification patterns. One need only describe a skeletal version of the pattern, and the analysis will generate the specific instances to check online; we used this feature to infer *precedence* patterns, like the ones discussed in Section 4, in addition to alternating patterns.

In general, for inferring properties of a class, the number of candidate pattern instances is b^k where b is the number of public methods defined in a class and k is the number of distinct *parameters* in the specification pattern. For NanoXML, the `IXMLParser`, `IXMLReader`, and `IXMLBuilder` interfaces have 9, 11, and 8 public methods,

AB	Observable Trace						
	o	e	e	e	e	c	Outcome
oc	2	2	2	2	2	1	$\surd$
co	s	-	-	-	-	-	$\times$
oe	2	1	s	-	-	-	$\times$
eo	s	-	-	-	-	-	$\times$
ec	1	2	s	-	-	-	$\times$
ce	1	s	-	-	-	-	$\times$

Table 5: File Trace and FSA Transitions

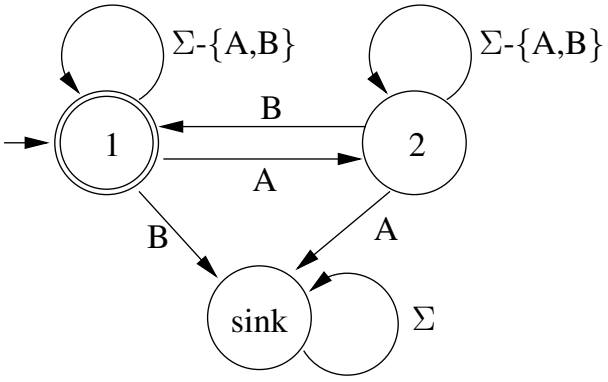

Figure 14: Generic $(AB)^*$ FSA

respectively. Consequently, precedence and alternating pattern inference required 72, 110, and 56 initial candidate patterns.

Like the example in Figure 5, when running inference on these APIs for `XML2HTML` and `JXML2SQL` we observed very rapid convergence to a small set of candidate patterns. All of the eager analyses ran in less than 2 minutes for relatively small input sizes, less than 10 kbytes, in part because large numbers of observable instrumentation was able to be disabled. Unlike the example in Figure 5, our inference analyses were unable to turn off all *periodic* observations, i.e., observations that occur repeatedly during a program run. For example, in the `IXMLBuilder` interface there are calls to `startElement` and `addAttribute` that occur in matching pairs for all elements in these applications. We believe this is due to the fact that the DTDs for these applications are quite simple and do not contain multiple attributes per XML element - giving the illusion that the calls occur in an alternating fashion. As another example, in the same interface `startElement` and `endElement` also occur in alternating fashion in these applications. This is due to the fact that the XML element nesting in the inputs we considered was trivial. For nested elements one would see consecutive `startElement` calls as the parser descends through the document, but for documents of one level deep, again, there is the illusion that calls to start and end elements alternate.

Clearly we need to study the performance of our adaptive analyses more broadly across a range of APIs, applications and inputs. Nevertheless, the eager analyses inferred nearly all of the properties we expected after reading API documentation. One property that we expected to infer was invalidated by `JXML2SQL`. Upon further investigation we discovered that this application was actually misusing the NanoXML APIs, at least with respect to the documented intended usage of the API. We modified the code and were able to infer the expected property.

5.2 Lazy Adaptive Property Inference

Our second approach avoids the initial cost of generating an exponential number of candidates, by inferring positive evidence from the trace to generate candidates, and then invalidating those candidates in subsequent monitoring where appropriate.

The logic of this analysis is quite tricky and is explained in Algorithm 1. It is worth noting that all of the special cases in this algorithm, e.g., handling the start and end of program traces, are handled by the eager approach without any modification. Since this algorithm avoids constructing an initial set of candidates, it is no surprise that it is significantly cheaper for small program inputs. For the examples described above all of the analyses terminated in less than 2 seconds and calculated the same results.

