Flexible and Extensible Runtime Verification for Java

Chengcheng Xiang[1], Zhengwei Qi[1], and Walter Binder[2]

[1]Shanghai Jiao Tong University, Shanghai, China
Email:{xiangchengcheng,qizhwei}@sjtu.edu.cn
[2]Università della Svizzera italiana (USI), Switzerland
Email:{walter.binder}@usi.ch

***Abstract*—Runtime verification validates the correctness of a program's execution trace. Much work has been done on improving the expressiveness and efficiency of runtime verification. However, current approaches require static deployment of the verification logic and are often restricted to a limited set of events that can be captured and analyzed, hindering the adoption of runtime verification in production systems. A popular system for runtime verification in Java, JavaMOP (Monitor-Oriented Programming in Java), suffers from the aforementioned limitations due to its dependence on AspectJ, which supports neither dynamic weaving nor an extensible join-point model. In this paper, we extend the JavaMOP framework with a dynamic deployment API and a new MOP specification translator, which targets the domain-specific aspect language DiSL instead of AspectJ; DiSL offers an open join-point model that allows for extensions. A case study on lambda expressions in Java8 demonstrates the extensibility of our approach. Moreover, in comparison with JavaMOP using load-time weaving, our implementation reduces runtime overhead by 21%, and heap memory usage by 16%, on average.**

Keywords*—*Runtime verification; Monitor-Oriented Programming (MOP); dynamic program analysis; dynamic deployment

I. Introduction

Runtime verification [1], [2] is a method that dynamically checks specific properties of an executing system both in testing and production environments. Compared with traditional verification approaches, such as model checking [3] and automated theorem proving [4], runtime verification reduces the state space by concentrating on the actual execution trace and eliminates the fallibility of formally modeling a system. In recent years, a lot of research has aimed at making runtime verification a practical way to improve program reliability [1], [5], [6].

A lot of techniques and tools have been developed for runtime verification of Java programs. Early research, including Java-MaC [7] and Hawk/Eagle [5], is focused on developing expressive logics for property description. Recently, Java-MOP [6] effectively reduces the runtime overhead thanks to an efficient management of monitors. Moreover, static program analysis techniques are used to reduce the amount of inserted instrumentation code [8].

However, a lack of flexibility and extensibility has prevented these techniques from becoming widely used in practice. Flexibility is important for two reasons. On the one hand, runtime verification systems may introduce a significant overhead of more than 100% when monitoring multiple properties simultaneously [9]. In some cases, such overhead may be inevitable, because checking multiple properties simultaneously may need to monitor a large number of events. Hence, it is necessary to verify properties sequentially, implying that code for event capture needs to be deployed and undeployed dynamically. On the other hand, as dynamic code evolution has been a timesaving way for development, property checkers should also be able to get updated dynamically. Moreover, since most runtime verification tools for Java, such as Tracematches [10] and JavaMOP [6], use AspectJ [11] as their instrumentation back-end, the categories of events are restricted to the AspectJ pointcuts, which can only be extended by modifying the AspectJ compiler. As shown e.g. in cite[12], the AspectJ join-point model is not well suited for dynamic program analysis.

DOI reference number: 10.18293/SEKE2015-117

In this paper, we present a flexible and extensible runtime verification framework for Java, MOP-DiSL. Our approach is based on JavaMOP [6], a framework for Monitoring-Oriented Programming for Java, and on DiSL [12], a domain-specific language for dynamic program analysis based on bytecode instrumentation. Our framework translates the MOP specification into DiSL code, and a new deployment API allows for flexible (un)deployment of instrumentation code at runtime. Extensibility of event categories is achieved through DiSL's open join-point model, which we demonstrate with a case study on Java8 lambda expressions.

