ICICIS'09, pp 72-77. 2009

Monitoring Oriented Programming - A Project Overview [*]

Feng Chen, Dongyun Jin, Patrick Meredith, Grigore Roşu
Department of Computer Science, University of Illinois at Urbana-Champaign
201 N Goodwin Ave
Urbana, IL 61801, USA
+1 217-244-7431
{fengchen,djin3,pmeredit,grosu}@cs.uiuc.edu

ABSTRACT

This paper gives a brief overview of Monitoring Oriented Programming (MOP). In MOP, runtime monitoring is supported and encouraged as a fundamental principle for building reliable software: monitors are automatically synthesized from specified properties and integrated into the original system to check its dynamic behaviors. When a specification is violated or validated at runtime, user-defined actions will be triggered, which can be any code from information logging to runtime recovery. Two instances of MOP are introduced: JavaMOP (for Java programs) and BusMOP (for monitoring PCI bus traffic). The architecture of MOP is discussed, and a brief explanation of parametric trace monitoring and its implementation is given. Finally, a comprehensive evaluation of JavaMOP attests to its efficiency, especially with respect to similar systems; BusMOP, in general, imposes 0 runtime overhead on the system it is monitoring.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification— *assert checker, class invariant, formal methods, programming by contract, reliability, validation.*

; D.2.4 [**Software Engineering**]: Testing and Debugging—*monitors, tracing.*

; D.3.3 [**Programming Languages**]: Language Constructs and Features—*constraints.*

General Terms

Reliability, Languages, Verification.

Keywords

monitoring oriented programming, runtime verification

*Supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, by the Microsoft/Intel funded Universal Parallel Computing Research Center at UIUC, and by several Microsoft gifts.

```
unsynchronized decentralized SafeLock(Lock l, Thread t)
  within Main.*{
   int acq_count, rel_count;
   event acq after(Lock l,Thread t) :
        call(* Lock.acquire()) && target(l) && thread(t)
        {++ acq_count;}
   event rel after(Lock l,Thread t) :
        call(* Lock.release()) && target(l) && thread(t)
        {++ rel_count;}
   cfg : S -> S acq S rel | epsilon
   @violation{
      System.out.println(acq_count + " acquires and "
        + rel_count + " releases at line " + __Loc);
   }
}
```

Figure 1: A JavaMOP Specification, SafeLock

1. INTRODUCTION

Runtime monitoring of requirements in software development can increase the reliability of the resulting systems. There is an increasingly broad interest in uses of monitoring in software development and analysis, as reflected, for example, by abundant approaches proposed recently [19, 1, 14], and also by the runtime verification (RV) and the formal aspects of testing (FATES) initiatives [15, 4, 17, 23, 16, 22] among many others.

Monitoring oriented programming (MOP) [11, 9, 12] is a generic monitoring framework that integrates specification and implementation by checking the former against the latter at runtime. In MOP, one specifies desired properties using definable formalisms and with actions to handle violations or validations of the specified property. MOP tools will then automatically synthesize monitors from property specifications and integrate them within the application together with user-provided handling code. Figure 1 shows an example specification of JavaMOP, an MOP tool for Java programs (see Section 3). Detailed explanation of the specification syntax can be found on the MOP website [3].

This specification describes a programming principal for safe operations of locks, namely that each method in each thread should release each lock as many times as it acquires it. It is composed of four parts. The first line is the header of the specification, starting with two modifiers, unsynchronized and decentralized; the former states that monitors for this property do not need to synchronize since they work within different threads and the latter chooses the way to index monitors for different parameter bindings, which is discussed in Section 4. An id for the specification is given after modifiers and followed by parameters of the property; in this example, two parameters are used, namely a lock object l and a thread instance t. At the end of the header is a within clause, which states that this property is monitored within the boundary of any method defined in the class Main (the wildcard * indicates "any"). The second part contains the declaration of two monitor variables: acq_count and rel_count. The third part of the specification contains event declarations. Two events are defined: acq for acquiring of a lock and rel for releasing of a lock. JavaMOP borrows the syntax of AspectJ [2] in event declarations. For example, the acq event is declared to occur "after" a function call to the acquire method of class Lock. Note that the target and thread clauses are used to bind parameters in the event. Each event also increments one of the monitor variables, which will be unique for each binding of the parameters.

