

Adaptive Online Program Analysis*

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

Abstract

Analyzing a program run can provide important insights about its correctness. Dynamic analysis of complex correctness properties, however, usually results in significant run-time overhead and, consequently, it is rarely used in practice. In this paper, we present an approach for exploiting properties of stateful program specifications to reduce the cost of their dynamic analysis. With our approach, analysis results are guaranteed to be identical to those of a traditional expensive dynamic analyses, while analysis cost is very low – between 23% and 33% more than the un-instrumented program for the analyses we studied. 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 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 se-

quencing constraints [7, 14, 19] and to find concurrency errors [12, 26, 29] have been proposed in recent years.

Existing monitoring approaches, however, are not entirely adequate to efficiently support such analyses. Techniques aimed at reducing the instrumentation necessary to monitor a program were designed for simpler properties related to program structure, such as basic blocks and paths [1, 3], and are not applicable to more complex properties. Sampling techniques can effectively control analysis overhead, but their lossy nature makes them inappropriate for properties that depend on exact sequencing information, where missing the observation of an event may result in either a false report of conformance or of error. The inability to drastically reduce instrumentation or utilize sampling makes dynamic analysis of stateful properties expensive, with run-time overheads ranging from a factor of “20-40” [12] to “several orders of magnitude” [4]¹. As a consequence they have not been widely adopted by practitioners.

There are two broad classes of dynamic analyses. 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 [31] 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., [5], which demands that the analysis be performed online.

In this paper, we present *adaptive online program analysis* (AOPA) as a means of reducing the run-time overhead

*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.

¹Most published research in this area fails to even mention run-time overhead, much less provide clear performance measurements as was reported for Atomizer [12], so we assume that it is one of the better performing techniques.

of performing dynamic analyses of stateful program properties. It may seem counter-intuitive to advocate an on-line 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 [6, 23, 28]. 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 execution proceeds 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 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. Furthermore AOPA, through the removal of instrumentation at points during analysis, can lead to orders of magnitude reduction in the overhead of run-time analysis of stateful properties.

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 two significant contributions. (1) In Section 3, we describe the implementation of an efficient infrastructure to support adaptive program analysis of Java programs using the Sofya [20] analysis framework; and (2) We define 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. In addition, in Section 4.5, we describe preliminary experience implementing adaptive online temporal sequence property inference techniques. We discuss related work in Section 5, 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 6.

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. Legal call sequences on this API are defined by the regular expression:

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

```
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 Example

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 the *sink* state. Let $\Sigma = \{\text{open}, \text{close}, \text{read}, \text{write}, \text{eof}\}$ 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 $\{\text{read}, \text{write}, \text{eof}\}$ 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

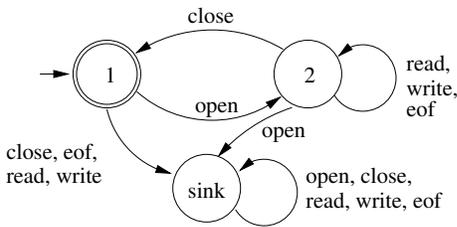


Figure 2. File API Protocol FSA

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

Table 1. Self and Outgoing Symbols

the 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 4.5, 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, traditional, non-adaptive, analysis for the example in Figure 1 would only consider 3 calls if $k = 0$, i.e., the file is empty. For a different property, such as a property stating that `open` must precede `close`, i.e.,

$(\sim[\text{close}]^*) \mid (\sim[\text{close}]^*; \text{open}; \cdot)^*$

where $\sim []$ means any symbol not inside the brackets, one would restrict the instrumentation to only `open` and `close` calls. Since our program only has 2 such calls, adaptive and non-adaptive analyses will process the same number of calls. Given the variations in performance that are possible, we would like to better understand the extent to which the purported benefits of adaptive analysis 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 lend themselves to efficient adaptive analysis. (1) In [10] we defined property specifications us-

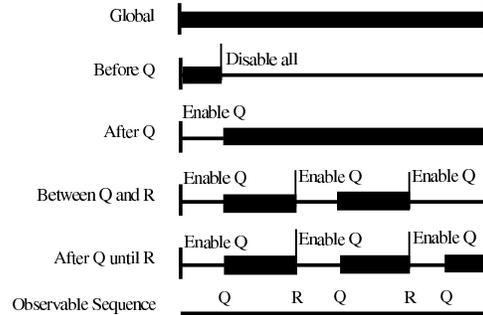


Figure 3. Specification Pattern Scopes

ing 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 may or may 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 [10], by studying existing temporal sequencing specifications, that properties like the cyclic `open-close` and precedence properties described 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 [4, 12].

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` [20] 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