This paper makes the following contributions:

- We present MOP-DiSL, a novel runtime verification framework for Java that offers flexible dynamic (un)deployment and extensibility in terms of event types that can be captured.
- We conduct a case study on lambda expression-related properties in Java8 programs, and we add several new pointcuts to show the extensibility of MOP-DiSL.
- We evaluate MOP-DiSL by verifying four properties with the DaCapo benchmarks [13], showing that MOP-DiSL introduces significantly less runtime overhead and consumes less heap memory than JavaMOP with load-time weaving.

This paper is structured as follows. Section II provides background information on JavaMOP, AspectJ, and DiSL. Section III gives a motivating example. Section IV presents the design and some implementation details of our framework. Section VI evaluates our framework in comparison with Java-MOP. Finally, Section VII concludes.

```
List<Integer> l1 = new ArrayList<>();
List<Integer> l2 = new ArrayList<>();
...
Iterator<Integer> itr = l1.iterator();
while (itr.hasNext()){
   l2.add(itr.next()*2);
}
```

Listing 1: Iterator example

```
l1.parallelStream()
  .map(e -> e*2)
  .forEach(e -> l2.add(e));
```

Listing 2: Stream example

II. Background

A. JavaMOP

JavaMOP is an implementation of Monitor-Oriented Programming (MOP) that enables runtime verification on the Java platform. The implementation consists of two parts: a translator and a set of runtime libraries. The translator parses specifications of properties in the form of finite state machines (FSM), context-free grammars (CFG), extended regular expressions (ERE), and other logical formalism, and generates AspectJ code to monitor events. Using AspectJ to weave code restricts the flexibility of JavaMOP and the categories of events it can get. However, adding new event types to MOP specifications requires both extensions to the JavaMOP translator and to the AspectJ compiler.

Many optimization techniques have been proposed to improve the efficiency of the runtime libraries of JavaMOP. [6] adopts centralized and decentralized indexing algorithms to optimize the lookup process of monitors. In order to efficiently reclaim monitor instances that are bound to parameter objects, [14] proposes a lazy garbage-collection (GC) strategy. Other than the aforementioned generic techniques, optimizations are also performed for specific property patterns [15]. According to [16], the verification code only causes an average runtime overhead of 15% on the DaCapo benchmark. However, the overhead for simultaneously monitoring multiple properties remains prohibitive in practice [9], indicating that there is need for a more flexible approach to runtime verification.

B. AspectJ

Used by JavaMOP for event definition and instrumentation, AspectJ [11] is an aspect-oriented programming (AOP) extension to the Java programming language. AspectJ adopts pointcuts to select join points, which are execution points in a program flow, and advice to define the actions to be executed before, after, or around each join point. Common AspectJ implementations weave advice in two ways: pre-load weaving and load-time weaving. Dynamic AOP, supported by tools such as AspectWerkz [17], Prose [18], and HotWave [19], [20], also enables runtime weaving and runtime adaption of the aspect code. Such techniques may endow runtime verification with more flexibility; however, there are still restrictions on the extensibility of event categories.

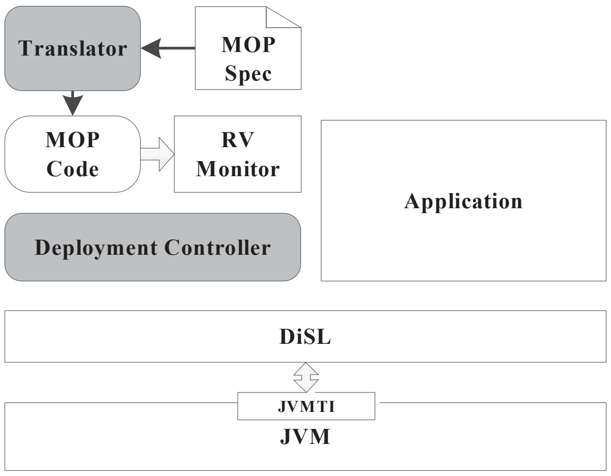


Figure 1: Framework Overview.