The fourth part of the specification is a formal description of the desired property. As discussed in Section 2, MOP is specification-formalism-independent, and one may choose different logics to specify properties. In this example, the property description begins with cfg, meaning that a context free grammar (CFG) is used, and continues with a CFG pattern of declared events, which denotes that acquires and releases of a lock should match. The last part of the specification consists of handlers to execute in different states of the corresponding monitor, such as violation and validation. In Figure 1, the handler starts with @violation, defining the action, a simple warning report in this case, to execute when the violation of the specified CFG pattern is detected. The handler reports the number of acquires and releases of a given lock in a given thread, and the line number that the violation occurs on (given by the variable __LOC).

By a clear separation of monitor generation and monitor integration, MOP provides a fundamental and generic support for effective and efficient applications of runtime monitoring in different problem domains, and can be understood from at least three perspectives:

1. As a discipline allowing one to improve safety, reliability and dependability of a system by monitoring its requirements against its implementation at runtime;
2. As an extension of programming languages with logics. One can add logical statements anywhere in the program, referring to past or future states of the program. These statements are like any other programming language Boolean expressions, so they give the user a maximum of flexibility on how to use them: to terminate the program, guide its execution, recover from a bad state, add new functionality, etc.;
3. As a lightweight formal method. While firmly based on logical formalisms and mathematical techniques, MOP's purpose is not program verification. Instead, the idea is to avoid verifying an implementation against its specification before operation, by not letting it go wrong at runtime.

