ICSE'12, IEEE, pp 1427-1430. 2012

JavaMOP: Efficient Parametric Runtime Monitoring Framework

Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, U.S.A.
{*djin3, pmeredit, clee83, grosu*}*@illinois.edu*

***Abstract*—Runtime monitoring is a technique usable in all phases of the software development cycle, from initial testing, to debugging, to actually maintaining proper function in production code. Of particular importance are parametric monitoring systems, which allow the specification of properties that relate objects in a program, rather than only global properties. In the past decade, a number of parametric runtime monitoring systems have been developed. Here we give a demonstration of our system, JavaMOP. It is the only parametric monitoring system that allows multiple differing logical formalisms. It is also the most efficient in terms of runtime overhead, and very competitive with respect to memory usage.**

***Keywords*-runtime verification; runtime monitoring; testing; debugging; aspect-oriented programming;**

I. Introduction

Monitoring executions of a system against expected properties plays an important role not only in system development stages, e.g., debugging, testing, but also in the deployed system as a mechanism to increase system reliability or security. Monitoring a program execution generates an execution trace consisting of events that the user is interested in. When an execution trace validates/violates the given property, appropriate actions according to the purpose of the monitor will be triggered.

Parametric properties are properties with free variables, allowing one to describe not only global behaviors but also behaviors of objects. Although it gives more expressiveness, it is not trivial to monitor parametric properties efficiently. There can be a huge number of parameter instances during monitoring. For example, in Java, it is not uncommon to see hundreds of thousands of iterators during one execution of a program, which leads to hundreds of thousands of parameter instances in a specification about iterators. There have been numerous approaches such as [1], [2], [3], [4], [5], [6], [7] to build effective and efficient parametric monitoring solutions for different applications. However, many monitoring systems follow a formalism-dependent approach, that is, they hardwire their parametric specification formalisms, as a feasible solution to this, limiting expressiveness and leading to inefficient monitoring; note that some properties can be monitored more efficiently using a different formalism. Figure 1 lists a number of other monitoring systems and the logical formalism they support. Note that all have some hardwired formalism.

Approach	Language	Logic	Scope	Mode	Handler
Hawk [8]	Java	Eagle	global	inline	violation
J-Lo [9]	Java	ParamLTL	global	inline	violation
Jass [10]	Java	assertions	global	inline	violation
JavaMaC [11]	Java	PastLTL	class	outline	violation
jContractor [12]	Java	contracts	global	inline	violation
JML [13]	Java	contracts	global	inline	violation
JPaX [1]	Java	LTL	class	offline	violation
P2V [14]	C, C++	PSL	global	inline	validation/ violation
PQL [5]	Java	PQL	global	inline	validation
PTQL [6]	Java	SQL	global	outline	validation
Spec# [15]	C#	contracts	global	inline/ offline	violation
RuleR [16]	Java	RuleR	global	inline	violation
Temporal Rover [2]	*several*	MiTL	class	inline	violation
Tracematches [4]	Java	Reg. Exp.	global	inline	validation

Figure 1. Runtime Monitoring Breakdown

Our approach slices an execution trace according to parameter instance, then checks each slice by a monitor dedicated to the slice. In this way, a monitor can be from any formalism and the framework can focus on efficient parametric slicing no matter what formalism is being used. The challenge of this approach that there can be a huge number of monitors, has been overcome by many techniques. By ignoring parameter instances that can never reach the target states [17], we can reduce the number of monitors to create, reducing runtime and memory overhead greatly. Also, some monitors can become unnecessary during execution because objects necessary for a policy violation can be garbage collected. We detect such unnecessary monitors and eliminate them efficiently [18]. In this way, the number of monitors to update upon an event decreases and monitors take less memory, resulting in better runtime and memory performance.

Supported by NSF grant CCF-0916893.