C. DiSL

As a tool for Java bytecode instrumentation, DiSL [12] is distinguished with several features. First, unlike AspectJ, DiSL supports an open join-point model, which means that any code region can be marked as a join point to trigger an event. Second, DiSL offers comprehensive bytecode coverage when weaving, including the Java core class library and dynamically generated code. Moreover, DiSL has been enhanced with the ability of dynamically deploying and undeploying analysis code [21]. Although DiSL possesses the necessary flexibility and extensibility for runtime verification, it lacks expressiveness to describe properties and efficient management of monitor instances.

III. Motivating Example

In this section, we present an example to show the necessity for flexibility and extensibility in runtime verification frameworks.

Listing 1 and Listing 2 demonstrate two code segments with the same function: doubling the integers in l1 and appending the outcome to l2. The difference resides in the way they realize it. In Listing 1, an iterator is used to traverse l1, while the code in Listing 2 utilizes a parallel stream (introduced in Java8) to eventually accelerate the process.

When runtime verification is applied to these two code segments, different properties are of concern. For Listing 1, as an iterator is created, we may be concerned about the HasNext property, which means hasNext() should always be called before the execution of next() on an iterator. For Listing 2, it is pointless to check the HasNext property, because the parallel stream internally traverses the list. Instead, since the statements in the lambda expressions are executed concurrently, it becomes necessary to verify whether they are thread-safe. In this example, the method call l2.add() is not safe.

Given a dynamic software update substituting Listing 1 with Listing 2 in a running system that should not be restarted, the properties for runtime verification should also be replaced dynamically. The lambda expressions pose an additional challenge, because there is no pointcut in current AspectJ to

```
event updatesource after(Collection c) :
     (call(* Collection+.remove*(..))
     || call(* Collection+.add*(..)) ) &&
         target(c) {}
```

Listing 3: Event example

```
@After(marker = CollectionRemoveMarker.class)
public void Updatesource0 (DynamicContext dc,
              ArgumentProcessorContext apc) {
  Collection c =
      (Collection)apc.getReceiver();
  RuntimeMonitor.updatesourceEvent(c);
}

@After(marker = CollectionAddMarker.class)
public void Updatesource1 (DynamicContext dc,
              ArgumentProcessorContext apc) {
  Collection c =
      (Collection)apc.getReceiver();
  RuntimeMonitor.updatesourceEvent(c);
}
```

Listing 4: DiSL example

capture the event that a lambda expression is called in a stream stage. More details about lambda expressions are discussed in Section V. These challenges demonstrate the need for flexibility and extensibility of runtime verification systems.

IV. Framework Design and Implementation

In this section, we first give an overview of the framework architecture, and then present the design and implementation details of each part.

A. Overview

Figure 1 depicts an overview of MOP-DiSL. In the beginning, the translator generates MOP code in Java, with inputs of MOP specifications. Then the deployment controller deploys the MOP code dynamically (*i.e.*, instruments an application with the MOP code). The MOP code will capture sensitive events and generate corresponding monitors, which are managed by RV-Monitor, a runtime library from JavaMOP. The whole framework is based on DiSL, which relies on the JVMTI, a standard tool interface for the JVM. Hence, a high degree of portability of MOP-DiSL is achieved.

Flexibility and extensibility are achieved through the deployment controller and the translator separately. On the one hand, the deployment controller instruments, updates, and removes bytecode for event capture without pausing or restarting the target application. On the other hand, the translator generates DiSL code as instrumentation back-end, gaining extensibility from DiSL's open join-point model.

B. Translator

The translator extends the original JavaMOP translator to generate DiSL code for instrumentation (instead of AspectJ). The JavaMOP translator parses event descriptions with the definition syntax of AspectJ advice, as shown in Listing 3, and the advice definitions are copied to the final aspect for event capture. Our translator takes the same input, but generates DiSL code for instrumentation. Listing 4 demonstrates DiSL code generated from the event definition in Listing 3, with an annotation to define the event, and a method to express the reactions to the event.