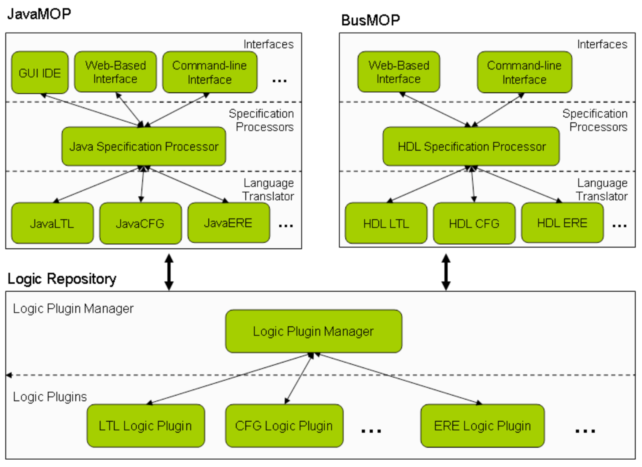


Figure 2: Architecture of MOP

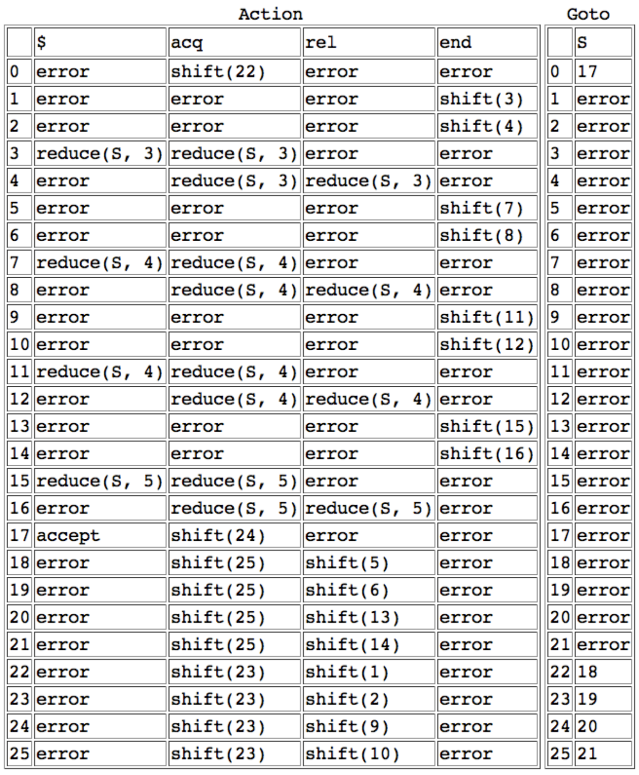
In the rest of the this paper, we first introduce the generic framework of MOP (Section 2), then discuss two language instances of MOP that have been implemented within the framework (Section 3). A specific topic of monitoring parametric properties is also discussed (Section 4) due to its significance in practice. Finally, we present an evaluation of JavaMOP, which testifies to its efficiency (Section 5). An evaluation of BusMOP (an MOP tool for monitoring PCI bus traffic) is unnecessary, as it has 0 runtime overhead.

2. MOP FRAMEWORK

Monitoring applications share some features, such as program instrumentation and monitor integration, even when they aim at different domains or goals. MOP separates monitor generation and integration and provides a generic, extensible framework for runtime monitoring, allowing one to instantiate MOP with specific programming languages and specification formalisms to support different domains. In this section, we focus on the overall architecture of MOP, into which one can easily plug in new specification formalisms.

2.1 Architecture

Figure 2 shows the architecture of MOP. There are two kinds of components in MOP, namely logic repositories and language clients. The logic repository, shown in the bottom of Figure 2, contains various logic plugins and a logic plugin manager component. The former is the core component to generate monitoring code from formulas written in a specific logic; for example, the PTLTL plugin synthesizes state machines from PTLTL formulas. The output of logic plugins is usually pseudocode and not bound to any specific programming language. This way, the essential monitoring generation can be shared by different instances of MOP using different programming languages. The logic plugin manager bridges the communication between the languages client and the logic plugin. More specifically, it receives the monitor generation request from the language client and distributes the request to an appropriate plugin. After the plugin synthesizes the monitor for the request, the logic plugin manager collects the result and sends it back to the language client. This way, one can easily add new logic plugins into the repository to support new specification formalisms



Action					Goto	
	$	acq	rel	end		S
0	error	shift(22)	error	error	0	17
1	error	error	error	shift(3)	1	error
2	error	error	error	shift(4)	2	error
3	reduce(S, 3)	reduce(S, 3)	error	error	3	error
4	error	reduce(S, 3)	reduce(S, 3)	error	4	error
5	error	error	error	shift(7)	5	error
6	error	error	error	shift(8)	6	error
7	reduce(S, 4)	reduce(S, 4)	error	error	7	error
8	error	reduce(S, 4)	reduce(S, 4)	error	8	error
9	error	error	error	shift(11)	9	error
10	error	error	error	shift(12)	10	error
11	reduce(S, 4)	reduce(S, 4)	error	error	11	error
12	error	reduce(S, 4)	reduce(S, 4)	error	12	error
13	error	error	error	shift(15)	13	error
14	error	error	error	shift(16)	14	error
15	reduce(S, 5)	reduce(S, 5)	error	error	15	error
16	error	reduce(S, 5)	reduce(S, 5)	error	16	error
17	accept	shift(24)	error	error	17	error
18	error	shift(25)	shift(5)	error	18	error
19	error	shift(25)	shift(6)	error	19	error
20	error	shift(25)	shift(13)	error	20	error
21	error	shift(25)	shift(14)	error	21	error
22	error	shift(23)	shift(1)	error	22	18
23	error	shift(23)	shift(2)	error	23	19
24	error	shift(23)	shift(9)	error	24	20
25	error	shift(23)	shift(10)	error	25	21

