jMonitor: Java Runtime Event Specification and Monitoring Library

Murat Karaorman [1]

Texas Instruments, Inc.
315 Bollay Drive, Santa Barbara, California USA 93117

Jay Freeman [2]

College of Creative Studies
University of California
Santa Barbara, California USA 93106

Abstract

jMonitor is a pure Java library and runtime utility for specifying event patterns and associating them with user provided event monitors that get called when the specified runtime events occur during the execution of legacy Java applications. jMonitor APIs define an event specification abstraction layer allowing programmers to design event patterns to monitor runtime execution of legacy Java applications. jMonitor instrumentation works at the Java bytecode level and does not require the presence of source code for the Java application that is being monitored. jMonitor overloads the dynamic class loader and takes the event specification and monitors (in the form of Java class files) as additional arguments when launching the target Java application. The class bytecodes of the monitored Java program are instrumented on the fly by the jMonitor class loader according to the needs of the externally specified jMonitor event patterns and event monitors.

Key words: Aspect-oriented programming, event monitoring, byte-code instrumentation, runtime verification, jMonitor.

1 Introduction

jMonitor is a pure Java library and runtime utility which allows programmers to specify event patterns to monitor runtime execution of legacy Java applications. jMonitor works by overloading the dynamic class loader. The jMonitor

[1] Email: muratk@ti.com
[2] Email: saurik@saurik.com

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

class loader instruments the class bytecodes of the monitored Java program on the fly according to the externally specified event patterns and event monitors. jMonitor instruments class bytecodes directly without requiring the source code of the monitored Java application.

During the execution of an instrumented application, each Java bytecode instruction that matches any of the specified event patterns triggers the call of one or more associated monitor methods. The monitor methods get called with the following runtime context information regarding the triggering event: the type of event, its target object, the call stack representing the method in which the event occurred, and the arguments to the method which collectively defines the full call context when the event occurred.

jMonitor events correspond to fundamental Java programming abstractions such as reading or writing of a field in a class, method invocation, method return or throw of an exception, and creation of a new object or array. Each event is also qualified with a Java application context such as the name of the field or the method and the names of the class and method context. The names are specified as strings representing POSIX compliant regular expressions.

Several distinct event monitors can be associated with any event. jMonitor instruments applications to capture the call context and call the monitor function with this information. Each monitoring function is called *before, after* or *instead of* the associated event depending on the event specification.

jMonitor presents a flexible and powerful event modelling and monitoring paradigm that offers the programmer some of the same benefits of aspect oriented programming.

The organization of the paper is as follows. In section 2 we introduce the jMonitor events, event patterns and event monitors. In section 3 we describe the different types of event monitors and the types of runtime context information collected and made available to the event monitor through jMonitor instrumentation. In section 4 we present the design and implementation overview. Section 5 covers how jMonitor relates to existing work in the field.

2 jMonitor Event Patterns

In this section we introduce jMonitor event patterns. Each event pattern describes a particular Java runtime event, such as the read or write of a field, or a method call, along with a constraining call context. The call context statically binds the specified runtime event to a legacy application domain. jMonitor uses the given event specifications to instrument legacy user application to detect when event is triggered and call any associated monitor function. Figure 1 illustrates the general idea of specifying event patterns using jMonitor APIs and attaching event monitors to each event pattern. A complete listing of jMonitor classes and method signatures is in Figure 4.

During start-up, the jMonitor instrumenting class loader calls the static `setEventPatterns` method of a user provided event pattern specification

