

# MOP: An Efficient and Generic Runtime Verification Framework \*

Feng Chen Grigore Roşu

University of Illinois at Urbana-Champaign  
{fengchen,grosu}@cs.uiuc.edu

## Abstract

Monitoring-Oriented Programming (MOP<sup>1</sup>) [21, 18, 22, 19] is a formal framework for software development and analysis, in which the developer specifies desired properties using definable specification formalisms, along with code to execute when properties are violated or validated. The MOP framework automatically generates monitors from the specified properties and then integrates them together with the user-defined code into the original system.

The previous design of MOP only allowed specifications without parameters, so it could not be used to state and monitor safety properties referring to two or more related objects. In this paper we propose a *parametric specification-formalism-independent extension of MOP*, together with an implementation of JavaMOP that supports parameters. In our current implementation, parametric specifications are translated into AspectJ code and then weaved into the application using off-the-shelf AspectJ compilers; hence, MOP specifications can be seen as formal or logical aspects.

Our JavaMOP implementation was extensively evaluated on two benchmarks, Dacapo [14] and Tracematches [8], showing that runtime verification in general and MOP in particular are feasible. In some of the examples, millions of monitor instances are generated, each observing a set of related objects. To keep the runtime overhead of monitoring and event observation low, we devised and implemented a *decentralized indexing* optimization. Less than 8% of the experiments showed more than 10% runtime overhead; in most cases our tool generates monitoring code as efficient as the hand-optimized code. Despite its genericity, JavaMOP is empirically shown to be more efficient than runtime verification systems specialized and optimized for particular speci-

fication formalisms. Many property violations were detected during our experiments; some of them are benign, others indicate defects in programs. Many of these are subtle and hard to find by ordinary testing.

**Categories and Subject Descriptors** D.2.4 [SOFTWARE ENGINEERING]: Software/Program Verification

**General Terms** Languages, Verification

**Keywords** runtime verification, aspect-oriented programming, monitoring-oriented programming

## 1. Introduction

Runtime verification [29, 41, 11] aims at combining testing with formal methods in a mutually beneficial way. The idea underlying runtime verification is that system requirements specifications, typically formal and referring to temporal behaviors and histories of events or actions, are rigorously checked at runtime against *the current* execution of the program, rather than statically against all hypothetical executions. If used for bug detection, runtime verification gives a rigorous means to state and test complex temporal requirements, and is particularly appealing when combined with test case generation [5] or with steering of programs [35]. A large number of runtime verification techniques, algorithms, formalisms, and tools such as Tracematches [2], PQL [38], PTQL [27], MOP [19], Hawk/Eraser [23], MAC [35], PaX [28], etc., have been and are still being developed, showing that runtime verification is increasingly adopted not only by formal methods communities, but also by programming language designers and software engineers.

We present a parametric extension together with a mature, optimized and thoroughly evaluated implementation of monitoring-oriented programming (MOP). MOP was first proposed in 2003 [21] as a software development and analysis framework based on runtime verification intuitions and techniques. It was further described and extended in [18, 22, 19], but, up to now, it was not able to handle parameters in specifications, and was not shown, through large-scale performance tests measuring run-time overhead, to be feasible in practice. An implementation of JavaMOP was carried out to support these, together with decentralized monitor indexing algorithms for reducing the runtime overhead.

As shown in this paper, MOP is expressive, generic, and efficient. MOP *logic-plugins* encapsulate monitor synthesis algorithms for logics of interest; these allow users comfort-

\* This material is based upon work supported by the National Science Foundation under Grant No. 0448501 and Grant No. 0509321. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

<sup>1</sup> Not to be confused with “meta-object protocol” [34].

```

class Resource {
/*@
  scope = class
  logic = PTLTL
  {
    event authenticate: end(exec(* authenticate()));
    event access: begin(exec(* access()));
    formula: access -> <*> authenticate;
  }
  violation handler { @this.authenticate(); }
/*@
void authenticate() {...}
void access() {...}
...
}

```

**Figure 1.** MOP specification for resource safety

able with formal notation to declare properties using high-level or application-specific requirements specification formalisms. Specifications using any of the logic-plugins are allowed to have parameters; this way, multiple monitor instances for the same property can coexist, one per collection of objects of interest. MOP also allows its users to implement monitors manually, using the full strength of the target language. In other words, MOP supports and encourages the use of formal specifications, but it does not require it.

### 1.1 Examples

Let us consider a simple and common safety property for a shared resource, namely that any access to the resource should be authenticated. For simplicity, suppose that all the operations on the shared resource are implemented in the class `Resource`, including methods `access()` and `authenticate()`. Then the safety property can be specified as a trivial “always past” linear temporal logic (LTL) formula over method invocations, namely

```
access -> <*> authenticate,
```

stating that “if `access` then `authenticate` held in the past” (“<\*>” reads “eventually in the past”); the “always” part is implicit, since MOP properties are continuously monitored.

Using MOP like in Figure 1, one can *enforce* this policy to hold in any system that manages the resource via the `Resource` class; by “enforce” we mean that MOP ensures that the system will satisfy the property even though it was not originally programmed (intentionally or not) to satisfy it.

The first line of the MOP specification in Figure 1 states that this property is a class invariant, i.e., it should hold in the scope of this class (specification attributes are discussed in Section 4.1). The second line chooses a desired formalism to express the corresponding formal requirement, in this case past-time LTL (PTLTL); MOP allows users to “plug-and-play” new specification formalisms, provided that they respect the standardized interface of logic-plugins (these are discussed in Section 3.2). The content enclosed by the curly brackets is specific to the chosen formalism. For PTLTL, the user needs to first build an abstraction that maps runtime events into logical elements, e.g., the invocation of `authenticate()` being mapped to an event `authenticate`. Using the elements produced by the ab-

```

/**MonitorAspect**/
public aspect MonitorAspect {
  /** Generated by JavaMOP for javamop.monitor PTLTL_0 */
  public boolean[] Resource.PTLTL_0_pre = new boolean[1];
  public boolean[] Resource.PTLTL_0_now = new boolean[1];
  pointcut PTLTL_0_Init(Resource thisObject):
    execution(Resource.new(..) && target(thisObject));
  after(Resource thisObject): PTLTL_0_Init(thisObject) {
    boolean authenticate = false;
    boolean access = false;
    thisObject.PTLTL_0_now[0] = authenticate;
  }
  pointcut PTLTL_0_authenticate0(Resource thisObject):
    target(thisObject) && execution(* Resource.authenticate());
  after (Resource thisObject) returning:
    PTLTL_0_authenticate0(thisObject) {
    boolean authenticate = false;
    boolean access = false;
    authenticate = true;
    thisObject.PTLTL_0_pre[0] = thisObject.PTLTL_0_now[0];
    thisObject.PTLTL_0_now[0] = authenticate ||
      thisObject.PTLTL_0_pre[0];
    if (access && ! thisObject.PTLTL_0_now[0]){
      thisObject.authenticate(); }
  }
  pointcut PTLTL_0_access0(Resource thisObject):
    target(thisObject) && execution(* Resource.access());
  before (Resource thisObject):
    PTLTL_0_access0(thisObject) {
    boolean authenticate = false;
    boolean access = false;
    access = true;
    thisObject.PTLTL_0_pre[0] = thisObject.PTLTL_0_now[0];
    thisObject.PTLTL_0_now[0] = authenticate ||
      thisObject.PTLTL_0_pre[0];
    if (access && ! thisObject.PTLTL_0_now[0]){
      thisObject.authenticate(); }
  }
}
/* Generated code ends **/
}

```

**Figure 2.** Generated monitor for the property in Figure 1

straction, a PTLTL formula is given to describe the desired property. The last part of the MOP specification contains the code that will be triggered when the specification is violated and/or validated. It may be as simple as reporting errors, or as sophisticated as taking recovery actions to correct the execution to avoid crashes of the system. In this example, when the safety property is violated, i.e., when some access is not authenticated, we enforce the authentication simply by making a call to `authenticate()`. The MOP tool is able to analyze this specification, generate monitoring code for the formula, and insert the monitor with the recovery handler into appropriate points of the program.