Figure 3: Monitor for the CFG pattern in Figure 1

in MOP without changing the language client. The language client hides the programming-language-independent logic repository and provides language specific support for applying MOP in particular programming languages. Every language client is usually composed of three layers: the bottom layer contains language translators that translate the abstract output of logic plugins into concrete code in a specific programming language; the middle layer is the specification processor that extracts formulas from the given property specification and then instruments the generated monitoring code into the target program; at last, the top layer provides usage interfaces to the user. Two examples of language clients are discussed in Section 3.

2.2 Logic Plugins

Every logic plugin implements and encapsulates a monitor synthesis algorithm for a particular requirements specification formalism, such as the past-time linear temporal logic (PTLTL) and the context free grammar (CFG) supported in the current MOP framework. The logic plugin accepts as the input a set of abstract events and a formula written in the underlying formalism and outputs an abstract monitor, usually a piece of pseudocode, which checks a trace of events against the given formula. For example, Figure 3 shows the monitoring code generated by the MOP CFG plugin from the CFG pattern in Figure 1. CFG monitors are contained within two tables: the Action and Goto tables, and every monitor has a stack used to track previous states of the monitor; the top of the stack is always the current state.

The Action table denotes what action should be taken when a given event is seen, in a given state. Four actions are possible: shift, reduce, accept, and error. When a shift action is encountered, the state denoted in the shift action is pushed onto the stack, in preparation for the arrival of the next event. Multiple reduce actions are possible in the presence of a given event, and the shift action is always the last action to occur. When an event causes a reduction, the number in the reduce denotes how many states to pop from the stack. The symbol in the reduce action is used as one of the two indices into the Goto table. The state left at the top of the stack after the necessary number of states is popped is the second index into the Goto table. The element found in the Goto table is similar to a shift action, in that it tells the monitor which state to push onto the top of the stack. This state is then used to check for further reductions and an eventual shift (or error). As one might imagine, an error occurs when the event received by the monitor is not valid in the current state; this action causes the monitor to run the violation handler. The accept action causes the monitor to run the validation handler.

Consider, again, Figure 3. In state 0, the initial state, if an acq event is seen, the monitor shifts to state 14. Because acq is the only valid start of a trace, we can see that, if rel is seen, it causes an error. The $ event denotes the guessed end of the trace. Anytime a reduction by $ is possible, $ is inserted in the stream, and reductions proceed with a copy of the current stack. This allows for a given run of a program to generate multiple validations. This algorithm is explained in detail in [20].

3. MOP INSTANCES

Every MOP instance needs to instantiate the MOP framework in four dimensions: 1) a specification language based on the problem domain, which is mainly related to how one defines events in the domain, 2) a target language for generated monitors, 3) supported specification formalisms, and 4) the handlers allowed in the specification. Two MOP instances have been implemented, namely JavaMOP and BusMOP. We expect to see more MOP instances in the future, as many problem domains can benefit from monitoring.

3.1 JavaMOP

JavaMOP is a development tool for Java, supporting several logical formalisms and a specification language to describe Java program behavior [11]. It compiles a specification into optimized monitoring code. This resulting code is AspectJ [18], and is program-independent. For example, a user can write a JavaMOP specification for a library. Then, JavaMOP generates monitoring code for this specification. This code can be applied to any program that uses the library.

In JavaMOP, an event corresponds to a pointcut that an AspectJ [2] compiler (such as ajc) can use to weave monitoring code into the original program. Pointcuts include function call, function return, function begin, function end, field assignment, object creation, and more complex ones with pointcut operators, which combine multiple simpler pointcuts. JavaMOP generates monitoring code for each pointcut to maintain monitoring state and check if the program conforms to the specification.