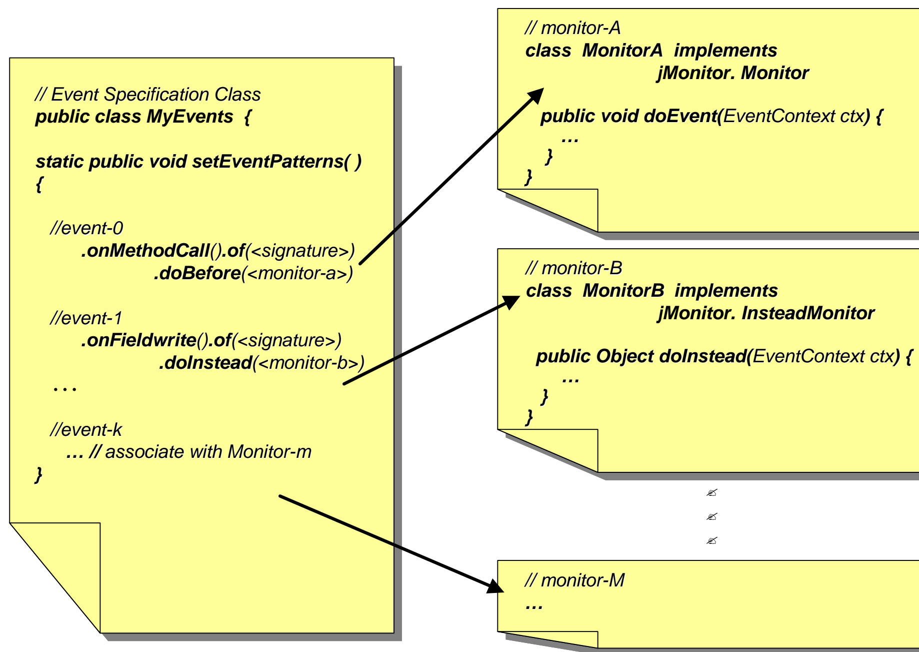

Fig. 1. User Defined Event Specifiers and Event Monitors

classes. Figure 2 illustrates the flow of information about the legacy application and the event monitoring layer. Each event pattern specified by the user subsequently guides jMonitor class loader to perform any needed on-the-fly instrumentation of the bytecodes of each class before it gets loaded.

2.1 jMonitor Event Types

Each jMonitor event pattern is based on at least one of the following fundamental Java language abstractions: the reading or writing of a field in a class, method invocation, method return or throw of an exception, or creation of a new object or array. The monitoring application layer builds each event pattern by calling `jMonitor.EventPattern` methods inside the `setEventPatterns` method. Table 1 summarizes the different types of Java events supported by jMonitor and the corresponding static `jMonitor.EventPattern` methods.

2.2 Specifying Event Contexts

Each jMonitor method listed in Table 1 returns a reference to a newly constructed `jMonitor.EventPattern` object. Each created EventPattern object

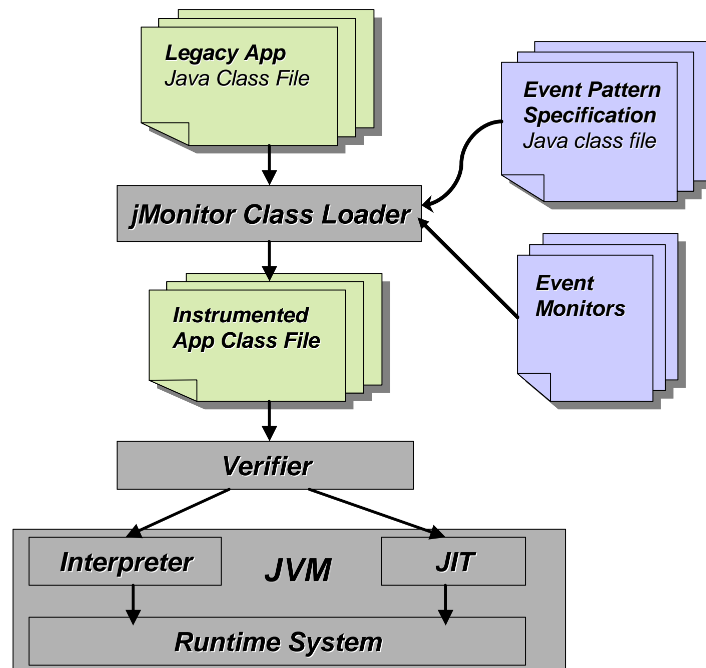

Fig. 2. jMonitor class loader performs on-the-fly bytecode instrumentation

can be further modified using one or more of the context definition APIs it supports as listed in Table 2 to narrow down the event context it matches.

For example, the following code snippet:

```
...
static public void setEventPatterns()
{
    EventPattern e1, e2, e3;
    e1 = jMonitor.EventPattern.onFieldWrite()
                              .of("\\.Foo.a$");
    e2 = e1.from("\\.MyApp\\.bar\\(");
    e3 = jMonitor.EventPattern.onMethodCall()
                              .of("\\.Foo\\.m\\(")
                              .from("Bar\\..*\\(");
    ...
}
```

defines a new event pattern, e1, corresponding to all “write” accesses to fields named “a” in all `Foo` classes, in any package. The second event pattern, e2, is derived from e1, but further constrains the new event pattern to match only when the write access to a `Foo.a` field happens during the execution of a