The main challenge for translating advice to DiSL lies in the composition of pointcuts. AspectJ offers a set of primitive pointcuts, which can be combined by "&&" and "||" to describe complex events. DiSL also offer a group of basic markers to mark code regions as join points. However, there is no easy way to combine these markers, except extending the marker class. Our translator combines basic markers by generating new markers, *e.g.*, CollectionRemoveMarker and CollectionAddMarker, which inherit from a basic MOP marker. To shorten the generated code, the base MOP marker is modeled as a pipeline of filters, each for one basic pointcut. An extended marker overrides several filters to express "&&" of these filters, while "||" will be translated into different markers. In order to generate markers for "||", the translator first transforms the pointcut composition into a disjunctive normal form, and each clause will be converted to a marker.

New pointcuts can be easily added to define events by extending the translator and the base marker class. Since the marker class manipulates bytecode through ASM[1], a lightweight framework, it is very convenient to expand it with more filters, corresponding to more primitive pointcuts. We have added two pointcuts, *.i.e.*, lambdaDef and thisLambda, to capture events about lambda expressions. More details are presented in Section V.

C. Deployment Controller

Table I: Deployment Controller API

Interface	Description
deploy(property, scope)	Deploy MOP code of a property in a specific scope
undeploy(property, scope)	Undeploy MOP code of a property from a specific scope
redeploy(property, scope)	Update MOP code of a property in a specific scope

The deployment controller provides several interfaces to attain flexibility, which are listed in Table I. The three interfaces are declared with identical parameters, two string variables: property refers to a property name, such as *hasNext*, and scope refers to its deploying scope. The scope can be either a package name or a class name with wildcards support.

These interfaces are implemented based on the work in [21], which implemented dynamic bytecode instrumentation through the retransformClasses interface in JVMTI. In our current implementation, the controller interfaces are exposed through a network client, with which programmers can deploy or undeploy property monitors to a running system. However, these interfaces can also be called by optimized monitor programs which only enable property monitors when necessary.

V. Case Study

In this section, we present a case study on checking whether unsynchronized collections are modified in a parallel

[1]http://asm.ow2.org/

```
UnsafeForEach(Object capVar0, Consumer c){
   event create after(Object capVar0)
      returning(Consumer c) :
      lambdaDef()&& args(capVar0)){}
   event forEach before(Consumer c):
      call(* ParallelStream.forEach(..))
      && args(c) {}
   event update after(Consumer c, Object
       capVar0):
      (call(*
          (!SynchronizedCollection+).remove*(..))
      || call(*
          (!SynchronizedCollection+).add*(..)))
      && target(capVar0) && thisLambda(c){}
   ere : create forEach update
   @match {
      ...
   }
}
```

Listing 5: UnsafeForEach

stream, of which a violation instance is shown in Listing 2 as the motivation example.

Listing 5 demonstrates the property definition in the form of MOP specification. The property is named as UnsafeForEach with two parameters defined in the event definitions, *i.e.*, capVar0 and c. Three types of events are captured by the verification process, and ERE is adopted to express the targeting event trace. When a matching trace is detected, an error message is printed.

The first event is about creating a lambda expression, defined with a new pointcut, *i.e.* lambdaDef. In Oracle's Java8 implementation, lambda expressions are translated using the bytecode instruction invokedynamic. This instruction causes the lambda metafactory to be invoked and generates a lambda object implementing the Consumer functional interface. The lambdaDef pointcut selects all invokedynamic instructions, checks whether they call the lambda metafactory, exposes captured variables and the return value through the corresponding pointcuts args and returning. In this case, there is only one captured variable, *i.e.*, the collection object. However, more objects can be exposed by args(), if more variables are captured by the lambda expression.