A system behavior can be described using one of several logical formalisms supported by JavaMOP. Logical formalisms include: ERE (extended regular expressions), FSM (finite state machines), CFG (context free grammars), PTLTL (past time linear temporal logic), FTLTL (future time linear temporal logic), and ptCaRet (past time linear temporal logic with calls and returns). A specification will be interpreted by the Logic Repository, a generic server used by all instances of MOP, and transformed into generic monitor code. JavaMOP translates this monitor to AspectJ code.

A user can write a handler in Java for each monitoring state. There can be more monitoring states than simple validation and violation, depending on logical formalism. A handler can be used for logging, recovering, blocking, or any other purpose. Since handlers are specified as arbitrary Java code, a user has a lot of freedom to achieve her purposes.

3.2 BusMOP

BusMOP [21] was designed to address the safety problem of third party consumer off-the-shelf components (COTS). The complexity of safety critical systems has grown to the point where the ability to use COTS in a safe manner is almost mandatory. Additionally, the vast majority of OS crashes in PCs are caused by faulty peripherals or their drivers. BusMOP answers both of these problems by allowing the specification and monitoring of properties with respect to PCI Bus traffic (soon to be expanded to other bus architectures).

In BusMOP, the events correspond to reads and writes of specified values to specified memory locations on the bus. PCI Bus interrupts are also allowed as events. The monitors, and the logic to glean events from bus traffic, are synthesized from hardware design language (HDL) code and programmed onto a field programmable gate array (FPGA), which is plugged into the PCI Bus.

BusMOP supports the FSM, ERE, and PTLTL plugins of MOP, with plans to add FTLTL soon. CFG has the problem of unbounded logic response time, which would cause the monitor to not meet timing constraints in some cases, and is thus not suitable for inclusion in BusMOP.

Handlers in BusMOP can be specified using arbitrary VHDL code. Several resources are provided for the user for use in handler code, such as serial output for logging, and the actual ability to write to the PCI Bus to perform recovery. Recovery actions in BusMOP require bus arbitration to undo deleterious actions of faulty peripherals or their drivers. This bus arbitration is the only possible overhead incurred by BusMOP, in cases of heavy Bus traffic. In the majority of systems, BusMOP can be used with 0 overhead.

4. PARAMETRIC MONITORING

We next discuss an important topic in monitoring, namely, the monitoring of parametric specifications. Parametric specifications, i.e., specifications associated with parameters, are widely used in practice, particularly in object oriented languages, like Java, where we need to describe properties over a group of objects. For example, in the specification in Figure 1, the events are parametrized by the Lock l and Thread t. This is because we do not want events from multiple threads to interfere with each other, and because we are only interested in the pattern with respect to a given lock. If a program locks a lock x and releases lock y, we should not see this as a validation of the property, in fact, assuming y has not been locked previously, this should be a violation of the pattern. Note that not all problem domains need parameters. For example, BusMOP does not support parametric events because the only possible candidate for parametrization is memory address, and specifications in BusMOP concern themselves with specific memory addresses (e.g., control registers on peripherals) or ranges of addresses (e.g., buffers), rather than all memory addresses.

When monitoring a parametric specification, the observed execution trace is parametric, i.e., the events in the trace come with parameter information. For example, a possible parametric trace for the specification in Figure 1 is acq$\langle l_1, t_1\rangle$ acq$\langle l_2, t_1\rangle$ acq$\langle l_1, t_1\rangle$ rel$\langle l_1,t_1\rangle$ rel$\langle l_1, t_1\rangle$ rel$\langle l_2, t_1\rangle$ acq$\langle l_1, t_2\rangle$ rel$\langle l_1, t_2\rangle$[1]. Every event in this trace is associated with a concrete parameter binding, such as $\langle l_1, t_1\rangle$ that indicates that the parameters l and t in Figure 1 are bound to concrete objects l_1 and t_1, respectively. Such a parametric trace represents a set of non-parametric traces each of which corresponds to a particular parameter binding. For example, the above trace contains three non-parametric traces for three parameter bindings as shown below:

$\langle l_1,t_1\rangle$	$\langle l_2,t_1\rangle$	$\langle l_1, t_2\rangle$
acq acq rel rel	acq rel	acq rel

Each of these matches the pattern in Figure 1, thus no violations are produced. It is highly non-trivial to monitor parametric specifications efficiently since there can be a tremendous number of parameter bindings during a single execution. For example, in a few experiments that we carried out, millions of parameter bindings were created [12]. Most other approaches for monitoring parametric specifications handle parameters in a logic-specific way [19, 1, 6], that is, they extended the underlying specification formalisms with parameter and devised algorithms for the extended formalism. Such solution results in very complicated monitor synthesis algorithms and makes it difficult to support new problem domains. In MOP, parameters are handled in a completely logical formalism independent manner and separated from the monitor synthesis process, vastly simplifying the implementation of new logic plugins. Also, surprisingly, this logic independent consideration of parameters turns out to be more efficient than those closely coupled systems (see Section 5) thanks to the clean separation of concerns.

Our solution to parametric specification is based on parametric trace slicing. Parametric trace slicing is the process of taking a parametric trace of events, and producing a set of not-parametric traces, such that the parameter instances of all the events grouped into a given resultant non-parametric slice are compatible. More formally, trace slicing is defined as a reduct operation that forgets all the events unrelated to the given parameter instance. Parameter instances are considered *compatible* if, for all parameters that are instantiated, the parameters agree. For example, $\langle l_1, t_1\rangle$ and $\langle l_1\rangle$ are compatible and $\langle l_1, t_1\rangle$ and $\langle l_1, t_2\rangle$ are incompatible.

Since we can cleanly slice a parametric trace into *sets* of

[1]This example is made easier by the fact that the only two events have the same number of parameters, more complicated examples can be seen in [13].

```
Algorithm $\mathbb{A}\langle X\rangle$
Input: parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$
Output: map $\mathbb{T} \in [[X \rightharpoonup V] \rightharpoonup \mathcal{E}^*]$ and
        set $\Theta \subseteq [X \rightharpoonup V]$
1 $\mathbb{T} \leftarrow \perp$; $\mathbb{T}(\perp) \leftarrow \epsilon$; $\Theta \leftarrow \{\perp\}$
2 foreach $e\langle\theta\rangle$ in order in $\tau$ do
3 : foreach $\theta' \in \{\theta\} \sqcup \Theta$ do
4 : : $\mathbb{T}(\theta') \leftarrow \mathbb{T}(\max(\theta']_\Theta)\, e$
5 : endfor
6 : $\Theta \leftarrow \{\perp, \theta\} \sqcup \Theta$
7 endfor
```

Figure 4: Parametric slicing algorithm $\mathbb{A}\langle X\rangle$.

non-parametric traces, we can use a set of monitor instances, each of which handle a non-parametric trace specialized to a given parameter instance, to verify the input parametric trace. The monitor is generated from the non-parametric formula in the specification, such as the CFG pattern in Figure 1. This way, the underlying logic plugin does not need to be aware of the parameters and can make use of any existing optimal algorithms for non-parametric formalisms.

4.1 Online Parametric Trace Slicing

In this section we briefly discuss the base algorithm for online parametric trace slicing first introduced in [13], which can be seen in Figure 4. By online, we mean that the trace arrives incrementally, one event at a time. Several optimizations have been applied to this base algorithm and those interested in the correctness and the optimizations of this algorithm are encouraged to read [13, 10].