Table 1
Event Types

Event type	**EventPattern method**	**Argument**	**Event description**
Field read	`onFieldRead`	None	Read of a field, directly or through an object or static class reference
Field write	`onFieldWrite`	None	Assignment of value to a field, directly or through an object or static class reference
Method invocation	`onMethodCall`	None	Method call through an object reference or directly within same scope or as a static method
Return	`onReturn`	None	Issue of a `return` instruction from a method context
Throw exception	`onThrow`	None	Issue of a `throw` instruction from a method context
Instance creation	`onNew`	None	Issue of a `new` instruction used to instantiate a new object of a concrete class
Array creation	`onArrayCreate`	None	Issue of a `new` instruction used to create a new array
Any event	`onAnyEvent`	None	Any of the events listed in this table

`MyApp.bar` function call. The event pattern e2 matches a subset of all events described by e1. Similarly, each `EventPattern` context definition method call builds a new event pattern with additional constraints. The event pattern e3 matches all method calls for `Foo.m()` placed from anywhere within package or class `Bar`.

2.3 On Regular Expression Syntax Notation

Something that must be noted while reading the code examples is that in POSIX and Perl regular expressions the "." symbol is used to match any single character. As Java package names use this character, in order to explicitly match a package name (such as java.lang) and prevent from matching other names that may match (such as javaSlang) the "." must be escaped to the regular expression engine. This is done with the "\" character. Unfortunately, "\" is also the escape character used by Java for it's strings, and "\" itself must

Table 2
EventPattern Context Definition Methods

EventPattern method	Argument type	Argument Description
`of`	Method, Field, Class or Exception signature	Regular expression to match against the event target's signature
`from`	Method signature	Regular expression to match against the signature of the method that immediately caused the event
`in`	Method signature	Regular expression to match against any method's signature in the runtime call stack
`setName`	String	Assigned name for the pattern
`getName`	none	

Table 3
EventPattern Logic Operators

EventPattern method	Argument	Description
`and`	EventPattern object	Returns new event pattern that must match both the target and argument event pattern
`or`	EventPattern object	Returns new event pattern that matches either the target or the argument event pattern
`not`	none	Returns new event pattern that matches all other events not matched by the argument event pattern

be escaped. Therefore, when the sequence "\\. " is seen in one of jMonitor's match expressions it should be read as "match a single period here" as doubly escaped through Java and regular expressions.

2.4 Logic Operators for Combinining Event Patterns

New event patterns can be constructed using the defined logical operators defined in Table 3. For example, the following code snippet:

```
jMonitor.EventPattern e1, e2, e3, e4;
e1 = jMonitor.EventPattern.onFieldRead().of("\\.Foo\\.a$");
e2 = jMonitor.EventPattern.onFieldWrite().of("\\.Foo\\.a$");
e3 = e1.or(e2);
e4 = e1.and(jMonitor.EventPattern.onAnyEvent().
          from("[\\. ]Bar\\..*\\(").not());
```

defines a new event pattern, e1, matching all “read” accesses to fields named “a” in all Foo classes, in any package. The second event pattern, e2, is essentially same pattern defined for “write”. Therefore e3 matches the “read” or “write” of the fields named “a” in all Foo classes, in any package. Finally, e4 is defined to match all “read” accesses of fields named “a” in all Foo classes, exluding those issued from any method in package “Bar”.

2.5 Attaching Monitors to Events

Creating an event pattern object by itself does not necessarily result in the instrumentation of any application class bytecode to set up a monitor call trigger. Only by associating an event pattern with an event monitor will jMonitor instrument the matching context’s bytecodes. This association is established by calling one of the `doBefore`, `doAfter`, or `doInstead` methods on the event pattern. Matching event patterns to actual instructions in class bytecodes of the monitored application is performed statically during initialization, prior to loading any application class. The instrumented application methods simply call the event monitor methods when execution reaches the specified event trigger locations.

For event patterns built using the `of` and `from` constructs, no additional runtime checks are needed to determine whether a specific Java instruction matches the event pattern. Event patterns that include the `in` context definitions incur a very slight extra runtime overhead (single boolean test) during application execution around each instruction that potentially match the event pattern. This also imposes a similar overhead to the call of the function mentioned by the `in` construct.

To attach an event monitor to a particular event pattern it is sufficient to call one of the `setEventPatterns` methods listed in Table 4. An example event specifier class is illustrated in Figure 3. This example specifies an event pattern that replaces all calls to the `mypackage.MyClass.foo(Object)` function with the doInstead method of `mypackage.MyNullMonitor` class. Each method can be called multiple times to attach additional monitor methods that get called when the event is triggered.