```

public final class InstrumentationManager {
    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, boolean synchronous)
    ...
    public void updateInstrumentation() { ... }
}

```

Figure 4. Sofya API (excerpt)

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 faithfully² in both single and multi-threaded programs, Sofya employs a novel combination of byte code instrumentation with the Java Debug Interface (JDI) [17] – 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 locks³), and because it cannot deliver some events efficiently (such as method entry and exit). 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.

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. 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. We use 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 aggre-

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

³Java 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.

gated 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.

To illustrate how the API is used, 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 attached to an object based splitter that produces an FSA checker, *C*, for each instance of the `File` type allocated during the program run. At the beginning of execution, *A* calls `enable("File", "File"); 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 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 checked at run-time. Assertions are now widely used during development [16, 25] and guidelines for safety critical software propose that assertions remain enabled during system operation [15]. 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. We also give a brief overview of how an analysis to *infer* such stateful properties can be made adaptive and thereby offer potential performance improvements.

4.1 FSA Checking

A variety of program properties can be expressed as finite-state machines or regular expressions, e.g., [10]. 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 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 of program behavior. 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 checker 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 5, 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 checker 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, since it performs no file IO, but for more realistic situations, where many FSAs may be checked 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 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

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

and the *self symbols* as

$$self(s) = \Sigma - out(s)$$

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. Ignoring self symbols in a trace will not change the acceptance of the trace by an FSA. 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

```
INIT()
1  scur = s0
end INIT()

HANDLER(oid a)
2  scur = δ(scur, a)
end HANDLER()
```

Figure 5. Online FSA checker

```
INIT()
1  scur = s0
2  enable(out(scur))
end INIT()

HANDLER(oid a)
3  snext = δ(scur, a)
4  enable(out(snext) - out(scur))
5  disable(self(snext) - self(scur))
6  scur = snext
end HANDLER()
```

Figure 6. Adaptive online FSA checker

FSA checking. Figure 6 sketches the algorithm for the analysis. Initially, program instrumentation for outgoing observables in the FSA start state are enabled. When 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 FSA checking. Disabling or failing to disable instrumentation for self symbols does not impact correctness, only performance.

4.3. Checking Multiple Properties

It is essential that adaptive analysis be able to check multiple properties in a single program run, since we may want to check multiple properties of a program or a single property over multiple instances of a class.

Our adaptive analyses are capable of checking FSA conformance over the lifetimes of independent objects during program execution, i.e., they are object-sensitive. To achieve this they are built using an *object based splitter*, which allows a single FSA checker to be instantiated many times to simultaneously check the property against multiple live instances of a class. Checking multiple FSAs, whether instances of the same property or different properties, complicates analysis. If the FSAs share any symbols in their alphabets, then their simultaneous operation can cause interference when one checker disables an observable that another requires. This would lead to incorrect analysis results relative to a non-adaptive analysis.

We describe a solution based on reference counting of instrumented observables by enhancing the algorithm in Figure 6 to obtain that of Figure 7. The concept is simple: reference counts are maintained that reflect the number of FSA checkers that require a given observable, and an observable is only disabled when the reference count reaches zero (lines 9-12 in Figure 7). An observable is enabled (lines 2-4 and 6-8 of Figure 7) when the first checker requests it after it has been disabled.

When multiple properties are analyzed simultaneously, the *global* alphabet of all observable symbols is the union

```

INIT()
1   $s_{cur} = s_0$ 
2  for each  $a \in out(s_{cur})$  do
3    if ( $count[a]++ == 1$ ) then
4       $enable(a)$ 
end INIT()
HANDLER(oid a)
5   $s_{next} = \delta(s_{cur}, a)$ 
6  for each  $b \in (out(s_{next}) - out(s_{cur}))$  do
7    if ( $count[b]++ == 1$ ) then
8       $enable(b)$ 
9  for each  $b \in (self(s_{next}) - self(s_{cur}))$  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 7. Adaptive online multi-FSA checker

of the individual property alphabets, $\Sigma_{global} = \bigcup_i \Sigma_i$. In this situation, each FSA, k , is redefined over this global alphabet by introducing self-loop transitions in each state for all symbols in $\Sigma_{global} - \Sigma_k$. We note that this changes neither the meaning of the property, nor the outgoing symbol set definitions. Our experience suggests that maintaining small outgoing symbol sets is a key factor in reducing the cost of adaptive analysis.