The input to the algorithm is a trace $\tau \in \mathcal{E}\langle X\rangle^*$. Here, $\mathcal{E}\langle X\rangle$ represents the set of parametric events, and $\mathcal{E}\langle X\rangle^*$ is thus the set of parametric traces. $[X \rightharpoonup V]$ is the set of parameter instances, which are, mathematically, the set of partial functions from the set of parameters X to program values V (e.g., considering our running example, we can see $\mathsf{acq}\langle l_1, t_1\rangle$ is simply shorthand for $\mathsf{acq}\langle l \rightarrow l_1, t \rightarrow t_1\rangle$). For outputs, Θ is a set of parameter bindings and $\mathbb{T}$ is a lookup map that takes, as an argument, a parameter binding, and returns a non-parametric trace from $\mathcal{E}^*$. This is a rather formal way to look at online monitoring; intuitively, a set of monitors, each specialized for a given parameter instance, can use this map to filter whether or not a given event belongs in its non-parametric trace slice.

The algorithm begins with $\mathbb{T}$ as having only one binding, from the empty parameter set ($\perp$) to the empty trace (ϵ). Θ begins knowing only about the empty parameter set. Intuitively then, the algorithm begins with one monitor that is keeping track of events for the empty parameter set. Any non-parametric events which arrive, will be added to the map index for the empty parameter binding. The foreach loop on line 2 in Figure 4 loops over each event in trace τ, note, however, that each event actually arrives incrementally, as the monitored program is running. The foreach loop on line 3 loops over every parameter binding $\theta' \in \Theta$, which is compatible with θ. As explained earlier, compatible means they agree on the values of all parameters for which they have a binding. For example, a non-parametric event will be compatible with *all* parameter bindings in Θ at the time it arrives, because $\perp$, its parameter instance, is compatible with all other parameter bindings. On line 4, the trace for θ', that is $\mathbb{T}(\theta')$, is updated using the trace from $\mathbb{T}(\max(\theta']_\Theta)$ appended with the current event. What the operator max does, is find the parameter instance that is maximally compatible with θ', that is the parameter binding which is both compatible, and instantiates the most variables. This ensures that if θ' is a new, never-before-seen parameter binding, it will be assigned the trace from the most compatible parameter instance, with the current event added to the end.

5. PERFORMANCE EVALUATION

	SafeMapIterator		SafeSyncCollection		SafeSyncMap		SafeIterator		
	TM	MOP	TM	MOP	TM	MOP	TM	MOP	PQL
antlr	-2	2	-2	1	-3	1	0	0	82
bloat	>10000	935	1448	712	2267	660	11258	749	8694
chart	-1	0	0	1	1	0	11	3	50
eclipse	8	1	0	0	0	1	2	1	1
fop	11	-3	-4	0	16	-3	5	1	24
hsqldb	29	0	24	0	22	0	17	0	78
jython	57	7	6	-4	8	-5	16	0	12
luindex	7	5	0	1	3	4	9	5	181
lusearch	9	-1	9	1	8	-1	34	2	132
pmd	>10000	196	33	15	50	12	196	14	1334
xalan	10	4	7	1	6	0	10	8	53

Table 1: Average Percent Runtime Overhead for Tracematches (TM), JavaMOP (MOP), and PQL (for SafeIterator only) (convergence within 3%).

We evaluated our implementation on the DaCapo benchmark suite[5]. Tracematches[1] was also evaluated for comparison. For one of the properties, SafeIterator, we also include numbers for PQL[19] that were originally published in [20]. Also note that the experiments discussed below are only part of our evaluation due to the limited space of this paper; more results can be found in [12, 20]. Briefly, JavaMOP performs better than any other existing monitoring techniques in most experiments. We do not have a performance evaluation of BusMOP, because the overhead of BusMOP is effectively 0.

5.1 Experimental Settings

Our experiments were performed on a machine with 1.5GB RAM and a Pentium 4 2.66GHz processor. The machine's operating system is UBuntu Linux 7.10, and we used version 2006-10 of the DaCapo benchmark suite. It contains eleven open source programs [5]: antlr, bloat, chart, eclipse, fop, hsqldb, jython, luindex, lusearch, pmd, and xalan. The default input for DaCapo was used, and we use the -converge option to ensure the validity of our test by running each test multiple times, until the runtime converges. After this convergence, the runtime is stabilized within 3%, thus numbers in Table 1 should be interpreted as "±3%". Furthermore, additional code introduced by the AspectJ weaving process changes the program structure in DaCapo, and sometimes this causes the benchmark to run a little bit faster due to better concurrency interleaving and/or cache layout.