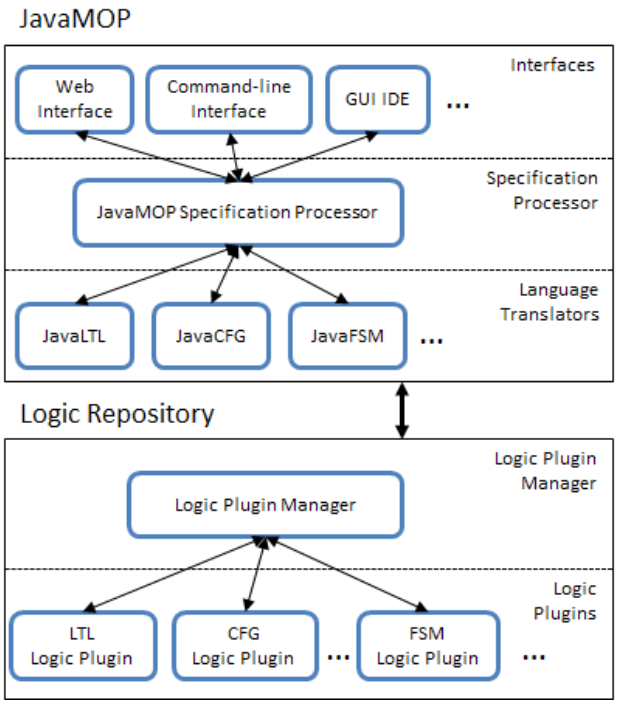

Figure 2. JavaMOP Architecture

II. JavaMOP

JavaMOP [19] is an instance of the generic MOP framework specific to the Java programming language. It allows concise descriptions of parametric properties using a combination of event specifications using an extension of AspectJ [20] as well as properties specified over these events. From these specifications, JavaMOP generates AspectJ code for monitoring, which is weaved into the target program by any AspectJ compiler, such as the standard AspectJ compiler ajc. In this way, the generated monitoring code observes the program, catches the events defined by a specification, and checks whether the program is compliant to the given specification. When a specification is validated or violated, user-defined actions, called handlers, are executed. User-defined actions can be any Java code from logging to runtime recovery. The ability to supply actual recovery code in the event of a violated specification allows JavaMOP-generated monitors to actually *enforce* a specification within a program.

JavaMOP is distinguished by several features which make it one of the best runtime monitoring systems. First, JavaMOP supports efficient formalism-independent parametric monitoring [17]. While other tools have some restrictions in their parametric monitoring or hardwire formalisms in order to support parametric monitoring efficiently, JavaMOP has no restrictions in parametric monitoring and is completely logical formalism agnostic. JavaMOP already has support for finite state machines (FSM), extended regular expressions (ERE), context-free grammars (CFG), past time (specific) linear temporal logic (PTLTL), a linear temporal logic supporting both future and past operators simultaneously (LTL), and past time linear temporal logic with calls and returns (PTCaRet). JavaMOP is unique[1] among runtime monitoring systems in supporting efficient parametric context-free properties, and is the only monitoring system of any kind that supports LTL specifications with simultaneous future and past operators. Additionally, JavaMOP can be extended to support more logical formalisms using its plugin interface. Since we believe that there is no ultimate formalism that every property can be expressed concisely, users will likely use different formalisms, depending on what properties they want to express. Currently, JavaMOP[2] has the best runtime performance of any monitoring system, while maintaining competitive memory performance. Several years of research [7], [21], [17], [18] have focused on improving its performance, resulting in only 15% average runtime overhead on the DaCapo benchmark [22].

Figure 2 shows the architecture of JavaMOP. JavaMOP communicates with the Logic Repository which generates platform-independent monitoring pseudo-code for the given property. All logical formalism plugins are contained within the Logic Repository, which is a standalone program that can be used with other instances of the MOP framework[3]. The JavaMOP component has several translators to interpret pseudo-code from the Logic Repository into AspectJ monitoring code. JavaMOP provides several interfaces including command-line and web interfaces, and it can be extended as well, for other usages.

Since JavaMOP relies on an external AspectJ compiler for weaving monitoring code, JavaMOP only allows events which can be defined in the standard AspectJ. Additionally, it cannot do any weaving-time optimization. For the broader scope of events and target code specific optimization, we leave integrating JavaMOP into an AspectJ compiler as a future work.

The installer and the source distribution of JavaMOP command-line interface are available from the JavaMOP website, http://javamop.com. The website also contains the web interface to JavaMOP, detailed JavaMOP syntax, instructions for use, and experimental results.