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, 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.

Properties. Figure 8 shows two examples of the kind of regular expression that our analyses accept as input; to conserve space we have elided the JNI strings used to define calls based on signatures. On the left is an instance of a *precedence* specification pattern [10] 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).

On the right of the figure is a more complicated *constrained-response* pattern instance specifying the cyclic occurrence of `setBuilder` and `getResult` calls; we refer to this property as **Parser Builder** (pb). This specification has named parameters, e.g., p and b , and *wild cards*, “*”, 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.

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

Table 2. Application code-base measures

Artifacts. We wrote two precedence properties and two constrained-response properties to capture expected usage patterns of the NanoXML library; we acquired this program from the SIR repository [8, 9]. NanoXML is an open-source library for parsing XML structures that is designed to 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 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 10s to 100s of 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 use. 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 generated sequences of 10 input files increasing in size by a fixed amount for use in our evaluation.

Analyses. We ran several different analyses for each application and each property specification on increasing sizes

```

for events {
  "IXMLParser:parse",
  "IXMLParser:setReader" }
~["parse"]*
|
(
  ~["parse", "setReader"]*;
  "setReader";
  .*
)

for events { "IXMLParser.parse", "IXMLParser.setBuilder",
             "IXMLBuilder.startElement", "IXMLBuilder.getResult" }
~["*.setBuilder(*)"]*;
( "p.setBuilder(b)";
  ~["p.setBuilder(*)", "p.parse", "b.startElement", "b.getResult"]*;
  "p.parse";
  ~["p.setBuilder(*)", "p.parse", "b.getResult"]*;
  "getResult";
  ~["p.setBuilder(*)", "b.startElement", "b.getResult"]*
)*

```

Figure 8. API call precedence and constrained-response properties

app-property	noinst (sec)	jrat (sec)	adaptive (sec)	adaptive %overhead
XML2HTML-pb	10.5	34.1	13.2	25.7%
XML2HTML-pr	10.5	271.0	13.1	24.8%
XML2HTML-sbp	10.5	14.5	13.0	23.8%
XML2HTML-sbbsa	10.5	26.4	12.9	22.9%
JXML2SQL-pb	12.6	30.1	16.8	33.3%
JXML2SQL-pr	12.6	277.4	16.5	31.0%
JXML2SQL-sbp	12.6	15.5	16.3	29.4%
JXML2SQL-sbbsa	12.6	24.8	16.6	31.7%

Table 3. Timing at input sequence midpoints

of input. **noinst** is a completely un-instrumented version of the application; we use it to assess the overhead of our analyses. **jrat** is an instrumentation framework that uses BCEL to capture trace data from Java programs; it 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 [31]. **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 (on several examples, we observed that the cost increased at a rate that was more than twice that of jrat) which confirmed the prohibitive cost of performing online analysis without adaptation. 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.

Results. 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 performing 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 additional computation.

Table 3 reports the mean time in seconds, for the 6th of the 10 input files, 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 StartAdd** (sbbsa), and a constrained-response property relating IXMLReader and IXMLParser, called **Parser Reader** (pr). Structurally these two properties are similar to the ones shown in Figure 8. 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 9 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.

We note that the performance advantage of the adaptive analysis relative to jrat is an underestimate, since an offline dynamic analysis would incur 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 provide only 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.