Table 4
EventPattern Monitor Specification Methods

EventPattern method	Argument	Description
`doBefore`	Class name	Argument contains the name of the class containing a `doEvent` *monitor method* that gets called with the call context information immediately *before* the specified matching Java event occurs
`doAfter`	Class name	Argument contains the name of the class containing a `doEvent` *monitor method* that gets called with the call context information immediately *after* the specified matching Java event occurs. Not applicable for *return* and *throw* events.
`doInstead`	Class name	Argument contains the name of the class containing a `doInstead` monitor method that gets called with the call context information. This method is called *instead of* the specified matching Java event. The value returned back from the doInstead monitor is plugged back where appropriate to replace the corresponding Java event's evaluation

```
public class MyEvents
{
    static public void setEventPatterns() {

        jMonitor.EventPattern.onMethodCall()
            .of("int mypackage\\.MyClass\\.foo\\(Object\\)")
            .doInstead("mypackage\\.MyNullMonitor");
    }
}
```

Fig. 3. Example Event Pattern Specifier

3 Event Monitors

Each event pattern is associated with zero or more event monitors. A jMonitor event monitor is a pure Java class inheriting from one of the abstract classes in the `jMonitor` package: `Monitor` or `InsteadMonitor`. Each class corresponds to a particular type of the monitor that is attached to an event pattern. The `doAfter` and `doBefore` monitors implement the `jMonitor.Monitor` interface

```
P jMonitor                                  P: Package
  |                                         C: Class
  |C EventType                              I: Interface
  |  |M int getType()                       S: Static Method
  |  |M String toString()                   A: Abstract
  |                                         M: Method
  |C EventContext
  |  |M EventType getEventType()
  |  |M String getSignature()
  |  |M Object getTarget()
  |  |M Object getValue()
  |  |M Object[] getArguments()
  |  |M StackFrame[] getCallStack()
  |  |M EventPattern getEventPattern()
  |  |M Object passThrough()
  |  |M Object passThrough(Object[])
  |
  |I Monitor
  |  |M void doEvent(EventContext)
  |
  |I InsteadMonitor
  |  |M Object doInstead(EventContext)
  |
  |C StackFrame
  |  |M String getSignature()
  |  |M Object[] getArguments()
  |  |M String getSourceFile()
  |  |M Integer getSourceLine()
  |
  |C EventPattern
     |S EventPattern onFieldRead()
     |S EventPattern onFieldWrite()
     |S EventPattern onMethodCall()
     |S EventPattern onReturn()
     |S EventPattern onThrow()
     |S EventPattern onNew()
     |S EventPattern onArrayCreate()
     |S EventPattern onAnyEvent()
     |
     |M EventPattern of(String)
     |M EventPattern from(String)
     |M EventPattern in(String)
     |
     |M EventPattern and(EventPattern)
     |M EventPattern or(EventPattern)
     |M EventPattern not(EventPattern)
     |
     |M void setName(String)
     |M String getName()
     |
     |M void doBefore(String)
     |M void doAfter(String)
     |M void doInstread(String)
```

Fig. 4. jMonitor Class Hierarchy and Overview

and the abstract `doEvent` method. The `doBefore` and `doAfter` monitors are intended to be observer monitors, although the monitors are implemented as unconstrained Java methods and can have side-effects.

The doInstead type monitors on the other hand are intended to allow user level behavior replacement of the monitored events. These monitors implement the `jMonitor.InsteadMonitor` interface and the abstract `doInstead`

method. The Object result returned by the `doInstead` method gets used in the event behavior replacement logic of jMonitor instrumentation. The class(es) containing the event monitor methods are passed to jMonitor at runtime, specified either as a command line argument or placed in the classpath.

jMonitor instruments each Java instruction in any loaded class that matches one of the specified event patterns based on all the event monitors attached to the event via calls to the `doBefore, doAfter`, or `doInstead` methods. If multiple monitors methods are attached to the same event pattern the order in which they get called is not defined.

The instrumented application packages the requested call context information and calls the attached monitor methods with the call context as an argument. Table 5 depicts the information that comprises the context accessible by the monitor method through its jMonitor.EventContext argument. Figure 5 illustrate a fairly generic monitor for logging event traces with all available context information.