There are two important observations regarding the example above, each reflecting a crucial aspect of MOP:

1. By synthesizing monitoring code from specifications and automatically integrating it together with the recovery code at relevant points in the program, the developer can and should have quite a *high confidence that the resource is used correctly* throughout the system. In fact, if we trust that the MOP tool generates and integrates the monitoring code correctly, then we can also trust that the resulting system is correct w.r.t. this safety property, no matter how complicated the system is.

```

/*@
scope = class
logic = ERE
{
  [static int counter = 0; int writes = 0;]
  event open : end(call(* open(..)) {writes = 0;});
  event write : end(call(* write(..)) {writes ++;});
  event close : end(call(* close(..)));
  formula : (open write+ close)*
}
violation handler{ @RESET; }
validation handler{ synchronized(getClass()){
    File.log(++counter) + ":" + writes; } }
@*/

```

**Figure 3.** MOP specification for file profiling

- Suppose that authentication-before-access was not a requirement of the system originally, but that it became a desired feature later in the development process (e.g., because of a larger number of clients). Suppose also that, as a consequence, one wants to add authentication to an initial implementation of the system that provided no support and no checking for authentication. Using MOP, all one needs to do is to add an (unavoidable) `authenticate()` method, together with the MOP specification in Figure 1. This way, the MOP specification together with its violation handler *added non-trivial functionality* to the system, in a fast, elegant and correct way.

Monitors corresponding to specifications may need to observe the execution of the program at many different points, which can be scattered all over the system. In this sense, every monitor can be regarded as a crosscutting feature, like in aspect-oriented programming (AOP) [33]. MOP can be regarded as a specialized instance of AOP, in which *aspects are (formal) specifications*. Existing AOP tools provide crucial support for MOP to integrate generated monitoring code as well as recovery code into the system. From this point of view, MOP acts as a *supplier of aspects*: it converts specifications into concrete aspects that can be handled by existing AOP tools. For instance, our MOP front-end for Java discussed in Section 3.3, JavaMOP, translates the specification in Figure 1 into the AspectJ code in Figure 2 (that code is further compiled using off-the-shelf AspectJ compilers).

Comparing Figure 1 with Figure 2, one can see that MOP provides an abstract programming environment, hiding underlying implementation details. Low-level error-prone tasks, such as transforming formulae into monitors or choosing appropriate join points to integrate monitors and recovery code, are all automatically handled by the MOP framework; this way, the user is freed to focus on the interesting and important aspects of the system.

The example above shows an “event-harmless” MOP specification, i.e., one that executes no auxiliary code when events are observed (except running the generated monitor), with a violation handler encapsulating all desired recovery code. Figure 3 depicts a more intrusive MOP specification with both violation and validation handlers, also showing how MOP can be used for profiling. The logic-plugin used

this time is for extended regular expressions (ERE), that is, regular expressions extended with complement (no complement is needed here, but the ERE plugin generates optimal monitors also for ordinary regular expressions).

Two auxiliary variables are defined as part of the MOP specification, a static counter and a per-object `writes`. The desired pattern to profile is `(open write+ close)*`, that is, how many times we see an open followed by one or more `writes` followed by a `close`. Each open event resets the `writes`, which is then incremented at each write event. The validation handler, which in the case of EREs is triggered whenever the automaton monitor reaches its final state, logs the `writes` and increments the static counter; note that this handler needs to synchronize on the class to avoid potential races. The violation handler, which for EREs is triggered whenever the automaton monitor cannot advance to a next state (in our case, that most likely happens when a file is open then closed without any writes), resets the monitor to its initial state using the MOP reserved command `@RESET`.

Both MOP specifications above are class scoped: they refer to behaviors of individual objects. There are, however, many safety properties of interest that refer to collections of two or more objects. Some of these are considered so important that language designers feel it appropriate to include corresponding runtime safety checks as built-in part of programming languages. For example, Java 5 raises a `ConcurrentModificationException` when running

```

Vector v = new Vector();
v.add(new Integer(10));
Iterator i = v.iterator();
v.add(new Integer(20));
System.out.println(i.next());

```

That is because the Iterators returned by Vector’s iterator methods are assumed *fail-fast* in Java: the Vector is not allowed to be modified while the Iterator accesses its elements. However, the Enumerations returned by Vector’s `elements` method are *not* assumed fail-fast in Java 5, and, obviously, neither are any other user-defined iterator-like objects. One can easily imagine many other similar tight relationships among two or more objects, either language-specific as above or application-specific. For example, a security policy in an application can be: for any password  $p$ , string  $s$  and file  $f$ , it is never the case that  $s$  is the decrypted version of  $p$  (as returned by some `decrypt` method) and  $s$  is written on  $f$ .

To support such important properties referring to groups of objects, MOP now provides a generic mechanism allowing for *universal parameters* to requirements specified using any of the logic-plugins. Figure 4 shows an MOP specification making enumeration objects corresponding to vectors also fail-fast. Note that this time the MOP specification is globally scoped, because it refers to more than one object. The property to check, which is also given an optional name here, `SafeEnum`, has two parameters: a Vector  $v$  and an Enumeration+  $e$ ; the “+” says that the property (and its monitors) is inherited by all subclasses of Enumeration.

```

/*@
scope = global
logic = ERE
SafeEnum (Vector v, Enumeration+ e) {
  [String location = ""];
  event create<v,e>: end(call(Enumeration+.new(v,...)) with (e));
  event updatesource<v>: end(call(* v.add*(..)) \ /
                             end(call(* v.remove*(..)) \ / ...
                             {location = @LOC;})
event next<e>: begin(call(* e.nextElement()));
formula : create next* updatesource+ next
}
validation handler { System.out.println("Vector updated at "
                                         + @MONITOR.location); }
/*@

```

**Figure 4.** MOP specification for safe enumeration

The event `create<v,e>` is parametric in both `v` and `e`, and is generated whenever enumeration `e` is created for vector `v`. The event `updatesource<v>` is generated when methods that modify the vector are called; to save space, we did not list all of them in Figure 4. The location (file and line number) of the update is also stored in the variable `location`, using the MOP reserved variable `@LOC`. An ERE formula expresses the faulty pattern: an `updatesource` event is seen after `create` and before a `next`; events in this pattern are assumed parameterized as above.

The validation handler here simply reports the location where the vector was wrongly updated (this info is useful for debugging); the MOP reserved keyword `@MONITOR` gives a reference to the corresponding monitor instance, which has the declared monitor variables (only `location` here) as fields. MOP will create as many monitors for this property as corresponding instances of `v` and `e` are generated during the execution of the application, and will dispatch the events correspondingly; for example, if several enumerations are created for the same vector `v`, then an `updatesource<v>` event is sent to each instance monitor corresponding to each enumeration of `v`. JavaMOP generates about 200 lines of AspectJ code from the specification in Figure 4.