Algorithm 1 Lazy AB miner pseudocode

```
{s = current symbol}
{map liveAsAOpen = symbol → set of candidate B symbols; new Bs can still be added}
{map liveAsAClosed = symbol → set of candidate B symbols; no new Bs can be added}
{map liveAsB = symbol → set of candidate A symbols; initialized by prefix at point when symbol is first seen; can only be reduced}
{set seenBefore = tracks all previously seen symbols}

if s is end of string then
  for all symbol t in keys(liveAsAOpen) do
    for all symbol u in liveAsAOpen(t) do
      record 'AB' candidate tu
    end for
  end for
  for all symbol t in keys(liveAsAClosed) do
    for all symbol u in liveAsAClosed(t) do
      if u is marked as retained then
        record 'AB' candidate tu
      end if
    end for
  end for
end if
if s not in seenBefore then
  add s to seenBefore
  create empty liveAsB(s)
  for all symbol t in keys(liveAsAOpen) do
    add s to liveAsAOpen(t)
    add t to liveAsB(s)
  end for
  if liveAsB(s) is empty then
    remove liveAsB(s)
  end if
  create empty liveAsAOpen(s)
else
  {s seen before}
  if s is in keys(liveAsAClosed) then
    for all symbol t in liveAsAClosed(s) do
      if t is not marked as retained then
        remove t from liveAsClosed(s)
        remove s from liveAsB(t)
        if liveAsB(t) is empty then
          remove liveAsB(t)
          if t not in keys(liveAsAClosed) then
            remove instrumentation for t
          end if
        end if
      else
        clear retained mark on t
      end if
    end for
    if liveAsAClosed(s) is empty then
      remove liveAsAClosed(s)
    end if
  else if s in keys(liveAsAOpen(s)) then
    if liveAsAClosed(s) is not empty then
      move liveAsAOpen(s) to liveAsAClosed(s)
    end if
  end if
  for all symbol t in keys(liveAsAOpen) do
    if s in liveAsAOpen(t) then
      remove s from liveAsAOpen(t) {'ABB'}
      remove t from liveAsB(s)
      if liveAsB(s) is empty then
        remove liveAsB(s)
      end if
    end if
  end for
  for all symbol t in {keys(liveAsAClosed) - s} do
    if s in liveAsAClosed(t) then
      if s is marked as retained then
        remove s from liveAsAClosed(t) {'ABB'}
        remove t from liveAsB(s)
        if liveAsAClosed(t) is empty then
          remove liveAsAClosed(t)
          if t not in keys(liveAsB) then
            remove instrumentation for t
          end if
        end if
        if liveAsB(s) is empty then
          remove liveAsB(s)
        end if
      else
        mark s as retained in liveAsAClosed(t)
      end if
    end if
  end for
  if s not in keys(liveAsAClosed) and s not in keys(liveAsB) then
    remove instrumentation for s
  end if
end if
```

There are some significant tradeoffs associated with the lazy inference approach, however. Most obvious is the complexity of designing and understanding the algorithm. Algorithm 1 is carefully constructed to achieve good time and space complexity characteristics. Despite the trivial nature of the pattern being inferred, constructing such an

algorithm to perform efficiently is a non-trivial task. Thus such an approach is likely to be much more difficult to apply to more complex patterns.

It is clear also that the lazy inference algorithm incurs a quadratic run-time perfomance in the worst case, as does the eager FSA technique. We believe in practice that this is highly unlikely, given the typical nature of usage of APIs in practice. It is, however, conceivable that certain patterns of behavior in monitored programs may result in poor performance of the algorithm that could exceed the costs of the eager approach. There are also circumstancs under which the eager approach can remove instrumentation for an event that has never even been observed, a scenario that is not possible with the lazy approach.[6] Such behaviors must be considered in comparison to the cost of implementing a manual design of the algorithm.

The lazy approach seeks to avoid the combinatorial blowup in pattern instances to match of the eager approach. It does this by calculating information about candidate $(AB)^*$ patterns only for observables that have occurred in the program execution. The eager would check for a pattern involving calls that never occur, whereas the lazy approach would not. A clear implication of this difference is that the lazy approach may fail to infer candidate patterns that the eager approach would still identify as possibilities. The interpretation of this situation depends on perspective and objectives. It may be that, even if evidence does not strictly contradict it, one would not consider a behavior that is never observed to be a candidate pattern. Conversely, if the test suite or inputs used to drive the inference process are inadequate, the absence of information may fail to reveal true patterns.