The second event is calling the forEach method of a parallel stream instance. The forEach method takes a Consumer object as its parameter, and registers the object to the stream. The new ParallelStream pointcut is added for that a parallel stream is also an instance of Stream, which means it is impossible to pick out a parallel stream only with current type check.

The last event happens when an unsynchronized collection is updated in a lambda expression. The new pointcut thisLambda ensures that the updating executes in a lambda context, and exposes the lambda object c. It is worth noting that the lambda object cannot be exposed by AspectJ with the pointcut execution("*lambda*")&&this(c), since the lambda body is implemented as a static method and this() will return null. Another way is by the pointcut call("*lambda*")&&target(c), which also selects nothing because the lambda expression is called by the parallel stream, a class in the Java core library that cannot be woven by AspectJ. MOP-DiSL takes advantage

```
public class LambdaDefMarker extends
     MOPBaseMarker {
   public List<MarkedRegion> mainFilterPipe(
       AbstractInsnNode insn){
      List<MarkedRegion>
          regions=super.mainFilterPip(insn);
      filterLambda(regions, insn);
      return regions;
   }
   void filterLambda(regions,
       AbstractInsnNode insn){
      if (insn instanceof
          InvokeDynamicInsnNode){
         if (matchIndy(indy))
            regions.add(new
                MarkedRegion(insn, insn));
      }
   }
   //This will be overridden by the generated
       code
   boolean matchIndy(AbstractInsnNode insn){
      return false;
   }
}
```

Listing 6: LambdaDefMarker

of DiSL's ability to instrument the Java core library to capture the call event and expose the target object.

Adding these pointcuts consists of two steps: implementing new markers and extending the translator. Listing 6 shows the marker code for implementing lambdaDef pointcut. The new marker is introduced as a subclass of the base marker, adding a new filter process, *i.e.* filterLambda, to the main filter pipeline. What filterLambda does is simply checking the type of current instruction and calling matchIndy() to decide whether the instruction should be marked for weaving. The translator needs to be extended to generate a subclass of LambdaDefMarker with a proper overriding of the method matchIndy according the pointcut definition, which is much simpler than modifying the AspectJ weaver.

VI. Evaluation

In this section, we evaluate the runtime overhead and heap memory consumption of MOP-DiSL, and compare it with JavaMOP.