## 1.2 Contributions

As already mentioned, the basic idea of MOP and a first JavaMOP prototype have already been discussed in several places [21, 18, 19, 22]. However, the previous design and implementation of MOP lacked parameters and thus had limited practical use. In particular, the safe enumerator example in Figure 4, the examples supported by other runtime verification systems such as Tracematches [2], PQL [38] and PTQL [27], as well as most of the examples in Sections 6 were previously not possible to define in MOP using formal specifications. Our contributions in this paper are:

### (1) *Universal parameters, decentralized indexing*

We present a generic technique to add *universal parameters* to trace-based logics, together with an optimized implementation based on *decentralized indexing*. Logical formalisms used in runtime verification and monitoring have traces as models; in particular, all our MOP logic-plugins are trace-based. However, existing runtime verification sys-

tems supporting parametric properties use centralized monitors and indexing, that is, all monitors are stored in a common pool and parametric events are resolved and dispatched at this centralized level, incurring unavoidable runtime overhead when the pool contains many objects. Our decentralized indexing technique is logic-independent, so it can be adopted by any runtime verification system. As empirically shown in Section 6, despite its genericity with regard to logical formalisms, MOP with decentralized indexing is more efficient than the current state-of-the-art runtime verification systems specialized and optimized for particular logics.

### (2) *New MOP language, raw MOP specifications*

We defined a new MOP specification language, which allows not only specifications of parametric properties using MOP logic-plugins, but also definitions of *raw MOP specifications*. Raw MOP specifications require no logic-plugin and consequently no logical formula, so they need to be explicitly implemented by users in the target language (e.g., Java); in this case, the MOP framework provides a useful abstraction allowing users to define monitor variables and/or event actions, to generate and handle violation or validation signals, to use MOP reserved keywords and commands, etc.; the developer of raw MOP specifications can fully utilize the strength of the target language. Raw MOP specifications may be preferred by users who are not comfortable with formal notation. We use them to write hand-optimized monitors for the experiments in Section 6. Due to its new enriched specification language, MOP now captures many other runtime verification frameworks as specialized instances (these are discussed in Section 2); this genericity comes at no performance penalty (on the contrary). Therefore, MOP is now a viable generic platform for runtime verification projects, allowing experimentation with new logics for monitoring, safety policies, monitor synthesis algorithms, and so on.

### (3) *Evaluation and Experiments*

A large number of experiments have been carried out to evaluate the feasibility and effectiveness of MOP: we used JavaMOP to check more than 100 property-program pairs. The results are encouraging: in most cases, the runtime overhead was negligible; only 8% of experiments showed noticeable slow-down of 10% or more. In some purposely designed extreme cases, the runtime overhead was still less than 200%, but we were able to write raw MOP specifications for the same properties, reducing the overhead below 30%. We did not focus on error detection, in the sense that no test generation techniques were used. However, many violations of specified properties were revealed; some of these are benign (but still interesting to be aware of), others indicate possible defects of programs: an inappropriate usage of `StringWriter` leads to a write-after-close violation in Xalan [42]; possible resource leaks in Eclipse [25] GUI packages; a violation of `SafeEnum` caused by concurrency in `jHotDraw` [32]; etc. (see Section 6.2). These subtle problems are difficult to detect using ordinary testing, but Java-

MOP provided good support to locate their root causes. Our experiments show that runtime verification in general and MOP in particular are feasible and effective in practice. Both JavaMOP and the experiments are publicly available at [20].

## 2. Related Work

We next discuss relationships between MOP and other related paradigms, including AOP, design by contract, runtime verification, and other trace monitoring approaches. Broadly speaking, all the monitoring approaches discussed below are runtime verification approaches; however, in this section only, we group into the runtime verification category only those approaches that explicitly call themselves runtime verification approaches. Interestingly, even though most of the systems mentioned below target the same programming languages, no two of them share the same logical formalism for expressing properties. This observation strengthens our belief that probably there is *no silver bullet logic* (or *super logic*) for all purposes. A major objective in the design of MOP was to avoid hardwiring particular logical formalisms into the system. In fact, as shown in Sections 3 and 4, MOP specifications are generic in four orthogonal directions:

MOP[logic, scope, running mode, handlers].

The logic answers *how to specify* the property. The scope answers *where to check* the property; it can be class invariant, global, interface, etc. The running mode answers *where the monitor is*; it can be inline, online, offline. The handlers answer *what to do if*; there can be violation and validation handlers. For example, a particular instance can be

MOP[ERE, global, inline, validation],

where the property is expressed using the ERE logic-plugin for extended regular expressions (EREs), the corresponding monitor is global and inline, and validation of the formula (pattern matching in this case) is of interest.

Most approaches below can be seen as such specialized instances of MOP for particular logics, scopes, running modes and handlers. There are, of course, details that make each of these approaches interesting in its own way.

### 2.1 Aspect Oriented Programming (AOP) Languages

Since its proposal in [33], AOP has been increasingly adopted and many tools have been developed to support AOP in different programming languages, e.g., AspectJ and JBoss [31] for Java and AspectC++ [4] for C++. Built on these general AOP languages, numerous extensions have been proposed to provide domain-specific features for AOP. Among these extensions, Tracematches [2] and J-LO [15] support history(trace)-based aspects for Java.

Tracematches enables the programmer to trigger the execution of certain code by specifying a regular pattern of events in a computation trace, where the events are defined over entry/exit of AspectJ pointcuts. When the pattern is matched during the execution, the associated code will be executed. In this sense, Tracematches supports trace-based

pointcuts for AspectJ. J-LO is a tool for runtime-checking temporal assertions. These temporal assertions are specified using LTL and the syntax adopted in J-LO is similar to Tracematches' except that the formulae are written in a different logic. J-LO mainly focuses on checking at runtime properties rather than providing programming support. In J-LO, the temporal assertions are inserted into Java files as annotations that are then compiled into runtime checks. Both Tracematches and J-LO support parametric events, i.e., free variables can be used in the event patterns and will be bound to specific values at runtime for matching events. Conceptually, J-LO can be captured by MOP, because LTL is supported by MOP and J-LO's temporal assertions can be easily translated into MOP specifications that contain only action events and validation handlers.

Although MOP supports regular expressions as part of its ERE logic plugin, they hold a different semantics from Tracematches' patterns. More specifically, the MOP regular pattern specifications are checked against the *whole execution trace*, while Tracematches' patterns can be matched with *partial traces*. For example, for a pattern "A\* B", a sequence of events "A B B" will trigger the validation handler of the generated MOP monitor only at the first "B" event and then the violation handler at the second "B". In Tracematches, the pattern will be matched twice, once for each "B" event: the first matches either the whole trace "A B" or the partial trace consisting of just the first "B" with zero occurrences of "A", while the second matches the subsequent partial trace "B" (the second "B" in the trace) with zero occurrences of "A"; thus, the related advice will be executed twice. Both semantics are useful for different purposes. For example, in Tracematches one can count matches of a pattern "open close" without a need to reset the monitor after each match as in MOP. On the other hand, total trace matching is more suitable for runtime verification of formal properties, because it is also the semantics adopted by conventional formal verification techniques and tools.

Note, however, that it is relatively easy to support partial trace matching in a total matching and vice versa. For example, to capture partial matching in a total matching setting such as MOP's, all one needs to do is to maintain a *set* of states, while a new monitor state is produced at each event; the set will contain at most as many states as the property monitor can have. Conversely, to capture total trace matching in a partial matching setting such as Tracematches', all one needs to do is to generate an artificial event only once at the beginning of the trace, say "start", and then automatically change any pattern "P" to "start P".