5.2 Properties

We used the following properties, borrowed from [7, 8].

- SafeMapIterator: Do not update a Map when using the Iterator interface to iterate its values or its keys;
- SafeSyncCollection: If a Collection is synchronized, then its iterator also should be accessed in a synchronized manner;
- SafeSyncMap: If a Collection is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner;

- SafeIterator: Do not update a Collection when using the Iterator interface to iterate its elements;

SafeMapIterator, SafeSyncCollection were chosen specifically because the events beginning traces do not contain instantiations of all the parameters. This is important because earlier versions of JavaMOP could not monitor properties such as these. SafeIterator was chosen primarily because is has generated some of the largest overheads in previous works[12, 20]. We do not use the specification given in Figure 1 because it is not expressible in Tracematches, which only supports regular language based specifications.

5.3 Results and Discussions

Table 1 summaries the results of our experiments. It shows the percent overheads of JavaMOP and Tracematches. All the properties were heavily monitored in the experiments. As shown in [12], millions of parameter instances were observed for some properties under monitoring, e.g., SafeIterator, putting a critical test on the generated monitoring code. Both systems generated unnoticeable runtime overhead in most experiments, showing their efficiency. For JavaMOP, only 7 out of 66 cases caused more than 10% runtime overhead. The numbers for Tracematches are 9 out of 66.

6. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 345–364. ACM, 2005.

[2] AspectJ. http://eclipse.org/aspectj/.

[3] MOP website. `http://fsl.cs.uiuc.edu/MOP`.

[4] H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma, editors. *Runtime Verification (RV'05)*, volume 144 of *ENTCS*. Elsevier, 2005.

[5] 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)*, pages 169–190. ACM, 2006.

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

[7] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Aspect-Oriented Software Development (AOSD'09)*, 2009. to appear.

[8] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 525–549, 2007.

[9] F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 357–372, 2004.

[10] F. Chen, D. Jin, P. Meredith, and G. Roşu. Efficient Formalism-Independent Monitoring of Parametric Properties (Extended Version). Technical Report UIUCDCS-R-2008-2977, University of Illinois at Urbana-Champaign, 2008.

[11] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 108–127, 2003.

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

[13] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, 2009. to appear.

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

[15] K. Havelund, M. Nunez, G. Roşu, and B. Wolff, editors. *Formal Approaches to Testing and Runtime Verification (FATES/RV'06)*, volume 4264 of *LNCS*. Springer, 2006.

[16] K. Havelund and G. Roşu, editors. *Runtime Verification (RV'02)*, volume 70 of *ENTCS*. Elsevier, 2002.

[17] K. Havelund and G. Roşu, editors. *Runtime Verification (RV'04)*, volume 113 of *ENTCS*. Elsevier, 2004.

[18] 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)*, volume 2072 of *LNCS*, pages 327–353, 2001.

[19] 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)*, pages 365–383. ACM, 2005.

[20] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. In *Automated Software Engineering (ASE '08)*, pages 148–157. IEEE/ACM, 2008.

[21] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Roşu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time System Symposium (RTSS'08)*, pages 481–491. IEEE, 2008.

[22] G. Roşu and K. Havelund. Monitoring java programs with Java PathExplorer. In *In Proceedings of Runtime Verification (RV'01)*, pages 97–114. Elsevier, 2001.

[23] O. Sokolsky and M. Viswanathan, editors. *Runtime Verification (RV'03)*, volume 89 of *ENTCS*. Elsevier, 2003.