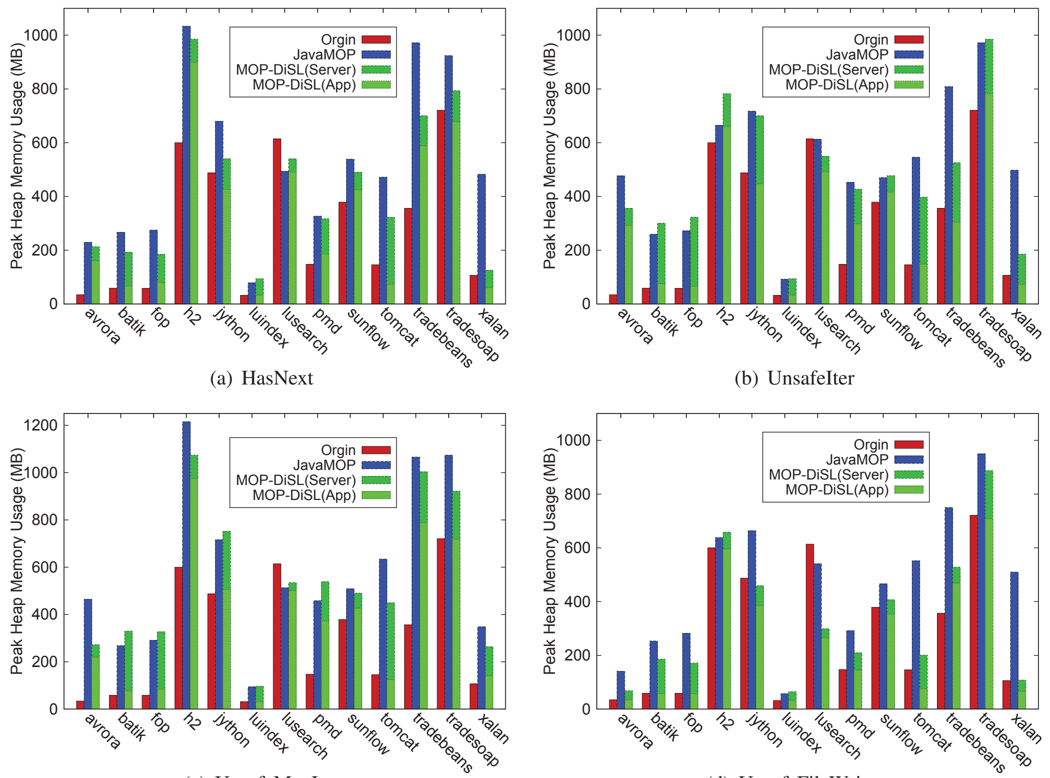
A. Experimental Settings

We set up the experimental environment on a Dell Optiplex 980 machine with 8GB memory and an Intel 3.20GHz i5 CPU. We use JDK 1.8 and AspectJ 1.8 for compilation on a 64 bits Debian 7 system. For AspectJ, load-time weaving is adopted, since DiSL uses dynamic weaving. DaCapo 9.12 [13], a popular benchmark suite for Java, is utilized to evaluate the runtime overhead and heap memory usage. The benchmark is run for 10 iterations and the result of the first iteration is also counted, as the code weaving only happens in the first iteration when classes are first loaded. Four well-known properties are verified in the evaluation:

1) HasNext: method hasNext() should be called before calling next() for an iterator;

Table II: Execution time and overhead percentage for JavaMOP and MOP-DiSL with different properties

Benchmark	origin	HasNext				UnsafeIter				UnsafeMapIter				UnsafeFileWriter			
		JavaMOP-LW		MOP-DiSL		JavaMOP-LW		MOP-DiSL		JavaMOP-LW		MOP-DiSL		JavaMOP-LW		MOP-DiSL	
	sec.	sec.	ovh.(%)	sec.	ovh.(%)	sec.	ovh.(%)	sec.	ovh.(%)	sec.	ovh.(%)	sec.	ovh.(%)	sec.	ovh.(%)	sec.	ovh.(%)
avrora	4.26	6.14	44.27	5.74	34.92	21.96	416.02	17.60	313.54	8.27	94.43	8.51	99.88	5.88	38.18	5.70	33.98
batik	2.25	2.75	22.19	2.59	15.08	2.96	31.64	3.35	48.82	2.87	27.46	2.92	29.78	3.90	73.36	3.50	55.80
fop	1.17	2.08	76.95	2.02	71.97	2.34	99.44	2.06	75.70	2.50	112.97	3.09	163.44	2.32	97.75	1.83	55.67
h2	6.94	7.84	13.01	7.45	7.38	7.86	13.33	7.10	2.38	8.53	22.91	10.13	46.00	8.80	26.89	8.62	24.29
jython	5.77	6.74	16.88	6.24	8.22	7.41	28.41	6.39	10.71	7.92	37.24	7.23	25.30	9.58	65.95	8.61	49.26
luindex	1.27	1.41	11.36	1.41	11.09	1.45	14.07	1.40	10.28	1.50	18.15	1.41	11.03	1.94	52.87	1.80	41.84
lusearch	2.35	2.62	11.27	2.65	12.49	3.04	29.16	2.82	19.79	2.60	10.39	2.79	18.49	3.41	44.94	3.14	33.52
pmd	4.11	5.95	44.83	5.66	37.81	8.64	110.28	7.79	89.67	10.27	149.89	8.25	100.87	6.12	48.94	5.48	33.27
sunflow	6.17	6.38	3.43	6.27	1.71	6.25	1.39	6.04	1.02	6.27	1.65	6.27	1.68	8.35	35.47	7.65	24.05
tomcat	3.83	5.27	37.63	4.17	8.93	5.48	43.20	4.89	27.65	5.66	47.77	4.79	25.02	6.41	67.52	4.81	25.77
tradebeans	6.84	18.53	171.18	17.77	160.06	14.83	117.02	13.55	98.28	21.13	209.12	20.22	195.83	9.51	39.10	9.14	33.67
tradesoap	12.31	18.19	47.81	17.99	46.17	24.25	97.00	23.59	91.62	25.32	105.69	26.35	114.07	14.88	20.87	14.98	21.71
xalan	2.91	3.12	7.15	3.06	5.17	3.59	23.47	3.55	21.97	3.18	9.42	3.40	16.81	3.55	22.25	3.94	35.56
geo.mean			23.63		16.55		38.11		26.38		34.95		36.21		44.20		34.41