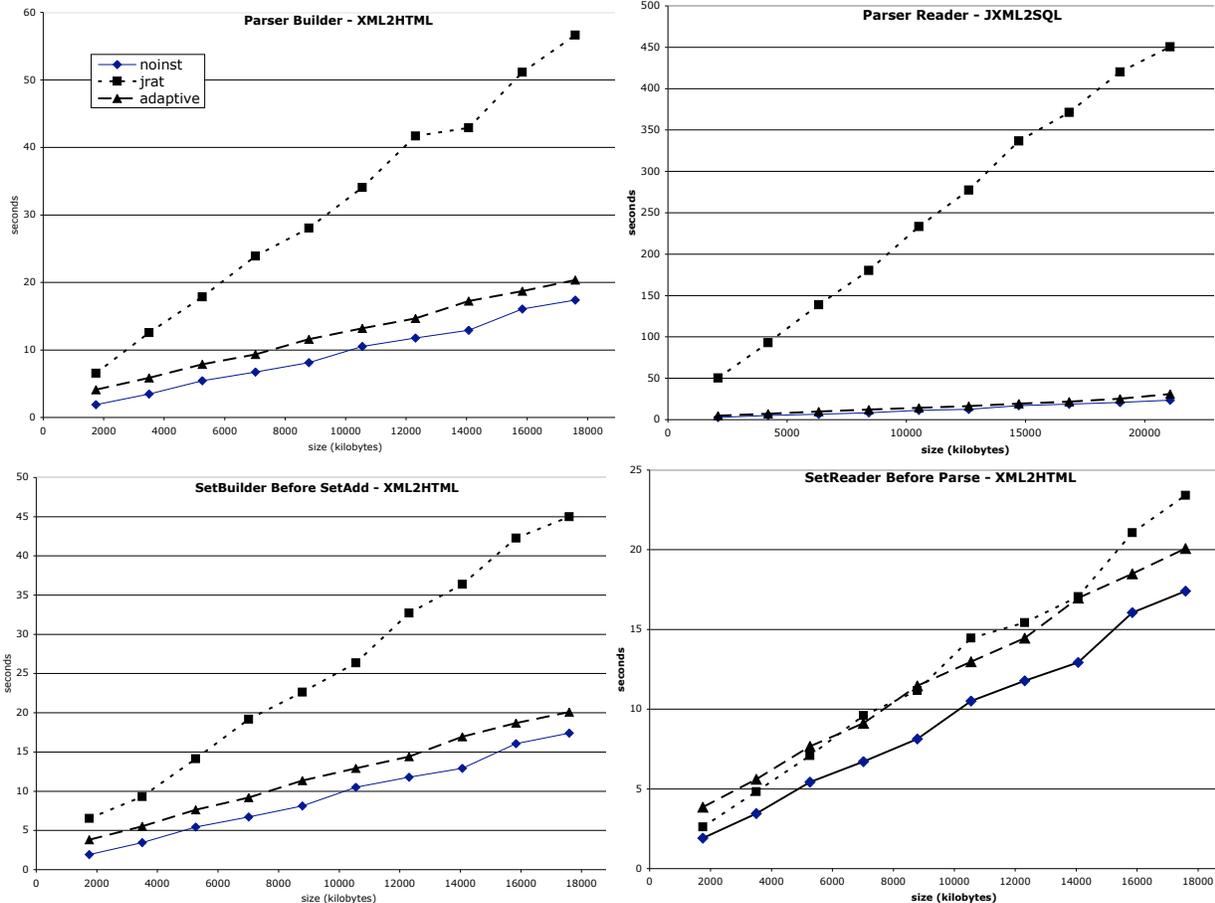


Figure 9. Growth of FSA checking analysis cost with XML file size

4.5. Adaptive Online Property Inference

In recent years a number of researchers have proposed analyses to *infer* sequencing properties from program traces. We have implemented adaptive variants of several inference algorithms, including one capable of inferring a range of sequencing patterns over public calls on a designated API. For illustration purposes, we explain how our analysis operates to infer Yang et al.’s *Alternating* pattern [31], i.e., $(AB)^*$ pattern.

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 10 gives the generic structure of an FSA for this pattern, where A and B are bound to each pair of calls. It may seem 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 4 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 10 are operating simultaneously, making independent transitions into different states (represented in each cell) based on the sequence of observable events; 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 $(\text{open}; \text{close})^*$. We note that after the third event has occurred, all FSAs involving `eof` (`oe`, `eo`, `ec`, and `ce`) have transitioned to their sink states and the instrumentation for `eof` is removed. Thus, property inference over this alphabet for this program will require 4 observable events, regardless of the size of the program input. In this simple example, only a single instance of the `File` is allocated, but, as discussed earlier in this section, multiple instances are handled naturally by the observable reference counting technique.

