Instrumentation of Java Bytecode for Runtime Analysis

Allen Goldberg and Klaus Havelund

Kestrel Technology, NASA Ames Research Center
Moffett Field, MS 269-3, California USA
Phone: 650-604-4858, Email: {goldberg,havelund}@email.arc.nasa.gov

Abstract. This paper describes **JSpy**, a system for high-level instrumentation of Java bytecode and its use with **JPaX**, our system for runtime analysis of Java programs. **JPaX** monitors the execution of temporal logic formulas and performs predicative analysis of deadlocks and data races. **JSpy**'s input is an instrumentation specification, which consists of a collection of rules, where a rule is a predicate/action pair. The predicate is a conjunction of syntactic constraints on a Java statement, and the action is a description of logging information to be inserted in the bytecode corresponding to the statement. **JSpy** is built using JTrek an instrumentation package at a lower level of abstraction.

1 Introduction

This paper describes an instrumentation package, **JSpy**, for instrumenting Java [3] bytecode [11], and our application of **JSpy** to *run time analysis*. Run time analysis [1,2] refers to examination of a single program execution trace to check, for example, for conformance to a temporal logic assertions, or for detecting the potential for deadlocks and data races. The Java PathExplorer, **JPaX** [10], is our general framework for performing runtime analysis. **JPaX** has been incorporated into an an automatic test case generation system. See [4] for details.

JSpy can be seen as an Aspect Oriented Programming [9] environment package in the sense that it is guided by rules, or *aspects*, which specify how a program should be transformed to achieve additional functionality. However, the main purpose of these aspects is to extract information from a running program. **JSpy** is built on top of the low-level JTrek instrumentation package [7]. Other low-level instrumentation packages could be used, such as for example BCEL [8].

The paper is organized as follows. Section 2 gives a description of the overall functionality of **JSpy**, and of the input specification that drives it. Section 3 discusses some implementation issues and how to run it. Section 4 lays out an architecture for using **JSpy** to generate events to monitors, that check for temporal logic property conformance and multi-threading problems. Section 6 discusses some planned enhancements and gives conclusions.

2 Input Instrumentation Specification

JSpy produces instrumented code based on an *instrumentation specification* whose main component is a set of rules. We refer to the code to be instrumented as the *target* and the result of instrumenting the target the *instrumented target*. The instrumented target is functionally equivalent to the target except that, when executed it generates a stream of *event log objects* as specified by the rules of the instrumentation specification. The rules form a high-level declarative description of where to insert log creation code and the contents of the log entries. A design goal for the instrumentation package is to minimize the impact of the instrumentation code on the time and space of the target program. Our run time analysis system is designed so that the substantial work of analyzing event logs is done off-line via file communication, or by a remote process that reads the log stream via a socket connection.

Each rule is a predicate/action pair. The predicate is a conjunction of atomic predicates. Each atomic predicate defines a syntactic constraint on the sequence of bytecode instructions corresponding to a single Java statement. Examples of predicates are summarized in Figure 1. A name specification, for example *methodNameSpec*, specifies a fully qualified method name and method signature but allows wildcards for most components. Actions specify instrumenting directives that are inserted at statements satisfying the rule predicate. Example actions are given in Figure 2. The action *ReportExpression* takes an arbitrary Java expression, encapsulates it into a method and inserts a call to the method into the target. The method is inserted into a generated Java file inserted into the package containing the target code. By placing the code in the same target as the target class file most data values available to the target code will be visible to such a method. Other data, e.g. private data or local variables can be transmitted as a parameters to the constructed method. When this auxiliary class is completely populated with such methods, the Java compiler is called and the resulting bytecode becomes part of the instrumented target.

An example instrumentation specification, using the **JSpy** Java API, is illustrated in Figure 3. An instrumentation specification is built up from one or more rules, each being a set of predicates and a set of actions that are inserted if the predicates all evaluate to true. The example specification contains one rule, with one predicate and two actions.

The goal of minimizing the impact of instrumentation on execution time affects a fundamental design decision regarding the logging of program data in response to a *ReportLocal*, *ReportField* or *ReportExpression* action. If the type of value to log is primitive then the value can be inserted into the log. Since the size of the value is just a few bytes the log entry is itself small. For values that are objects there are four choices:

- A deep copy of the object is transmitted. That is, all fields of the object are transmitted, including all objects reachable from the fields of the object.
- A shallow copy is transmitted. That is, the values of primitive-type fields and an external reference to the values of non-primitive fields are transmitted.