(a) HasNext

(b) UnsafeIter

(c) UnsafeMapIter

(d) UnsafeFileWriter

Figure 2: Peak heap memory usage. MOP-DiSL (Server) refers to the memory consumption in the DiSL instrumentation server, which runs in another Java process. The actual runtime memory for application + analysis is noted as MOP-DiSL (App).

2) UnsafeIter: a Collection should not be updated when its iterator is accessed;
3) UnsafeMapIter: a Map should not be updated when its iterator is accessed;
4) UnsafeFileWriters: no writing is allowed after a file is closed.

The results are presented with JavaMOP-LW referring to JavaMOP using AspectJ load-time weaver. The results of eclipse benchmark in DaCapo are not presented because there is a bug when it is run on Java8[2]. We do not evaluate the performance of the lambda property for there is currently no standard benchmark with much use of the Java8 stream API.

B. Runtime Overhead

Table II displays the execution time and percent overhead of JavaMOP-LW and MOP-DiSL. On average (geometric mean), MOP-DiSL incurs less overhead than JavaMOP-

[2]http://mail.openjdk.java.net/pipermail/aarch64-port-dev/2014-February/000844.html

LW for property HasNext, UnsafeIter, and UnsafeFileWriter. MOP-DiSL achieves more overhead reduction for applications with higher overhead, which have more instrumented code executed, and the reason is that code instrumented by DiSL is more efficient than AspectJ. However, weaving code in an isolated instrumentation server, DiSL spends more time on communicating with the server so that more time on weaving. For UnsafeMapIter, there are more event types, which means more code needs to be instrumented, resulting that more communication with the sever is needed for DiSL. Consequently, MOP-DiSL causes similar overhead with JavaMOP-LW and more overhead for some benchmarks, *i.e.*, avrora, batik, fop, h2, lusearch, tradesoap, and xalan. Overall, MOP-DiSL causes 21% less runtime overhead than JavaMOP-LW.

C. Memory Consumption

Figure 2 illustrates the peak Java heap usage when the four properties are verified. The memory consumption of MOP-DiSL consists of the server part and the application part. The instrumentation server, which does code weaving, is run in a separated process and in theory can be run in another physical machine. We accumulate the memory usage mainly to show that even combining both application and instrumentation server, our approach still outperforms JavaMOP by 16% in terms of heap consumption (evaluated by geometric mean). The application part of MOP-DiSL consumes 54% lower Java heap than JavaMOP on average. This feature benefits the application process with less GC time. For some benchmarks, such as lusearch, JavaMOP and MOP-DiSL even cause lower peak heap consumption than the original application. This is mainly due to different GC behavior—with more memory usage in the runtime analysis, GC may be triggered more often and hence keep the memory consumption at a lower level.