Event though partial and total trace semantics are ultimately equivalent as explained above, the partial trace one is less conventional and thus can lead to confusing behaviors if one is not careful. For example, let us re-consider the safety property in Figure 1. By exhaustively enumerating all the methods in the Resource class, one may use a pattern,

```

class Resource {
/*@
  scope = class
  logic = ERE
  {
    // not-authenticate includes all methods other than
    // authenticate() and access()
    event not-authenticate: end(exec(* ...));
    event authenticate: end(exec(* authenticate()));
    event access: begin(exec(* access()));
    formula: not-authenticate* access;
  }
  validation handler { @this.authenticate(); }
/*@
void authenticate() {...}
void access() {...}
...
}

```

**Figure 5.** Regular MOP specification for resource safety

`not-authenticate* access`, to detect non-authenticated accesses, as in Figure 5. If the authentication is carried out before any access, this specified pattern will not be matched by the monitor generated by MOP. Otherwise, when the *first* unsafe access occurs, the MOP monitor will trigger the validation handler, enforcing the authentication right before the access and ensuring all the following accesses safe. Without an explicit capturing of total trace matching in a partial trace matching setting as explained above, it is difficult, if not impossible, to achieve the same behavior using Tracematches. For instance, the above pattern will be matched even for a correct execution, e.g., `authenticate access`, because there exists a matching partial trace consisting of only `access`.

In addition to inline and global monitors currently supported by Tracematches and J-LO, MOP also provides class-scoped properties, outline and offline monitor settings, and more. Fixing a logic allows for developing static analysis and logic-specific optimizations. We have not attempted to devise any logic-specific optimizations yet in MOP, because we do not regard MOP's runtime overhead as a bottleneck yet. In Section 6, we show that the MOP instance `MOP[ERE, class/global, inline, validation]` using decentralized indexing adds significantly less runtime overhead than Tracematches with static analysis enabled. It is also worth mentioning that Tracematches and J-LO are implemented using Java bytecode compilation and instrumentation, while MOP acts as an aspect synthesizer, making it easier to port to other target languages provided they have AOP tool support.

## 2.2 Runtime Verification

In runtime verification, monitors are automatically synthesized from formal specifications, and can be deployed *offline* for debugging, or *online* for dynamically checking properties during execution. MaC [35], PathExplorer (PaX) [28], and Eagle [12] are runtime verification frameworks for logic based monitoring, within which specific tools for Java – Java-MaC, Java PathExplorer, and Hawk [23], respectively – are implemented. All these runtime verification systems work in outline monitoring mode and have hardwired spec-

ification languages: MaC uses a specialized language based on interval temporal logic, JPaX supports just LTL, and Eagle adopts a fixed-point logic. Java-MaC and Java PathExplorer integrate monitors via Java bytecode instrumentation, making them difficult to port to other languages. Our approach supports inline, outline and offline monitoring, allows one to define new formalisms to extend the MOP framework, and is adaptable to new programming languages.

Temporal Rover [24] is a commercial runtime verification tool based on future time metric temporal logic. It allows programmers to insert formal specifications in programs via annotations, from which monitors are generated. An Automatic Test Generation (ATG) component is also provided to generate test sequences from logic specifications. Temporal Rover and its successor, DB Rover, support both inline and offline monitoring. However, they also have their specification formalisms hardwired and are tightly bound to Java.

Although our current JavaMOP prototype does not support all these techniques yet, it is expected that all the RV systems would fall under the general MOP architecture, provided that appropriate logic-plugins are defined.

## 2.3 Design by Contract

Design by Contract (DBC) [39] is a technique allowing one to add semantic specifications to a program in the form of assertions and invariants, which are then compiled into runtime checks. It was first introduced by Meyer as a built-in feature of the Eiffel language [26]. Some DBC extensions have also been proposed for a number of other languages. Jass [13] and jContractor [1] are two Java-based approaches.

Jass is a precompiler which turns the assertion comments into Java code. Besides the standard DBC features such as pre-/post-conditions and class invariants, it also provides refinement checks. The design of trace assertions in Jass is mainly influenced by CSP [30], and the syntax is more like a programming language. jContractor is implemented as a Java library which allows programmers to associate contracts with any Java classes or interfaces. Contract methods can be included directly within the Java class or written as a separate contract class. Before loading each class, jContractor detects the presence of contract code patterns in the Java class bytecode and performs on-the-fly bytecode instrumentation to enable checking of contracts during the program's execution. jContractor also provides a support library for writing expressions using predicate logic quantifiers and operators such as *forall*, *exists*, *suchThat*, and *implies*. Using jContractor, the contracts can be directly inserted into the Java bytecode even without the source code.

Java modeling language (JML)[36] is a behavioral interface specification language for Java. It provides a more comprehensive modeling language than DBC extensions. Not all features of JML can be checked at runtime; its runtime checker supports a DBC-like subset of JML, a large part of which is also supported by JavaMOP. Spec# [10] is a DBC-like extension of the object-oriented language C#. It extends

the type system to include non-null types and checked exceptions and also provides method contracts in the form of pre- and post-conditions as well as object invariants. Using the Spec# compiler, one can statically enforce non-null types, emit run-time checks for method contracts and invariants, and record the contracts as metadata for consumption by downstream tools.

We believe that the logics of assertions/invariants used in DBC approaches fall under the uniform format of our logic engines, so that an MOP environment following our principles would naturally support monitoring DBC specifications as a special methodological case. In addition, our MOP design also supports outline monitoring, which we find important in assuring software reliability but which is not provided by any of the current DBC approaches that we are aware of.

## 2.4 Other Related Approaches

Acceptability-oriented computing [40] aims at enhancing flawed computer systems to respect basic acceptability properties. For example, by augmenting the compiled code with bounds checks to detect and discard out-of-bound memory accesses, the system may execute successfully through attacks that trigger otherwise fatal memory errors. Acceptability-oriented computing is mainly a philosophy and methodology for software development; one has to devise specific solutions to deal with different kinds of failures. We do believe though that MOP can serve as a platform to experiment with and support acceptability-oriented computing, provided that appropriate specification formalisms express the “acceptability policy” and appropriate recovery ensures that it is never violated.

Program Query Language (PQL) allows programmers to express design rules that deal with sequences of events associated with a set of related objects [38]. Both static and dynamic tools have been implemented to find solutions to PQL queries. The static analysis conservatively looks for potential matches for queries and is useful to reduce the number of dynamic checks. The dynamic analyzer checks the runtime behavior and can perform user-defined actions when matches are found, similar to MOP handlers.

PQL has a “hardwired” specification language based on context-free grammars (CFG) and supports only inline monitoring. CFGs can potentially express more complex languages than regular expressions, so in principle PQL can express more complex safety policies than Tracematches. There is an unavoidable trade-off between the generality of a logic and the efficiency of its monitors; experiments performed by Tracematches colleagues [6] and confirmed by us (see Section 6) show that PQL adds, on average, more than twice as much runtime overhead as Tracematches. We intend to soon take a standard CFG-to-pushdown-automata algorithm and to implement it as an MOP logic-plugin; then MOP will also support (the rare) CFG specifications that cannot be expressed using parametric extended regular expressions or temporal logics, and

MOP[CFG,global,inline,validation] will provide an alternative and more general implementation of PQL.