Predicate Name	Parameters	Returns *true* iff. current statement:
InClass	**String** *classNameSpec*	is within class with name that matches *classNameSpec.*
InMethod	**String** *methodNameSpec*	is within method with name that matches *methodNameSpec.*
AtMethodStart	**String** *methodNameSpec*	is the first statement within method with name that matches *methodNameSpec.*
AtMethodEnd	**String** *methodNameSpec*	is the last statement or a return statement within method with name that matches *methodNameSpec.*
CallsMethod	**String** *methodNameSpec*	contains a call to a method who's name matches *methodNameSpec.*
AtFieldAccess	**String** *fieldNameSpec* **boolean** *onlyUpdates*	accesses a field who's name matches *fieldNameSpec.* If *onlyUpdates* is true, only write accesses are matched.
AtLocalAccess	**String** *varNameSpec* **boolean** *onlyUpdates*	accesses a local variable who's name matches *varNameSpec.* If *onlyUpdates* is true, only write accesses are matched.
AtSyncStart		enters a synchronized block, taking a lock.
AtSyncEnd		exits a synchronized block, releasing a lock.
AtStatementType	**String** *stmtType*	is a statement of type *stmtType*, which ranges over 32 different statement types, such as: "assign", "break", "return", "try", "throw", "while", etc.
InStatementRange	**int** *lowerBound* **int** *upperBound*	is within the range of line numbers *lowerBound* and *upperBound.*

Fig. 1. Examples of Predicates

- An external reference to the object is transmitted.
- Object values are not transmitted at all.

For some applications transmitting deep copies may be appropriate, e.g. for offline program debuggers. Our approach is to just transmit an external reference. Note, using *ReportExpression*, the value of any field is transmittable, so the effect of a shallow copy is achievable. Not transmitting object values is too limiting. In particular, it is necessary to log object references in order to properly log the taking and releasing of locks on those objects, information that is needed for example for deadlock and data races analysis.

However this raises a problem of how to externally reference an object. Java serialization is a potential solution to this since it provides external unique identifiers to objects, but serialization externalizes a portion of the heap at exactly one time instant. For our purposes a reference to the same object reported at

Action Name	Parameters	Inserts code that reports:
ReportMethodStart	**String** *methodNameSpec*	name of method entered, who's name matches *methodNameSpec*.
ReportMethodEnd	**String** *methodNameSpec*	name, and return value, of method exited, who's name matches *methodNameSpec*.
ReportMethodCall	**String** *methodNameSpec*	name of method that is called, who's name matches *methodNameSpec*.
ReportField	**String** *fieldNameSpec* **boolean** *onlyUpdates*	name of field that is accessed, who's name matches *fieldNameSpec*. If *onlyUpdates* is true, only write accesses are matched.
ReportLocal	**String** *varNameSpec* **boolean** *onlyUpdates*	name of local variable that is accessed, who's name matches *varNameSpec*. If *onlyUpdates* is true, only write accesses are matched.
ReportSyncStart	**String** *classNameSpec*	identity and class (name) of object that is locked, where the class name matches *classNameSpec*.
ReportSyncEnd	**String** *classNameSpec*	identity and class (name) of object that is released, where the class name matches *classNameSpec*.
ReportTimeStamp		the current time.
ReportProgramPoint		the current line number.
ReportExpression	**String** *expressionName* **String** *expressionBody* **byte** *expressionType* **String** *parameters*	the value of *expressionBody*, and an identifer *expressionName*. *parameters* and *expressionType* indicate the expression's free parameters and the result type.

Fig. 2. Examples of Actions

two points during execution must be the same, and this is not within the scope of serialization.

Unfortunately we do not have a completely satisfactory solution for constructing such an external reference. Our approach is to transmit as an external reference the hash code of the object, when viewed as an instance of `java.lang.Object`. This value is computed by the method `java.lang.System.identityHashCode(Object o)`. There is the remote chance that two distinct objects are assigned the same hash code. We decided that this imperfect solution is preferred to disallowing the transmission of object values. We treat strings as a special case, transmitting the whole object when requested.

An additional feature of **JSpy** is that actions can be conditionally executed. A conditional action is specified by providing a boolean-valued Java expression that gets compiled using the same mechanism as described for *ReportExpression*. The target is instrumented so that a call is made to a method at execution time, whose body uses the boolean expression to guard the logging action. In some

```
class LogLockAcquisitions{
  ...

  public void instrument(){
    InstrumentSpecification spec = new InstrumentSpecification();
    Rule rule = new Rule();
    rule.addPredicate(new AtSyncStart());
    rule.addAction(new ReportTimeStamp());
    rule.addAction(new ReportSyncStart("*"));
    spec.addRule(rule);
    Instrumenter ins = new Instrumenter(spec);
    ins.instrumentTheCode();
  }
}
```

Fig. 3. Example Instrumentation Specification

simple cases encapsulation can be avoided in favor of a low-overhead conditional placed directly in the code.