One significant advantage of this approach is that it is

AB	Observable Trace						Outcome
	o	e	e	e	e	c	
oc	2	2	2	2	2	1	✓
co	s	-	-	-	-	-	×
oe	2	1	s	-	-	-	×
eo	s	-	-	-	-	-	×
ec	1	2	s	-	-	-	×
ce	1	s	-	-	-	-	×

Table 4. File Trace and FSA Transitions

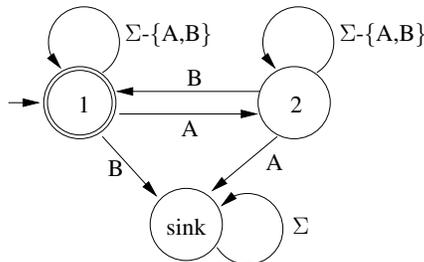


Figure 10. Generic $(AB)^*$ FSA

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 in addition to Alternating patterns. In general, there may be combinatorially many pattern instances, but for NanoXML the relatively modest sizes of the public APIs, which ranged from 9 to 14 method calls, and the patterns of client usage that exercised that API quite extensively, allowed our analyses to finish in less than 2 minutes for the applications considered in the previous section. The analyses inferred all of the expected properties.

5. 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 [3], and optimizing the instrumentation payload code can yield overhead improvements [27]. 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 sample size and the sampling strategy. Techniques are available to sample across multiple dimensions, such as time [13], population of events [2, 21], or deployed user sites [11, 24], 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 [21], or stratified proportional samples on multiple populations [11]. 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 [22], DynInst [30] and the commercial JFluid (now a part of the NetBeans professional package [18]) 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 [6, 23, 28]. 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., 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 to capture more observation data, but for state-based properties, e.g., [4, 14], this would be necessary. Since *Sofya* can observe all writes to fields, we believe it would be straightforward to implement adaptive online temporal logic checking in our framework.

6. Conclusions

We have proposed a new approach to dynamic program analysis, AOPA, 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. AOPA also appears to be very effective, reducing the overhead of demanding stateful analysis problems from orders of magnitude to less than 33% percent over the un-instrumented program. Furthermore, for many properties it appears that adaptive overhead is confined to initialization time, and the rate of growth in run-time 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 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] 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.
- [3] T. Ball and J. R. Larus. Efficient path profiling. In *Int'l. Symp. on Microarchitecture*, pages 46–57, 1996.
- [4] E. Bodden. J-lo : A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, Germany, Nov 2005.
- [5] 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.
- [6] 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.
- [7] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *Int'l. W. Dyn. Anal.*, 2005.
- [8] H. Do, S. G. Elbaum, and G. Rothermel. Subject infrastructure repository. <http://esquared.unl.edu/sir>.
- [9] 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.
- [10] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Int'l. Conf. on Softw. Eng.*, May 1999.
- [11] S. G. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans. Softw. Eng.*, 31(4):312–327, 2005.
- [12] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Symp. Princ. Prog. Lang.*, pages 256–267, 2004.
- [13] 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.
- [14] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Meth. Sys. Design*, 24(2):189–215, 2004.
- [15] G. Holzmann. The power of 10: rules for developing safety-critical code. *IEEE Computer*, 39(6), 2006.
- [16] Java: Programming With Assertions. <http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>.
- [17] <http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/>.
- [18] The netbeans profiler project. <http://profiler.netbeans.org/>.
- [19] 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.
- [20] 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.
- [21] B. Liblit, A. Aiken, and A. Zheng. Distributed program sampling. In *Conf. on Prog. Lang. Design and Impl.*, pages 141–154, 2003.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.
- [26] 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.
- [27] 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.
- [28] 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.
- [29] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Trans. Softw. Eng.*, 32:93–110, Feb 2006.
- [30] C. C. Williams and J. K. Hollingsworth. Interactive binary instrumentation. In *Int'l. Works. on Remote Anal. and Measurement of Softw. Sys.*, May 2004.
- [31] J. Yang, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Int'l. Conf. Softw. Eng.*, 2006.