Program Trace Query Language (PTQL) [27] is a language based on SQL-like relational queries over program traces. The current PTQL compiler, Partique, instruments Java programs to execute the relational queries on the fly. PTQL events are timestamped and the timestamps can be explicitly used in queries. PTQL queries can be arbitrary complex and, as shown in [27], PTQL’s runtime overhead seems acceptable in many cases but we were unable to obtain a working package of PTQL and compare it in our experiments because of license issues. PTQL properties are globally scoped and their running mode is inline. PTQL provides no support for recovery, its main use being to detect errors. It would be interesting to investigate the possibility of developing an SQL logic-plugin for MOP and then to compare the corresponding MOP instance to Partique.

## 3. Overview of MOP and JavaMOP

We here briefly introduce MOP and JavaMOP. Interested readers are referred to [19, 18] for more details, and also to [20] for tool downloads and the latest development news.

### 3.1 MOP Monitoring Model

Many properties can be monitored at the same time in MOP. The execution trace against which the various properties are checked is extracted from the running program as a sequence of events taking state snapshots. Events produce sufficient information about the concrete program state in order for the monitors to correctly check their properties. A monitor is typically interested in a subset of events. Figure 6 illustrates the monitoring model adopted by MOP.

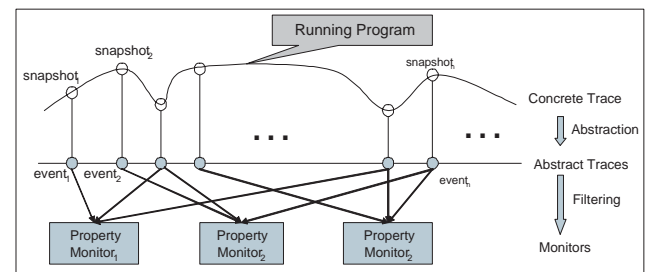


Figure 6. MOP Monitoring Model

In MOP, the runtime monitoring process of each property consists of two orthogonal mechanisms: *observation* and *verification*. The observation mechanism extracts property-relevant and filtered system states at designated points, e.g., when property-specific events happen. The verification mechanism checks the obtained abstract trace against the (monitor corresponding to the) property and triggers desired actions in case of violations or validations. For instance, for the simple global property “always ( $x > 0$ )”, the events to observe are the updates of the variable  $x$  and the relevant state information (or snapshot) to extract is the value



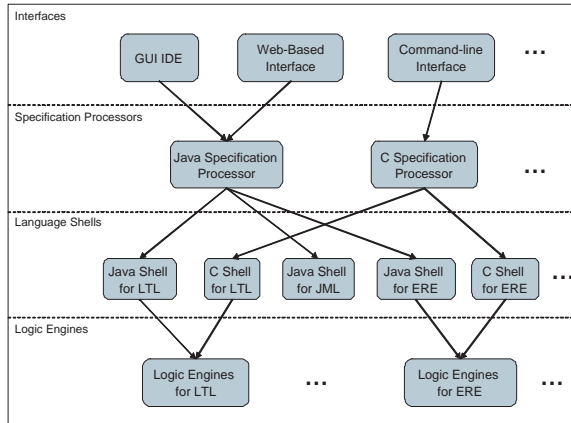


Figure 7. MOP architecture

of  $x$ . This observation process yields a sequence of values of  $x$ : the relevant abstract trace. The corresponding monitor checks whether the value of  $x$  is larger than zero. Observation and verification are therefore independent, in the sense that the algorithm used within the monitor does not affect how the execution is observed, and vice versa.

MOP is a highly configurable and extensible runtime verification framework. Depending upon configuration, the monitors can be separate programs reading events from a log file, from a socket or from a buffer, or can be inlined within the program at the event observation points. The various MOP configuration attributes are discussed in Section 4.

### 3.2 MOP: An Extensible Monitoring Framework

MOP separates monitor generation and monitor integration by adopting the layered architecture in Figure 7. This architecture is especially designed to facilitate extending the MOP framework with new formalisms or new programming languages. By standardizing the protocols between layers, new modules can be added easily and independently. Modules on lower layers can be reused by upper-level modules.

The topmost layer, called the *interface layer*, provides user friendly programming environments. For example, the reader is encouraged to try the web-based interface for JavaMOP at [20] (no download needed, examples provided). The second layer contains *specification processors*, which handle monitor integration. Each specification processor is specific to a target programming language and consists of a program scanner and a program transformer. The scanner extracts MOP specifications from the program and dispatches them to appropriate modules on the lower layer to process. The transformer collects the monitoring code generated by the lower layer and integrates it into the original program. AOP plays a critical role here: the program transformer synthesizes AOP code and invokes AOP compilers to merge the monitors within the program. In particular, as discussed in Section 3.3, JavaMOP transforms generated monitoring code into AspectJ code.

The two lower layers contain the *logic-plugins*, which allow the user to add, remove, or modify specification formalisms. Logic-plugins are usually composed of two modules: a *language shell* on the third layer and a *logic engine* on the bottom layer. The former generates programming language and specification formalism specific monitoring code in a standardized format, which can be understood by the specification processor on the upper layer. The logic engine, acting as the core of monitor generation, synthesizes monitors from specifications in a programming language independent way, e.g., as state machines. This way, logic engines can be reused across different programming languages.

### 3.3 JavaMOP

JavaMOP is an MOP development tool for Java. It provides several interfaces, including a web-based interface, a command-line interface and an Eclipse-based GUI, providing the developer with different means to manage and process MOP specifications. To flexibly support these various interfaces, as well as for portability reasons, we designed JavaMOP following a client-server architecture (see [19]) as an instance of the general MOP architecture in Figure 7. The client part includes the interface modules and the JavaMOP specification processor, while the server contains a message dispatcher and logic-plugins for Java. The specification processor employs AspectJ for monitor integration. In other words, JavaMOP translates outputs of logic-plugins into AspectJ code, which is then merged within the original program by the AspectJ compiler. The message dispatcher is responsible for the communication between the client and the server, dispatching requests to corresponding logic-plugins. The communication can be either local or remote, depending upon the installation of the server.

An immediate advantage of this architecture is that one logic server can provide and cache monitor generation services, which can require intensive computation, to multiple clients. Also, our clients are implemented in Java to run on different platforms, while some of the logic engines are implemented in non-conventional languages and consequently run best on Linux or Unix. Therefore, this architecture increases portability, since the client and the server are allowed to run on different platforms; also the server can cache monitors for common formulae.

Four logic-plugins are currently provided with JavaMOP: Java Modeling Language (JML) [36], Extended Regular Expressions (ERE) and Past-Time and Future-time Linear Temporal Logics (LTL) (see [19] for more details).

## 4. The MOP Specification Language

MOP provides a specification language to define safety properties. The design of this language was driven by two factors: uniformity in supporting different formalisms and languages, and the ability to control monitor behaviors. Language-specific and logic-specific notations are carefully distinguished from other notations in MOP specifications.



The developer is also given the possibility to directly *program* the monitor if she/he wants to fully control the monitoring process (see Section 4.4). The MOP specification language can be regarded as a specialized AOP language, tuned to support specifying monitors either formally using logics or informally by programming.