Finally, it is worth noting that the comparisons given here between the two approaches likely provide an incomplete picture. The initialization cost of the eager approach is highly sensitive to the size of the API, independent of the size of the input. This cost is more effectively amortized over longer runs of the program. As a consequence, the small size of inputs used in our preliminary comparisons likely unfairly penalize the performance of the eager approach. A question for future investigation is whether the two approaches may tend toward similar costs on larger inputs, such that the extra complexity of the lazy approach may not be justifiable.

6 Related Work

There have been many research efforts to enhance the efficiency of profiling activities. Most of these efforts can be classified into three groups.

The first group includes techniques that perform up-front analysis to minimize the number or improve the location of the probes necessary to profile the events of interest. These techniques utilize different program analysis approaches to avoid inserting probes that can render duplicated or inferable information. For example, discovering domination relationships can reduce the number of probes required to capture coverage information [1], identifying and tagging key edges with predetermined weights can reduce the cost of capturing path information [4], and optimizing the instrumentation payload code can yield overhead improvements [30]. Since these techniques operate before program execution, they are complementary to, and can be applied in combination with, the adaptive technique we propose.

The second group of techniques utilize the notion of sampling. These techniques select and profile a subset of the population of events to reduce profiling overhead while sacrificing accuracy. Their effectiveness depends on the

[6] At best, the lazy algorithm may determine removal of instrumentation for an event immediately after it is first observed. While minor, there is nonetheless an extra cost associated with such behavior.

sample size and the sampling strategy. Techniques are available to sample across multiple dimensions, such as time [15], population of events [3, 23, 38], or deployed user sites [13, 27], while their strategies range from basic random sampling (used by many of the commercial and open source tools) to conditionally driven paths based on a predefined distribution [23], or stratified proportional samples on multiple populations [13]. The flexibility offered by the various sampling schemes makes them very amenable for profiling activities that can tolerate some degree of data loss. Our approach could be perceived as performing a form of directed systematic sampling, where the subset of observables is selected by a given FSA state.

The third group of techniques that has emerged recently aims at adjusting the location of probes during the execution of the program, by removing or inserting probes as certain conditions are met. Several frameworks such as Pin [25], DynInst [35] and the commercial JFluid (now a part of the NetBeans professional package [20]) have appeared to support such activities. Our community has started to leverage these capabilities to, for example, reduce the coverage collection overhead through the removal of probes corresponding to events that have already been covered by a test suite [7, 8, 26, 31]. This has been particularly effective when applied to extensive and highly repetitive tests, resulting in overhead reductions of up to one order of magnitude.

Adaptive on-line program analysis fits in the latest group of techniques that adjust the required probes during the execution of the program. It is more general than existing techniques oriented toward coverage-probes removal, since it can handle more complex properties that may require the insertion of probes as well. And the technique is generic enough that it can be implemented on any dynamic instrumentation framework that supports the ability to add and remove instrumentation at runtime.

There is a significant and growing body of literature on run-time verification and temporal property inference. We have explained our work in terms of event observations of program behavior, e.g., entering a method or exiting a method, with restricted forms of data, e.g., thread and receiver object id's. It is important to note that arbitrary portions of the data state can also be captured by `Sofya` instrumentation. The instrumentation cost is higher when large amounts of data are captured, but for state-based properties, such as those captured by [5, 16], this would be necessary. Given that `Sofya` can observe all writes to fields, it is easy to see how adaptive temporal logic monitoring can be implemented in our framework. To the best of our knowledge, inference techniques have not considered data, other than receiver object, as a means of correlating sequences of API calls. This would be possible in our approach and, moreover, including additional constraints would speed the disabling of observables, thereby improving performance. We view adaptive analysis infrastructure as providing potential added value to any stateful checking or inference analysis, since it is independent of the particular analysis problem. It relies only on notification of when an observable is relevant and when it is irrelevant to the analysis.

7 Conclusions

We have proposed a new approach to dynamic program analysis that leverages recent advances in run-time systems to adaptively vary the instrumentation needed to observe relevant program behavior. This approach is quite general, as is the `Sofya` infrastructure on which we have implemented it. It also appears to be very effective, reducing the overhead of demanding stateful analysis problems from orders of magnitude to less than 33% percent of the un-instrumented program. Furthermore, for many properties it appears that the overhead burden is confined to initialization time, and

the rate of growth in runtime of adaptively analyzed programs and un-instrumented programs parallel each other as input sizes increase.