III. Demonstration

In the accompanying video, we demonstrate three JavaMOP properties with different formalisms, how to instrument example programs with the generated monitoring code, and the results of monitoring the example code. We monitor

[1]RV-Monitor, a product under development by Runtime Verification, Inc. also supports these, but it is a derivative of JavaMOP.

[2]And its RV-Monitor derivative.

[3]A C instance of MOP is forthcoming.

```
SafeEnum(Vector v, Enumeration e) {
  event create after(Vector v)
    returning(Enumeration e) :
         call(Enumeration Vector+.elements())
      && target(v) {}
  event updatesource after(Vector v) :
      (call(* Vector+.remove*(..))
    || call(* Vector+.add*(..))
    || call(* Vector+.clear(..))
    || call(* Vector+.insertElementAt(..))
    || call(* Vector+.set*(..))
    || call(* Vector+.retainAll(..)))
    && target(v) {}
  event next before(Enumeration e) :
       call(* Enumeration+.nextElement())
    && target(e) {}

  ere : create next* updatesource+ next
  @match {
    System.out.println("improper enumeration usage");
    __RESET;
  }
}
```

Figure 3. SafeEnum Property in ERE

```
SafeLock(Lock l, Thread t) {
  event acquire before(Lock l, Thread t):
    call(* Lock.acquire()) && target(l)
      && thread(t) {}
  event release before(Lock l, Thread t):
    call(* Lock.release()) && target(l)
      && thread(t) {}
  event begin before(Thread t) :
    execution(* *.*(..)) && thread(t)
      && !within(Lock+) {}
  event end after(Thread t) :
    execution(* *.*(..)) && thread(t)
      && !within(Lock+) {}

  cfg : S -> S begin S end |
        S acquire S release | epsilon
  @fail {
    System.out.println("improper lock usage");
  }
}
```

Figure 4. SafeLock Property in CFG

our example programs against three well-known properties that have been discussed in other works [21], [4], [7]. We rewrote the properties into three different formalisms for demonstration purposes. For the last property, we also demonstrate how we can provide a recovery action so that the property is actually enforced, rather than simply checked.

- SafeEnum (ERE): Ensures that one does not modify a Collection while its Enumeration is in use;
- SafeLock (CFG): Ensures that each method in each thread releases each Lock as many times as the method acquires the lock;

```
HasNext(Iterator i) {
  event hasnexttrue after(Iterator i)
    returning(boolean b) :
      call(* Iterator.hasNext())
      && target(i) && condition(b) {}
  event hasnextfalse after(Iterator i)
    returning(boolean b) :
      call(* Iterator.hasNext())
      && target(i) && condition(!b) {}
  event next before(Iterator i) :
      call(* Iterator.next()) && target(i) {}

  ltl: [](next => (*) hasnexttrue)
  @violation {
    System.out.println("improper iterator usage");
  }
}
```

Figure 5. HasNext Property in LTL

- HasNext (LTL): Ensures that one does not use the next element in an Iterator without checking for the existence of a next element.

In the first specification, SafeEnum (Figure 3), we have three events: create, updatesource, and next. Once we see any updatesource event, there should be no following next event. Our first example program follows the specification and no errors are reported by the monitoring code. Our second program violates the specification, and the monitoring code detects it and signals an error.

In the second specification, SafeLock (Figure 4), each acquire and release event should be paired in the same method. Any mismatched acquire or release is considered to be a violation to the property. Note that this property cannot be expressed using regular patterns. We again show both validation and violation of the property with example programs.

In the last specification, HasNext (Figure 5), there are three events: hasnexttrue, hasnextfalse, and next. The events hasnexttrue and hasnextfalse represent the same method call but different return values. The property states that every time a next event is seen, there should be a hasnexttrue event immediately preceding it. This time we show only a violating program, and we show how recovery from the violation can be performed by checking that another element is present before allowing the call to the next() method by using an around pointcut (not pictured).