MOP specifications can be either embedded into the source code as special annotations or stored in separate files. Each format has different advantages. Annotations are more suitable for properties related to specific positions in the source code, e.g., assertions and pre-/post-conditions for methods. On the other hand, separate specification files are conceptually clearer when their corresponding properties refer to multiple places in the program, e.g., global properties. JavaMOP supports both kinds of specifications.

```

<Specification> ::= /*@ <Header> <Body> <Handlers> @*/
<Header> ::= <Attribute>*[scope =<Scope>][logic =<Logic>]
<Attribute> ::= static | outline | offline | centralized
<Scope> ::= global | class | interface | method
<Name> ::= <Identifier>
<Logic> ::= <Identifier>
<Body> ::= [(<Name>)][(<Parameters>)]{<LogicSpecificContent>}
<Parameters> ::= ( <Type> <Identifier> )+
<Handlers> ::= [(<ViolationHandler>)] [(<ValidationHandler>)]
<ViolationHandler> ::= violation handler { <Code> }
<ValidationHandler> ::= validation handler { <Code> }

```

**Figure 8.** Syntax of MOP specifications

Figure 8 shows the syntax of MOP specifications. An MOP specification is composed of three parts: the header, the body and the handlers. We next discuss each of these.

#### 4.1 Header: Controlling Monitor Generation and Integration

The header contains generic information to control monitor generation and integration, consisting of optional attributes, the scope, and the name of the formalism (or logic-plugin) used in the specification.

*Attributes* are used to configure monitors with different installation capabilities. They are orthogonal to the actual monitor generation but determine the final code generated by the MOP tool. Four attributes are available. One is `static`, which has an effect only upon class/interface scoped properties, and says that the specification refers to the class, not to the object. For a static specification, only one monitor instance is generated at runtime and is shared by all the objects of the corresponding class. By default, monitors are non-static, meaning that objects will be monitored individually. In JavaMOP, the variables used to represent the state of the monitor are added to the corresponding class as either static or non-static fields, according to staticness of the monitor; inserting new class fields is done through the inter-type member declaration of AspectJ (e.g., the declaration of `Resource.PTLTL_0_pre` in Figure 2). To avoid name conflicts, these fields are renamed by the specification processor.

Two other attributes, `outline` and `offline`, are used to change the running mode of the monitor. Different properties may require different running modes. For example, a monitor can be executed in the context (thread) of the monitored system, or it can run outside of the monitored system, as a standalone process or thread. We call the former an *inline* monitor, which is also the default mode of the specification, and the latter an *outline* monitor. An inline monitor can interact with the system directly, facilitating information retrieval and error recovery, but some problems, e.g., deadlocks, cannot be detected by inline monitors. Besides, inline monitors may cause significant runtime overhead when running the monitor involves intensive computation. An outline monitor provides a better solution for such cases. In the outline mode, the monitored system sends messages that contain relevant state information to the monitor. However, communication with outline monitors may reduce the performance of the system and, equally importantly, an outline monitor cannot access the internal state of the monitored system, limiting its capability for error recovery.

Another way to check an execution trace, which can sometimes make expensive monitoring feasible by allowing random access to the trace, is offline monitoring: log the trace in a file and make it available to the “monitor”. Since such monitors can run after the monitored system ceases, they are called *offline* monitors. Offline monitors are suitable for properties that can be decided only after the system stops or properties that require a backward traversal of the trace; they may also be useful for debugging and analysis.

These running modes impose different requirements on monitor synthesis. In JavaMOP, inline monitors are merged into the program by encapsulating the monitoring code as an aspect, such as the example in Figure 1 and Figure 2. For outline and offline monitors a standalone monitor class is synthesized, which can run independently as a new thread or process. The MOP tool then generates aspects containing either message passing code (in outline mode) or event logging code (in offline mode).

The last attribute, named `centralized` from “centralized indexing”, can only be combined with global parametric specifications. The default indexing is “decentralized” in MOP, that is the references to monitors are piggybacked into states of some objects in order to reduce the runtime overhead. This technique is discussed in Section 5. As seen also in Section 6, there are situations when a centralized pool of monitors is more suitable; we therefore allow the users the possibility to choose *centralized* indexing.

The *scope* of specifications defines the working scope of monitors, determining the points where properties are checked. Five scopes are supported: `global`, `class`, `interface`, `method`, and a default scope. Properties which are `global` may involve multiple components/objects in the system. The scope `class` says that the property is a class invariant; both `global` and `class` properties are checked when

```

⟨LogicBody⟩ ::= [[⟨VarDeclaration⟩]]⟨Event⟩* [⟨Formula⟩]
⟨Event⟩ ::= ⟨EventHeader⟩ : ⟨EventDecl⟩ [⟨Code⟩];
⟨EventHeader⟩ ::= event⟨Identifier⟩[⟨Parameters⟩>
⟨EventDecl⟩ ::= ⟨EventPoint⟩[with⟨(Type) ⟨Name⟩⟩][&&⟨BExp⟩]
⟨EventPoint⟩ ::= (begin | end)(⟨EventPattern⟩)
⟨EventPattern⟩ ::= (call|exec)(⟨Method⟩) | update(⟨Field⟩)
⟨Formula⟩ ::= formula : ⟨LogicFormula⟩

```

**Figure 9.** MOP syntax for trace-based logic formalisms

the involved fields are updated or the involved methods are called. The scope `interface` denotes a constraint on the interface, and is checked at every observable state change, i.e., on boundaries of public method calls; MOP interface-scoped properties are therefore similar to class invariants in JML [36]. The scope `method` is used to specify constraints on the designated method: pre-, post-, and exceptional conditions. The default scope is “assertion” or “check point”: the generated monitoring code replaces the specification and is therefore checked whenever reached during the execution.

The *logic name* designates the formalism to use in the specification and also identifies the corresponding logic-plugin. Logic-plugins should have different names. Presently, the following logic names can be used in JavaMOP: JML, ERE, FTLTL and PTLTL. If no logic is designated, the specification is regarded as a *raw MOP specification*, where the user provides his/her own code to monitor the desired property. This is explained in detail in Section 4.4.

## 4.2 Body: Describing Properties

The body of an MOP specification defines the desired property, and is sent to the corresponding logic-plugin by the specification processor. It starts with an optional name and an optional list of parameters. The name, if provided, can be useful for documentation purposes or as a reference; otherwise, the MOP tool will generate a unique internal name. The parameters can only be combined with global properties. MOP provides a generic, logic-independent way to add parameters to specifications, discussed in Section 5.

Considering the diversity of specification formalisms, it is difficult, and also undesirable, to design a uniform syntax for all possible formalisms. So the syntax of the specification body varies with the underlying formalism. For JML, we adopted its original syntax. Since formalisms used to express properties over traces, such as ERE and LTL, show many common features, we designed a general syntax for all of them, shown in Figure 9. The body is composed of an optional block for local variable declarations, a list of event definitions and a formula specifying the desired property.

As discussed in Section 3.1, an execution trace is abstracted as a sequence of events generated dynamically; they usually correspond to concrete actions, e.g., invocation of certain methods or updates of some variables, and contain relevant information about the state of the program, e.g., values of accessed variables. Events are regarded *atomic* and *unique*. In other words, two events are different even when

they are generated at the same point. When two events are generated at the same point, the user should *not* assume any pre-determined order between them, even though the underlying instrumentation mechanism, e.g., AspectJ, may impose some implementation-specific ordering. Properties of traces are then defined in terms of events. For example, the property specified in Figure 1 involves two types of events, namely, the end of the execution of `authenticate()` and the beginning of the execution of `access()`. Definitions of events are orthogonal to the particular formalism used for the property.