3.1 Behavior Modification Using Instead Monitors

When an application reaches the Java bytecode instruction that corresponds to a jMonitor event associated with an *instead monitor*, instead of executing the Java instruction, the monitor's `doInstead` method gets called, passing the call context as an argument. The value returned back from the doInstead monitor is subsequently plugged back where appropriate to replace the corresponding Java event's evaluation by the instrumented bytecodes.

Each doInstead monitor implements the `jMonitor.InsteadMonitor` interface and can use the `passThrough` method of the `jMonitor.EventContext` passed to the monitor to perform the original event that is being replaced. The `passThrough` method takes an `Object[]` representing the arguments for each event type:

- field read: no args
- field write: arguments[0] gets written
- method call: gets called with possibly modified `arguments[]` from the call context
- return: can't call `passThrough`
- throw exception: can't call `passThrough`
- new object or array: constructor is called with the `arguments[]` from the call context

The `passThrough` method can also be called with no arguments to assume the same argument that were originally passed to the monitored event. The `passThrough` call forces the execution of the event that is otherwise being replaced and returns the resulting object (where appropriate). The instead-Monitor may subsequently choose to return its own computed result or a result

Table 5
jMonitor Event Context Interface

EventContext method	Return type	Description
`getEventType`	`jMonitor.EventType`	type of jMonitor event that triggered monitor call
`getSignature`	`String`	Signature of the target that matched the "of" constraint.
`getTarget`	`Object`	The target object corresponding to the event
`getValue`	`Object`	Get the result or the exception returned or the value about to be written
`getArguments`	`Object[]`	Arguments supplied to the target event. getArguments()[0]holds the value for any 'write' event
`getCallStack`	`jMonitor.StackFrame[]`	Gets an array of stack frames corresponding to the runtime Java call stack. A stack frame is an object that contains: signature of the method, the arguments passed to the method (if available), source code file, line of the call for this method (if available)
`getEventPattern`	`jMonitor.EventPattern`	Gets the event pattern specification object that matched the current event

obtained from a `passThrough` call. Whatever value the doInstead monitor returns is used to replace the behavior of the original event being monitored. An example instead monitor is shown in figure 6.

4 Design and Implementation Overview

In order to monitor runtime events during the execution of a legacy Java application the developer must launch the target application using the jMonitor application launcher. The only additional information that must be provided at the command line to start a monitoring session is the list of event specifica-

```
public class TraceMonitor implements jMonitor.Monitor {

    public void doEvent(jMonitor.EventContext context) {

        System.out.println("Event: " +
            context.getEventType().toString() +
            " of " + context.getSignature());
        System.out.println("Pattern Name: " +
            context.getPattern().getName());
        System.out.println(" Target = " + context.getTarget());
        System.out.println(" Value = " + context.getValue());

        Object[] args = context.getArguments();

        if (args != null) {
            for (int a = 0; a != args.length; ++a) {
                System.out.println(" Arg #" + a + ": " + args[a]);
            }
        }
        jMonitor.StackFrame[] stack = context.getCallStack();

        if (stack != null) {
            System.out.println(" Call Stack:");

            for (int i = 0; i != stack.length; ++i) {
                jMonitor.StackFrame frame = stack[i];
                System.out.println(" " + frame.getSignature());

                Object[] args = frame.getArguments();

                if (args != null) {
                    for (int a = 0; a != args.length; ++a) {
                        System.out.println(" Arg #" + a + ": " + args[a]);
                    }
                }
                String file = frame.getSourceFile();
                if (file != null) {
                    System.out.print(" at: " + file);
                    Integer line = frame.getSourceLine();
                    if (line != null)
                        System.out.print("[" + line + "]");
                    System.out.println();
                }
            } } } }
```

Fig. 5. Example Monitor: TraceMonitor

Table 6
StackFrame Information

StackFrame method	Return type	Description
getSignature	String	When not null, signature of the method of stack frame in the call stack
getArguments	Object[]	When not null, arguments supplied to the method of the stack frame.
getSourceFile	String	When not null, name of the file for the method of stack frame
getSourceLine	Integer	line number in the source file for the method of stack frame

```
package mypackage;

public class MyNullMonitor implements jMonitor.InsteadMonitor {

    public Object doInstead(jMonitor.EventContext context) {

        Object[] args = context.getArguments();

        if (args.length != 0 && args[0] == null) {
            return new Integer(10);
        } else {
            return context.passThrough();
        }
    }
}
```

Fig. 6. Example InsteadMonitor: MyNullMonitor

tion classes. The names of the event monitor classes do get explicitly passed into the jMonitor application launcher as they will be dynamically loaded by the JVM on demand when an instrumented application class that references a monitor method gets loaded.