VII. Conclusion

In this paper, we present a novel framework, MOP-DiSL, to achieve flexibility and extensibility in runtime verification for Java. A deployment API is designed for flexibility and a new MOP translator is devised with extensibility. We demonstrate a case study on adding event types to check properties associated with lambda expressions in Java8, which requires extensibility. Evaluation results show that our framework causes less runtime overhead and lower peak heap usage than JavaMOP with AspectJ load-time weaving. As a result, our framework achieves flexibility and extensibility with no more runtime and memory overhead, making runtime verification more practical in real-world settings.

Acknowledgment

This work is supported by NSFC (No. 61272101), National R&D Infrastructure and Facility Development Program (No. 2013FY111900), NRF Singapore CREATE Program E2S2, Sino-Swiss Science and Technology Cooperation (SSSTC), and Shanghai Key Laboratory of Scalable Computing and Systems.

References

[1] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[2] C. Kloukinas, G. Spanoudakis, and K. Mahbub, "Estimating event lifetimes for distributed runtime verification," in *SEKE*, pp. 117–122, 2008.

[3] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[4] M. Fitting, *First-order logic and automated theorem proving*. Springer Science & Business Media, 1996.

[5] M. d'Amorim and K. Havelund, "Event-based runtime verification of Java programs," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–7, ACM, 2005.

[6] F. Chen and G. Roşu, "Mop: an efficient and generic runtime verification framework," in *ACM SIGPLAN Notices*, vol. 42, pp. 569–588, ACM, 2007.

[7] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, "Java-MaC: a run-time assurance tool for Java programs," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 218–235, 2001.

[8] E. Bodden, P. Lam, and L. Hendren, "Clara: A framework for partially evaluating finite-state runtime monitors ahead of time," in *Runtime Verification*, pp. 183–197, Springer, 2010.

[9] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Serbanuta, and G. Rosu, "RV-Monitor: Efficient parametric runtime verification with simultaneous properties," in *Proceedings of the 14th International Conference on Runtime Verification (RV'14)*, LNCS, September 2014.

[10] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. De Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *ACM SIGPLAN Notices*, vol. 40, pp. 345–364, ACM, 2005.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP 2001*, pp. 327–354, Springer, 2001.

[12] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi, "DiSL: a domain-specific language for bytecode instrumentation," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pp. 239–250, ACM, 2012.

[13] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *ACM Sigplan Notices*, vol. 41, pp. 169–190, ACM, 2006.

[14] D. Jin, P. O. Meredith, D. Griffith, and G. Rosu, "Garbage collection for monitoring parametric properties," in *ACM SIGPLAN Notices*, vol. 46, pp. 415–424, ACM, 2011.

[15] P. Meredith and G. Rosu, "Efficient parametric runtime verification with deterministic string rewriting," in *Proceedings of 28th IEEE/ACM International Conference. Automated Software Engineering (ASE'13)*, p. NA, IEEE/ACM, May 2013.

[16] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "JavaMOP: Efficient parametric runtime monitoring framework," in *Proceeding of the 34th International Conference on Software Engineering (ICSE'12)*, pp. 1427–1430, IEEE, 2012.

[17] J. Bonér, "Aspectwerkz: Dynamic AOP for Java," in *Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, Citeseer, 2004.

[18] A. Nicoară and G. Alonso, "Dynamic AOP with Prose," in *1st International Workshop on Adaptive and Self-Managing Enterprise Applications*, pp. 125–138, 2005.

[19] A. Villazón, W. Binder, D. Ansaloni, and P. Moret, "Advanced runtime adaptation for Java," in *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, pp. 85–94, ACM, Oct. 2009.

[20] A. Villazón, W. Binder, D. Ansaloni, and P. Moret, "HotWave: creating adaptive tools with dynamic aspect-oriented programming in Java," in *ACM Sigplan Notices*, vol. 45, pp. 95–98, ACM, 2009.

[21] Y. Zheng, L. Bulej, C. Zhang, S. Kell, D. Ansaloni, and W. Binder, "Dynamic optimization of bytecode instrumentation," in *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pp. 21–30, ACM, 2013.