Some actions do not make sense in certain contexts. For example the action *ReportMethodStart* only makes sense instrumenting the first statement of a method. Consequently such actions will only be inserted at appropriate program points.

3 Implementation

In this section we discuss some implementation issues and how an instrumented program is run.

3.1 JTrek

JSpy is built using Jtrek [7]. Jtrek is a lower level bytecode modification tool with monitoring being one of the intended applications. JTrek was developed at COMPAQ SRC. Jtrek iterates through the bytecode instructions of the target and uses callbacks to perform user-specific instrumentation. JTrek allows insertion of certain types of code, but does not allow definition of new local variables, fields, or methods. Since calls to arbitrary methods are supported, it is possible to encapsulate any instrumentation action into a method that is inserted into the code.

Iteration may be at the level of Java statements or individual bytecode instructions. Jtrek limitations have made it difficult or impossible to cleanly implement some of the features of **JSpy**, but overall, it is a stable and well-designed software package.

As described above Jtrek provides an idiom of use based on an iteration, or trek, over the bytecode. **JSpy** is structured as such a trek; for each bytecode translation of a Java statement the predicate of each rule of the instrumentation specification is tested. The actions of all rules whose predicates are true are collected, sorted according to a canonical ordering on actions, and inserted into the code using Jtrek primitives. In most cases the inserted code is a call to a method that generates an entry into the event stream. For example, consider the implementation of the *ReportLocal* action for a field of type boolean. The code inserted will invoke a static method called *reportLocalBoolean* with three parameters, a string holding the name of the class and method in which the variable is defined, a string containing the local variable name, and the (boolean) value itself. This method will write a log entry.

Jtrek iterates over multiple class files; specifying the target classes to be instrumented is achieved in a manner similar to Java class loading. A root class, classpath, and a scope indicator are given. The scope indicator can be "all", "user", or "package". All files directly and indirectly referenced in the root class and consistent with the scope designator are instrumented. The "package" designator restricts instrumentation to files in the same package as the root class, "user" excludes instrumentation of system files, "all" means all reachable files are instrumented.

3.2 Execution of the Instrumented Target

As described above, the target is instrumented with calls to methods that construct a log entry and then write the entry to an output stream. In addition, auxiliary classes are constructed "on the fly" as necessary, and added to user packages containing the target code. Finally the instrumented target contains predefined supporting classes where such methods as *reportLocalBoolean* are found. Within the supporting classes is an abstract class of log objects and specializations of this class for each type of action.

The instrumented target is started as usual, e.g. an application is started by invoking the "main" method of the specified class. When an instrumentation method is first encountered in the code, the instrumentation class is loaded and a static initializer for the class is executed. This code attempts to read a property file that describes where the log output stream is routed. If no such file exists, then default routing is used. The output stream is routed to either a socket or file. The output stream consists of serialized log objects. The static initializer also sets up a shutdown hook (see `java.lang.Runtime.addShutdownHook`) to flush and close the buffered output stream when the application completes.

Synchronization occurs at the finest level of granularity, namely writing the log entry to the buffered output stream. This minimizes the effect of synchronization on performance of multi-threaded targets. However, as a consequence, events from different threads may be interleaved. To enable re-creation of events at the receiving end, each log entry contains an external reference to the executing thread.

3.3 Logging Monitor Exit

One of our applications of **JSpy** is run time analysis of programs for deadlocks and data races. This requires logging the taking and releasing of synchronization locks. At the JVM level, locks are obtained when a synchronized method or `monitorenter` instruction is executed. As stated in the JVM specification [11]: “Normally, a compiler for the Java programming language ensures that the lock operation implemented by a `monitorenter` instruction executed prior to the execution of the body of the synchronized statement is matched by an unlock operation implemented by a `monitorexit` instruction whenever the synchronized statement completes, whether completion is normal or abrupt.” Thus instrumenting all `monitorenter` and `montitorexit` correctly tracks the number of locks held by a thread on an object relating to `synchronized` statements. However if a synchronized method abruptly terminates, then the lock obtained on entry to the method is released by the JVM, but there is no way to instrument the bytecode to record the release of the lock. The not-to-elegant fix to this problem is to modify the body of each synchronized method to surround the whole body with as `try` block that catches all throwables, logs the release of the lock, and rethrows the exception.