During its start-up initialization, jMonitor instrumenting class loader calls the static `setEventPatterns` method of the user provided event pattern specification classes. The jMonitor.EventPattern method calls within the `setEventPatterns` method builds the monitor event patterns. Event patterns then are associated with user specified event monitors using one of the `doBefore`, `doAfter`, or `doInstead` methods on the event pattern. Each event pattern specified by the user and attached to a monitor subsequently guides jMonitor class loader to perform any needed on-the-fly instrumentation of the bytecodes of

each class before it gets loaded. The following psuedo code illustrates the instrumentation logic used by jMonitor class loader.

```
For each loaded class, c
    For each method, m, in c
        For each instruction, i, in c.m
            For each user specified event-pattern, ep
                if i.matches(ep, m, c)
                    add ep.monitors to i.monitors
            if i.monitors not empty
                insert stub for context extraction
                if i.beforeMonitors is not empty
                    insert doBefore calls
                if i.insteadMonitors is empty
                    insert instruction i
                else
                    insert doInstead call
                    plug doInstead return
                if i.afterMonitors is not empty
                    insert doAfter calls
            else // i has no monitors
                insert instruction i
```

4.1 Status and Limitations

A prototype implementation of jMonitor for proof-of-concept has been implemented. Current implementation uses BCEL bytecode engineering library [2] and Apache Perl5 style regular expression library. An open source implementation offering full jMonitor functionality is planned.

One limitation imposed on the user is that event monitoring classes must be distinct from event specification classes (or declared within the specification classes as inner classes). This is necessary to prevent dynamic class from attempting to load monitor or legacy application classes before the event specification classes are loaded. jMonitor needs to load specification classes first and learn about all user defined event patterns before any other class is loaded, otherwise the instrumentation will be partial.

4.2 Performance

The matching of event patterns to actual intructions in class bytecodes of the monitored application that needs to be instrumented is performed statically during initialization, prior to loading any application class. There is no additional runtime overhead associated with event pattern matching involving those built using the `of` and `from` constructs. The instrumented application methods simply call the event monitor methods when execution reaches the

specified event trigger locations.

Event patterns that include the `in` context definitions, however, incur a very slight extra runtime overhead (single boolean test) during application execution around each instruction that potentially match the event pattern. It is important to note, however, that there is no runtime regular expression match overhead for matching the `in` patterns. All regular expressions are matched at instrumentation time.

To illustrate the mechanism, suppose there is an `in("^int .*\\(")` pattern, to match any function that returns an `int`. For this pattern object, jMonitor introduces a thread specific static boolean (so there will be one boolean per `in()` pattern per thread). When instrumenting a method, jMonitor checks to see if it matches the pattern for *all* `in` patterns anywhere for any event. If it does match, then instrumentation adds some code around the method to add a local boolean variable. If the current thread's boolean for this pattern is false, then this boolean gets set `true`. If not, then we set the local boolean to `false` and leave the pattern boolean as `true`. Then a finally clause is added to this function that checks if the local boolean is `true` (i.e., this call was the call that set the pattern's boolean to `true`) then instrumentation sets the pattern's boolean to `false`.

Subsequently, whenever jMonitor instruments an instruction for which there is an event pattern that contains an `in` constraint, it checks the current thread's boolean for each `in` pattern to see if it is set to `true`. Thus, we can avoid doing any runtime regular expression matches can support `in` constraints without much runtime overhead.

5 Related Work

5.1 Aspect Oriented Programming

jMonitor facilitates a programming model that appears to match the power of 'aspect-oriented programming' [3]. There are, however, several differences. The paradigm expressed by aspect oriented programming is one of development. It changes the way one designs and implements software. In comparison, the concept of runtime monitoring as implemented by jMonitor is put forth as something done more after the fact. When one is completed with their application and is attempting to get more of an understanding of why a particular behavior is happening, she may decide to attach monitors to various events for this purpose. All of the instrumentation is done at runtime, and these monitors may be used for only a subset of the application through the usage of the monitoring class loader. We envision debugging environments may be developed around this technology, not development environments. While the implementation of monitoring may share some likenesses to aspect oriented programming, the usage cases, and thereby the programming method it puts forth, are different.