Events are related to entries and exits of actions during the execution. An action can be one of calling a method (in the caller’s context), executing a method (in the callee’s context) and updating a variable. A `with` clause can be attached to an event to fetch the return value of the event, i.e., the value returned from a method call or a variable update. In parametric specifications, events can be parametric; the event parameters, if any, must be a subset of the parameters of the specification. A boolean expression can be associated with each event, acting as a condition: the event is generated only if the boolean expression evaluates to true.

To capture the defined events at runtime, MOP tools need to statically insert the monitors at appropriate points in the original program. AOP plays a critical role here: the MOP tool chooses instrumentation points according to the event definitions and then uses the AOP compiler to integrate the monitor into the program. In order to ease the translation from event definitions to join points in AOP, the syntax of the `⟨Method⟩` and `⟨Field⟩` may adopt the syntax of the employed AOP tool. For example, JavaMOP uses AspectJ syntax.

Events can be used as atoms in formulae. During monitor synthesis, the language shell extracts and sends the formula to the logic engine, which then generates the monitoring code from the formula. The monitor generated by the logic engine can use some pseudo code that is independent of any specific programming language. It will then be translated into the target language by the language shell. Therefore, the syntax of the formula varies with the formalisms. No formula is needed for raw MOP specifications.

The developer can declare local variables in the specification and associate actions to event definitions. The declared variables are called *monitor variables* and are only visible inside the monitor. They can be used in event actions and in handlers. *Event actions* can be any code and are executed upon occurrences of the corresponding events. Using monitor variables and event actions, one can specify more complex properties and implement more powerful handlers. For example, one may add counters into regular expressions to express properties like  $AB^3A$ . Events defined in the specification body are also monitor variables: they can be used in event actions and handlers as boolean variables.

## 4.3 Handlers: Taking Actions

MOP users can provide special code to be executed when the property is violated or validated. Although many errors

are related to violations, sometimes it is easier to define patterns of erroneous behaviors (e.g., patterns of security attacks): the match, or validation, of the pattern means “error”. In MOP, handlers can be associated not only to violations but also to validations of properties. Even though handlers support runtime error recovery, they need not necessarily be “error recovery” code. An MOP specification can therefore be regarded as a complex branch statement with the specified property (which can refer to past and future events) as the condition and the handlers as true/false branches.

The handlers use the target programming language and will be part of the generated monitoring code. Since monitors are synthesized and integrated into the program after one writes the handler code, the handlers do not have full access to information about the context in which the monitor will be executed. To mitigate this problem, MOP provides several built-in variables and commands: `@this` refers to the current object; `@RESET` resets the state of the monitor to the initial state; `@LOC` refers to the current location (file and line number) – different events take place at different locations. These variables are replaced with appropriate values or pieces of code during monitor synthesis. For example, `@this` in Figure 1 is renamed to `thisObject` in Figure 2.

#### 4.4 Raw MOP Specifications

MOP encourages the use of logical formalisms to specify desired system behaviors concisely and rigorously. However, there are cases where one may want to have full control over the monitoring process; for example, some properties can be difficult or impossible to specify using existing logical formalisms, or existing logic-plugins generate inefficient monitoring code. Moreover, there may be developers who wish to benefit from monitoring but who are not trained to or are not willing to write formal specifications, preferring instead to use the programming language that they are familiar with.

MOP supports *raw specifications* to implement and control the monitoring process exclusively by ordinary programming, without any reference to or use of logic formalisms and/or logic-plugins. As an example, Figure 10 shows a raw MOP specification that detects SQL-injection attacks [3]: malicious users try to corrupt a database by inserting unsafe SQL statements into the input to the system.

In SQL injection, a string is “tainted” when it depends upon some user input; when a tainted string is used as a SQL query, it should be checked to avoid potential attacks. In Figure 10, a `HashSet` is declared to store all tainted strings. Three types of events need to be monitored: `userInput` occurs when a string is obtained from user input (by calling `ServletRequest.getParameter()`); `propagate` occurs when a new string is created from another string; finally, `usage` occurs at using a string as a query.

Appropriate actions are triggered at observed events: at `userInput`, the user input string is added to the tainted set; at `propagate`, if the new string is created from a tainted string then it is marked as tainted, too; at `usage`,

```

/*@
scope = global
{
[Set taintedStrings = new IdentitySet();]
event userInput :
  end(call(String ServletRequest.getParameter(...))
    with (String tainted)
    { taintedStrings.put(tainted); }
event propagate :
  end(call(StringBuffer StringBuffer.new(String s))
    with (StringBuffer newS)
  \ end(call(StringBuffer StringBuffer.append(String s))
    with (StringBuffer newS)
    ...
    { if (taintedStrings.contains(s))
      taintedStrings.put(newS.toString()); }
event usage :
  begin(call(* Statement.executeQuery(String s))
    { if taintedStrings.contains(s) Util.checkSafeQuery(s); }
}
@*/

```

Figure 10. Raw MOP specification for SQL injection

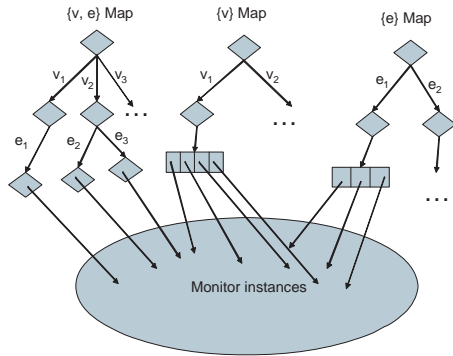
if the query string is tainted then a provided method, called `Util.checkSafeQuery`, is called to check the safety of the query. Thus the safety check, which can be an expensive operation, is invoked dynamically, on a by-need basis. In particular, for efficiency and separation of concerns reasons, a developer may even ignore the SQL injection safety aspect when writing code; the raw MOP specification above will take care of this aspect entirely.

This example shows that the event/action abstraction provided by raw MOP specifications is easy to master and useful for defining interesting safety properties compactly and efficiently. Event names were not needed here, so they could have been omitted. No formulae or violation/validation handlers are needed in raw MOP specifications; the developer fully implements the monitoring process by providing event actions using the target programming language.

All logical MOP specifications can be translated into raw specifications; in other words, each specification formalism can be regarded as *syntactic sugar* within the raw MOP specification language. MOP thus provides a focused and expressive AOP language for specifying safety properties and enforcing them by means of monitoring and recovery. However, the correctness of raw specifications is solely based on the capability of the developer to understand and implement the safety requirements. Formal specifications and their corresponding logic-plugins, on the other hand, can be assumed (and even formally proved) to generate correct monitoring code for the specified property. In Section 6 we use raw MOP specifications to implement “hand-optimized” monitors.

## 5. Adding Parameters to Specifications

As discussed in Section 1.1, many safety properties of interest in OO applications refer to groups of objects rather than to individual objects. It is, however, a nontrivial matter to support and efficiently monitor such parametric specifications. A natural solution is to use powerful logics that allow universally quantified formulae  $(\forall \vec{p})\varphi$  and to treat parametric specifications as particular formulae universally quantified over the parameters. The challenge that techniques



**Figure 11.** Centralized indexing for MOP spec in Figure 4

based on this “super-logic” approach face is how to synthesize an efficient monitor from a universally quantified formula. Several runtime verification systems follow this approach explicitly or implicitly, including Eagle/Hawk [23], Tracematches [2], and PQL[38]. MOP does *not* prevent the logic designer from employing logics with universal quantifiers: once a logic-plugin is implemented for such a logic, the logic can be immediately used to specify parametric properties in MOP. For example, Eagle or the publicly available monitoring algorithms of PQL and Tracematches can be organized as MOP logic-plugins with little effort.