IV. Conclusion

Parametric monitoring is useful at all stages during the software development cycle, either to aid in the detection of errors, or to recover from errors gracefully. Many systems have been developed to allow the monitoring of parametric properties, but they all have fixed formalisms. JavaMOP allows the user to use whatever logical formalism with which they feel most comfortable, in fact, they may even add new formalisms if they desire. Not only is JavaMOP not fixed in

logical formalism, but it is also the most efficient monitoring system in terms of runtime overhead, and very competitive in terms of memory efficiency.

With our tool demo, we showcase the JavaMOP tool. We demonstrate three different properties. For two of the properties, we show two example programs, in each case one that causes a violation of the property and one that does not. For the third property, we show an example program that violates the property, and then show how JavaMOP's recovery actions allow for ensuring that the violation can be avoided entirely.

The demonstration video is available at:
http://fsl.cs.uiuc.edu/JavaMOPDemo.html.

REFERENCES

[1] K. Havelund and G. Roşu, "Monitoring Java programs with Java PathExplorer," in *Runtime Verification (RV'01)*, ser. ENTCS, vol. 55, no. 2. Elsevier, 2001.

[2] D. Drusinsky, "The Temporal Rover and the ATG Rover," in *Model Checking and Software Verification (SPIN'00)*, ser. LNCS, vol. 1885. Springer, 2000, pp. 323–330.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-Based Runtime Verification," in *VMCAI'04*, ser. LNCS, vol. 2937. Springer, 2004, pp. 44–57.

[4] P. Avgustinov, J. Tibble, and O. de Moor, "Making trace monitors feasible," in *Object Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*. ACM, 2007, pp. 589–608.

[5] M. Martin, V. B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Object Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*. ACM, 2005, pp. 365–383.

[6] S. Goldsmith, R. O'Callahan, and A. Aiken, "Relational queries over program traces," in *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*. ACM, 2005, pp. 385–402.

[7] F. Chen and G. Roşu, "MOP: An efficient and generic runtime verification framework," in *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*. ACM, 2007, pp. 569–588.

[8] M. d'Amorim and K. Havelund, "Event-based runtime verification of Java programs," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[9] E. Bodden, "J-LO, a tool for runtime-checking temporal assertions," Master's thesis, RWTH Aachen University, 2005.

[10] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass-Java with Assertions," in *Runtime Verification*, ser. ENTCS, vol. 55.2. Elsevier, 2001.

[11] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *Europoean Conference on Real-Time Systems (ECRTS'99)*, 1999.

[12] P. Abercrombie and M. Karaorman, "jContractor: Bytecode instrumentation techniques for implementing DBC in Java," in *Runtime Verification*, ser. ENTCS, vol. 70.4. Elsevier, 2002.

[13] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs, "JML: notations and tools supporting detailed design in Java," in *Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00)*. ACM, 2000, pp. 105–106.

[14] H. Lu and A. Forin, "The design and implementation of P2V, an architecture for zero-overhead online verification of software programs," Microsoft Research, Tech. Rep. MSR-TR-2007–99, 2007.

[15] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview," in *CASSIS'04*, ser. LNCS, vol. 3362. Springer, 2004, pp. 49–69.

[16] H. Barringer, D. Rydeheard, and K. Havelund, "Rule systems for run-time monitoring: from EAGLE to RULER," *J. Logic Computation*, November 2008.

[17] F. Chen, P. Meredith, D. Jin, and G. Roşu, "Efficient formalism-independent monitoring of parametric properties," in *Automated Software Engineering (ASE'09)*. IEEE, 2009, pp. 383–394.

[18] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu, "Garbage collection for monitoring parametric properties," in *Programming Language Design and Implementation (PLDI'11)*. ACM, 2011, pp. 415–424.

[19] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *International Journal on Software Techniques for Technology Transfer*, 2011.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *European Conference on Object Oriented Programming (ECOOP'01)*, ser. LNCS, vol. 2072. Springer, 2001, pp. 327–353.

[21] P. Meredith, D. Jin, F. Chen, and G. Roşu, "Efficient monitoring of parametric context-free patterns," in *Automated Software Engineering (ASE'08)*. IEEE, 2008, pp. 148–157.

[22] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*. ACM, 2006, pp. 169–190.