5.2 *Java-MaC*

The tool bearing closest resemblance to jMonitor is Java-MaC, an implementation for Java of the Monitoring and Checking architecture [4]. Java-MaC supports a language for specifying events and alarms on a Java program in the form of a Java expression that begins at a static object. These events can detect changes made to fields of objects in the system as well as the beginning and end of method calls. Various other contexts are supported to limit the scope of such events.

One feature that Java-MaC has that makes it useful for particular kinds of events is that it tracks the references to a particular object. This allows it to determine that the fields of a specific object (as determined by a code path from some static class object) have been changed rather than simply that a field of an object of a specific type was changed. This functionality causes Java-MaC to incur a performance penalty, however.

Java-MaC lacks some of JMonitor's features, for example the equivalent of doInstead monitors, as well as the `in` constraints. jMonitor also has the ability to monitor field modifications or writes to fields based on name and type as matched by a string regular expression. Because of these differences we find JMonitor to be a lower-level tool than Java-MaC, and would be interested in seeing how tools such as Java-MaC could be implemented directly in Java using JMonitor rather than having to delve into complicated bytecode engineering directly.

5.3 *Valgrind*

Another project that is similar in nature and design is Valgrind [5]. Valgrind is a framework for doing instrumentation of compiled x86 code. Some of the tools that have been implemented using Valgrind are memory leak and overrun detectors, as well as profilers. Valgrind has a rather large runtime performance cost, however, in that even if no instrumentation is to be performed there is about a four to five times speed hit. Some other limitations of the tool is due to the limitations of its target domain: the environment of compiled x86 code. There is not nearly enough meta information in compiled x86 binaries to design general runtime instrumentation skins in Valgrind as most of the useful code details are lost during the compilation process. jMonitor, implemented in and for Java, has access to metadata information regarding what functions or fields are actually being accessed by any particular instruction.

Tools like Valgrind and other commercially successful binary instrumentation packages such as Rational's Purify and Quantify and Code coverage tools, or BoundsChecker do provide very valuable benefits to software developers in monitoring and detecting dynamic memory access and usage violations, program profiling, and code coverage. jMonitor offers fundamentally everything necessary to develop these types of tool support for Java application development.

5.4 jContractor

The system most similar in design and implementation approach to jMonitor is jContractor[1], a pure Java library based implementation of Design By Contract for the Java language. It is available as an open source project currently hosted at http://jcontractor.sourceforge.net/. jContractor was designed as part of Karaorman's Ph.D. thesis, designing pure library and reflection based techniques for extending object oriented languages[9].

jContractor contracts are written as Java methods that follow a simple naming convention. jContractor provides runtime contract checking by instrumenting the bytecode of classes that define contracts. jContractor can either add contract checking code to class files to be executed later, or it can instrument classes at runtime as they are loaded. Both jContractor and jMonitor are purely library based, requiring no preprocessing or modifications to the JVM. jContractor offers some limited runtime monitoring capabilities by allowing contract methods to use unconstrained Java expressions. Pre-, post condition and invariant methods can be used for monitoring purposes only at function entry and exit points, without control over or access to its call context. jMonitor is a much more fine-grained, lower level and light weight instrumentation approach ideally suited for event specification and monitoring.

5.5 Binary Component Adaptation

Another project sharing some similarities with jMonitor is the binary component adaptation (BCA) mechanism based on load time modification of Java byte codes [6]. Binary component adaptation (BCA) allows components to be adapted and evolved in binary form and on-the-fly (during program loading). Similar to jMonitor, BCA rewrites component binaries before (or while) they are loaded and requires no source code access. The approach is very flexible, allowing a wide range of modifications (including method addition, renaming, and changes to the inheritance or subtyping hierarchy). The differences between jMonitor and BCA are largely due to the application domain. BCA is designed to transform components or applications to adapt and evolve with changing interfaces and other design changes. The adaptations are prescribed in the form of delta specifications, such as adding or renaming methods or fields, extending interfaces, and changing inheritance or subtyping hierarchies. Some of these changes such as those that do not require modifications to the inheritance hierarchy can be supported by jMonitor. BCA, on the other hand is not designed to support detection and monitoring of the type of low level Java events that jMonitor provides.

5.6 Query Based Debugging

Lencevicius *et al.* [7] have developed a query-based debugging tool which, working somewhat similar to an SQL database query tool, finds all object tuples satisfying a given boolean constraint expression. The dynamic query based debugger continually updates the results of queries as the program runs, and can stop the program as soon as the query result changes. To provide this functionality, the debugger finds all places where the debugged program changes a field that could affect the result of the query and uses sophisticated algorithms to incrementally reevaluate the query. The on-the-fly debugger adds a capability to stop the java program just at prescribed execution phases and enables querying as well as allowing to change the query later. They have implemented such a dynamic query-based debugger for Java written in pure Java with no JVM modifications.