We believe that there are a wealth of research opportunities to be explored with adaptive online program analysis, such as making a wider variety of analyses adaptive, studying the cost and effectiveness of those analyses over a broad range of programs, and further optimizing the performance and usability of adaptive analysis infrastructure.

References

[1] H. Agrawal. Efficient coverage testing using global dominator graphs. In *Works. on Prog. Anal. for Softw. Tools and Eng.*, pages 11–20, 1999.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Symp. Princ. Prog. Lang.*, 2002.

[3] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Conf. on Prog. Lang. Design and Impl.*, pages 168–179, 2001.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Int'l. Symp. on Microarchitecture*, pages 46–57, 1996.

[5] E. Bodden. J-lo : A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, Germany, Nov 2005.

[6] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Int'l. Conf. Tools Alg. Const. Anal. Sys.*, LNCS, 2005.

[7] K.-R. Chilakamarri and S. Elbaum. Reducing coverage collection overhead with disposable instrumentation. In *Int'l. Symp. Softw. Rel. Eng.*, pages 233–244, Washington, DC, USA, 2004. IEEE Computer Society.

[8] K.-R. Chilakamarri and S. Elbaum. Reducing coverage collection overhead with disposable instrumentation. *Journal of Softw. Test., Rel., and Verif.*, 2007 (to appear).

[9] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *Int'l. W. Dyn. Anal.*, 2005.

[10] H. Do, S. G. Elbaum, and G. Rothermel. Subject infrastructure repository. http://esquared.unl.edu/sir.

[11] H. Do, S. G. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proc. Int'l. Symp. Emp. Softw. Eng.*, pages 60–70, 2004.

[12] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Int'l. Conf. on Softw. Eng.*, May 1999.

[13] S. G. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans. Softw. Eng.*, 31(4):312–327, 2005.

[14] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Symp. Princ. Prog. Lang.*, pages 256–267, 2004.

[15] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Symp. on Compiler Construction*, pages 120–126, 1982.

[16] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Meth. Sys. Design*, 24(2):189–215, 2004.

[17] G. Holzmann. The power of 10: rules for developing safety-critical code. *IEEE Computer*, 39(6), 2006.

[18] Java: Programming With Assertions. http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html.

[19] http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/.

[20] The netbeans profiler project. http://profiler.netbeans.org/.

[21] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. V. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Meth. Sys. Design*, 24(2):129–155, 2004.

[22] A. Kinneer, M. Dwyer, and G. Rothermel. Sofya: A Flexible Framework for Development of Dynamic Program Analyses for Java Software. Technical Report TR-UNL-CSE-2006-0006, University of Nebraska - Lincoln, April 2006.

[23] B. Liblit, A. Aiken, and A. Zheng. Distributed program sampling. In *Conf. on Prog. Lang. Design and Impl.*, pages 141–154, 2003.

[24] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *Conf. on Prog. Lang. Design and Impl.*, pages 219–230, 2004.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Conf. on Prog. Lang. Design and Impl.*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[26] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *Int'l. Conf. Softw. Eng.*, pages 156–165, New York, NY, USA, 2005. ACM Press.

[27] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *Int'l. Symp. Softw. Test. Anal.*, pages 65–69, 2002.

[28] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comp. Sys.*, 15(4):391–411, 1997.

[30] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *Conf. on Prog. Lang. Design and Impl.*, pages 196–205, New York, NY, USA, 1994. ACM Press.

[31] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Int'l. Symp. Softw. Test. Anal.*, pages 86–96, New York, NY, USA, 2002. ACM Press.

[32] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Trans. Softw. Eng.*, 32:93–110, Feb 2006.

[33] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Conf. Tools Alg. Constr. Anal. Sys.*, pages 461–476, April 2005.

[34] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Int'l. Symp. Softw. Test. Anal.*, pages 218–228, 2002.

[35] C. C. Williams and J. K. Hollingsworth. Interactive binary instrumentation. In *Int'l. Works. on Remote Anal. and Measurement of Softw. Sys.*, May 2004.

[36] J. Yang, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Int'l. Conf. Softw. Eng.*, 2006.

[37] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. In *In Int'l. Symp. Softw. Rel. Eng.*, pages 340–351, Washington, DC, USA, 2004. IEEE Computer Society.

[38] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.