4 Writing Monitors

The Java PathExplorer, **JPaX**, in which **JSpy** has been applied, provides a general framework for writing program execution monitors. One kind of monitoring is conformance checking against temporal logic formulas. Temporal logic formulas are assertions about a sequence of states, intended to describe properties of programs [12]. Formulas of temporal logic are constructed from atomic formulas using Boolean operators and temporal operators such as $\diamond$ (meaning eventually in some future state). In propositional temporal logic the atomic formulas are predicates on states. Using techniques such as model checking, a program may be verified with respect to a temporal logic formula, by showing the all possible execution traces of the program satisfy the formula. Run time analysis checks that one execution of the program satisfies the formula. This section examines practical considerations of mapping the meaning of satisfaction in temporal logic to Java programs. For propositional temporal logic this concerns how to define sequences of states and predicates over such states. The “obvious” correspondence is that a state is the computation state of the JVM, i.e. the state of the heap, invocation stacks and instruction counters for threads, etc. Transition to a new state is achieved by execution of a JVM instruction. Immediately one practical problem is clear: state transition at the level of individual instructions is too fined grained. If at each instruction cycle of the JVM a log entry is transmitted then the instrumented target will execute many times slower than the un-instrumented target.

Our objective of limiting the impact of instrumentation on program execution time leads to our design decision to check satisfaction of temporal formula off-line. We call the off-line checker an *observer*. The design of **JPaX** includes a *dispatcher* which routes the event stream to one or more such observers. This architecture implies that atomic predicates are restricted to the information about the JVM state obtainable from the **JSpy** log stream. In fact, it is desirable to define within each observer a data structure which records an *observer state* that is updated by the log stream. Thus, the observer state is an abstraction of the state of the JVM. The predicates of the propositional formula are viewed as predicates over the observer state, rather than the JVM state.

Propositional temporal logic has expressibility limitations that impact it's practical application. For example, it is impossible with propositional temporal logic to count the number of times a predicate is true within a computation. However one can add components to the observer state that perform such counting and reference these components in the definition of propositional predicates. Thus a temporal logic observer is defined by defining

- an instrumentation specification,
- an observer state,
- how the observer state is updated by the event stream,
- predicates over the observer state,
- a temporal logic formula over the predicates

We have defined a simple extension to propositional temporal logic that introduces a binding operator. The binding operator binds a variable in a formula to a component of the observer state. Predicates within the scope of the binding operator may be parameterized by the variable.

JPaX furthermore includes an observer that can detect the potential of a program to contain a deadlock or data race. Such an observer requires an event stream that includes taking and release of locks and updates to shared objects. The state of the observer consists of data structures such as lock graphs. Details can be found in [5,6,10].

5 Comparison with Other Approaches

The intended application for **JSpy** is **JPaX** and so our primary concern designing **JSpy** was insuring that application was well supported. Logging is a classic application of aspect-oriented programming, but AOP systems like AspectJ do not have the needed flexibility for defining code insertion points required by **JPaX**. Furthermore AspectJ modifies Java source, rather than bytecode a significant usability concern, and one that inhibits deadlock and data race analysis. Java debugging an profiling interfaces are either to slow or too restricted. Modification of the JVM is another possible approach, but that is a significant undertaking and one that prevents adoption barriers.

6 Conclusion

We presented an instrumentation package and showed how it is used within our runtime analysis system **JPaX**. We are planning to investigate minimizing the impact of instrumentation on the execution time of a program, by instrumenting different parts of the code for different test executions and then correlating the results. We also plan to combine dynamic analysis with static analysis.

References

1. 1st CAV Workshop on Runtime Verification (RV'01). In K. Havelund and G. Roşu, editors, *Proceedings of Runtime Verification (RV'01)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
2. 2nd CAV Workshop on Runtime Verification (RV'02). In K. Havelund and G. Roşu, editors, *Proceedings of Runtime Verification (RV'02)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2002.
3. K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.
4. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, and W. Visser. Experiments with Test Case Generation and Runtime Analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003, LNCS 2589, Taormina, Italy*, pages 87–107. Springer, March 2003.
5. C. Artho, K. Havelund, and A. Biere. High-Level Data Races. In *VVEIS'03, The First International Workshop on Verification and Validation of Enterprise Information Systems*, April 2003. Angers, France.
6. S. Bensalem and K. Havelund. Reducing False Positives in Runtime Analysis of Deadlocks. Submitted for publication, October 2002.
7. S. Cohen. Jtrek. Developed by Compaq.
8. M. Dahm. BCEL. Compaq, `http://jakarta.apache.org/bcel`.
9. T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming. *Comm. ACM*, 44(10):29–32, 2001.
10. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science.
11. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1999.
12. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.