It seems possible to use a tool based on jMonitor to assist in similar type of debugging scenarios. jMonitor monitors can be written by the programmer to provide instant error alerts by continuously checking inter-object relationships while the debugged program is running. The monitor can continually update the results of queries (expressed as user level Java expressions) as the program runs, and can stop the program as soon as the query result changes. The programmer can specify event patterns matching all contexts where the debugged program changes a field that could affect the results of the query for an efficiency.

6 Conclusion

We have introduced jMonitor, a pure Java library and runtime utility for user level specification of event patterns and associating them with user defined event monitors. jMonitor class loader seamlessly instruments application classes to call the specified monitor functions when triggerring Java runtime events occur during execution.

One of the key benefits of jMonitor is the ease of use and intuitiveness of its approach to event modelling and monitoring. The approach is light weight and non-intrusive to typical programming and software development processes. Supporting regular expressions is very powerful and leads to very concise and simple usage when designing event-patterns. jMonitor does not require special compilers, pre-processors or special IDEs and since it does not require source-code or forced recompilation it supports legacy applications well.

jMonitor presents a flexible, powerful and yet pragmatic and intuitive event modelling and monitoring paradigm that offers the programmer most of the same benefits of aspect oriented programming but without requiring requiring significant changes to the way most Java programmer design and implement their software, and while supporting their legacy development tools and prac-

tices.

jMonitor can be used during the development, debugging, testing and deployment stages of the software lifecycle. When a developer needs to get more of an understanding of when and why a particular behavior is happening, he or she may decide to design event patterns and attach monitors to analyze the relevant events. We envision powerful tooling and debugging environments to be developed around jMonitor technology. Additionally, we envision adding tooling support to automate some of the mechanical (i.e. programmed specification) aspects of event pattern specification and event monitor selection.

jMonitor supports dynamic program monitoring and analysis. It can be used to gather information during program execution and use it to conclude properties about the program, either during test or in operation.

jMonitor supports program instrumentation. It can be used to instrument programs without requiring source code, to emit relevant events to an observer or to modify behavior of legacy applications.

jMonitor supports program guidance. It can be used to alter the behavior of a legacy program for example to adapt to a new paradigm or when its specification is violated. This ranges from standard exceptions to advanced planning. Guidance can also be used during testing to expose errors.

References

[1] Abercrombie, P., M. Karaorman, *jContractor: Bytecode instrumentation techniques for implementing design by contract in Java,* In Proceedings of Second Workshop on Runtime Verification, RV 02, Copenhagen, Denmark, July 26, 2002, (also in Electronic Notes in Theoretical Computer Science, URL: http://www.elsevier.nl/locate/entcs).

[2] Dahm, M., *Byte Code Engineering with the BCEL API*, Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin (1998).

[3] Kiczales, G., et al., *Aspect-Oriented Programming*, In Proceedings European Conference on Object-Oriented Programming (1997), 220–242.

[4] Kim, M., et al., *Java-MaC: A Run-time Assurance Tool for Java Programs*, in In Klaus Havelund and Grigore Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[5] Nethercote, N., Seward, J., *Valgrind: A Program Supervision Framework*, In Proceedings of the Third Workshop on Runtime Verification (RV'03), Boulder, Colorado, USA, July 2003, (also in *Electronic Notes in Theoretical Computer Science* Vol.89 (2003), URL: http://www.elsevier.nl/locate/entcs).

[6] Keller, R., and U. Hölzle, *Binary Component Adaptation,* In Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, July 1998, 307–332.

[7] Lencevicius, R., U. Hölzle, A.K. Singh, *Dynamic Query-Based Debugging,* In Proceedings of the 13th European Conference on Object-Oriented Programming'99, (ECOOP'99), Lisbon, Portugal, June 1999, (Also published as Lecture Notes on Computer Science 1628, Springer-Verlag,135–156).

[8] Lindholm, T., and F.Yellin, *The Java Virtual Machine Specification,* Addison-Wesley, Reading, 1999.

[9] Karaorman, M., "Pure Library and Reflection Based Techniques for Extending Object Oriented Languages, " Ph.D. thesis, University of California, Santa Barbara, 2000.