Synthesizing *efficient* monitors from formulae in logics allowing quantification is hard. Such monitors need to keep track of all the instances of all the quantified variables. Large hash tables or other similar structures may need to be generated, nested, garbage-collected and accessed multiple times per event, making it difficult to maintain an acceptably low runtime overhead in real-life applications. Even if one disallows nested quantifiers in formulae and even if one knows how to monitor an unquantified formula  $\varphi$  efficiently, it is still non-trivial to monitor the quantified formula  $(\forall \vec{p})\varphi$ .

We next describe a novel *logic-independent* technique to support universal, non-nested parameters in specifications using any trace-related logics with *no need* to modify the existing monitoring generation algorithm. One is then able to write parametric specifications using any of the existing logic-plugins in MOP. One would expect that such a genericity must come at a performance price. However, as shown in Section 6, our generic technique presented next, when used with the ERE logic-plugin, produces significantly less runtime overhead than Tracematches with all its optimizations (including static ones) enabled (see Table 4).

In our solution, a monitor instance checking the specified property will be created for every specific group of values of parameters; if a monitor instance  $m$  is created for a group of values containing  $o$ , then we say that  $m$  is *related* to  $o$ . For the SafeEnum specification in Figure 4, a monitor instance will be created for every pair of concrete  $v$  and  $e$  if  $e$  is the enumeration of  $v$ . When a relevant event occurs, concrete values are bound to the event parameters and used to look up related monitor instances; related monitors are then

invoked to handle the observed event. Several monitors can be triggered by an event since the event may contain fewer parameters than the parameters of the enclosing specification. For the SafeEnum example, when an `updatesource` event occurs, the target Vector object is bound to the parameter  $v$  and used to find all the related monitors to process `updatesource` (there may be several enumerations of  $v$ ).

The monitor lookup process is external to the monitor in our approach and makes no assumption on the implementation of the monitor; consequently, it is independent of the monitor generation algorithm. Also, the monitor does not need to be aware of the parameter information and can proceed solely according to the observed event. Hence, the monitoring process for parametric specifications is divided into two parts in MOP: the logic-specific monitor (generated by the logic plugin) and the logic-independent lookup process (synthesized by the specification processor). Consequently, given any logic-plugin, MOP allows one to write parametric specifications using that logic with no additional effort.

Current runtime verification approaches supporting logics with universal quantifiers construct a centralized monitor whose state evolves according to the parameter information contained in received events. Our approach, on the contrary, creates many isolated monitor instances, but it maintains indexing information so that it can quickly find relevant monitors. Experiments (Section 6) show that our “decentralized-monitoring” strategy performs overall better than the centralized ones. The rest of this section presents two instances of our decentralized monitoring technique, both supported by JavaMOP: one using centralized indexing and the other using decentralized indexing.

## 5.1 Centralized Indexing

Efficient monitor lookup is crucial to reduce the runtime overhead. The major requirement here is to quickly locate all related monitors given a set of parameter instances. Recall that different events can have different sets of parameters: e.g., in Figure 4, all three events declare different parameter subsets. Our centralized indexing algorithm constructs multiple indexing trees according to the event definitions to avoid inefficient traversal of the indexes; more specifically, for every distinct set of event parameters found in the specification, an indexing tree is created to map the set of parameters directly into the list of corresponding monitors.

The number and structure of indexing trees needed for a specification can be determined by a simple static analysis of event parameter declarations. For example, for the parametric specification in Figure 4, since there are three different sets of event parameters, namely  $\langle v, e \rangle$ ,  $\langle v \rangle$  and  $\langle e \rangle$ , three indexing trees will be created to index monitors, as illustrated in Figure 11: the first tree uses a pair of  $v$  and  $e$  to find the corresponding monitor, while the other two map  $v$  and, respectively,  $e$  to the list of related monitors.

We use hash maps in JavaMOP to construct the indexing tree. Figure 12 shows the generated monitor look up code

```

Map SafeEnum_v_map = makeMap();
pointcut SafeEnum_updatesource0(Vector v) :
    call(* Vector.add*(..)&& target(v);
after (Vector v) : SafeEnum_updatesource0(v) {
    Map m = SafeEnum_v_map;
    Object obj = null;
    obj = m.get(v);
    if (obj != null){
        Iterator monitors = ((List)obj).iterator();
        while (monitors.hasNext()) {
            SafeEnumMonitor monitor=(SafeEnumMonitor)monitors.next();
            monitor.updateSource(v);
            if (monitor.succeeded()) {
                //validation handler
            }
        }
    }
}

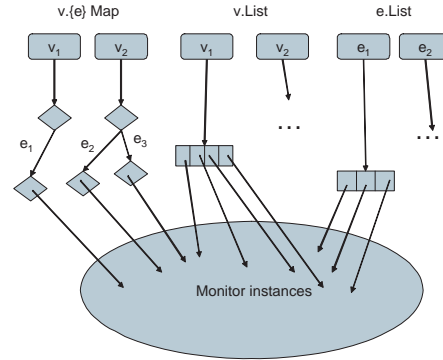
```

**Figure 12.** Centralized indexing monitoring code generated by JavaMOP for `updateSource` (from spec in Figure 4)

for the `updateSource` event in Figure 4. This code is inserted at the end of every call to `Vector.add` or other vector changing methods, according to the event definition. One parameter is associated to this event, namely, the vector `v` on which we invoke the method. A map, `SafeEnum_v_map`, is created to store the indexing information for `v`, i.e., the  $\{v\}$ Map in Figure 11. When such a method call is encountered during the execution, a concrete vector object will be bound to `v` and the monitoring code will be triggered to fetch the list of related monitors using `SafeEnum_v_map`. Then all the monitors in the list will be invoked to process the event.

An important question is when to create a new monitor instance. This is a non-trivial problem in its full generality, because one may need to create “partially instantiated” monitors when events with fewer parameters are observed before events with more parameters. While this partial instantiation can be achieved in a logic-independent manner, motivated by practical needs we adopted a simpler solution in JavaMOP: we let the logic-plugin tell which events are allowed to create new monitors; these events are also required to be parametric by all the specification parameters, such as the `create<v, e>` event in Figure 4. All MOP’s logic-plugins have been extended to mark their monitor-initialization events. Thus, if an event is generated and a monitor instance for its parameters cannot be found, then a new monitor instance is created for its parameters only if the event is marked; otherwise the event is discarded. This way, no unnecessary monitor instances are created; indeed, it would be pointless and expensive to create monitor instances for all vector updates just because they can be potentially associated with enumerations – monitor instances are created only when enumerations are actually created.

A performance-related concern in our implementation of JavaMOP is to avoid memory leaks caused by hash maps: values of parameters are stored in hash maps as key values; when these values are objects in the system, this might prevent the Java garbage collector from removing them even when the original program has released all references to them. We use weakly referenced hash maps in JavaMOP.



**Figure 13.** Decentralized indexing for monitor in Figure 11

The weakly referenced hash map only maintains weak references to key values; hence, when an object that is a key in the hash map dies in the original program, it can be garbage collected and the corresponding key-value pair will also be removed from the hash map. This way, once a monitor instance becomes unreachable, it can also be garbage collected and its allocated memory released.

Note that a monitor instance will be destroyed *only* when it will never be triggered in the future. Since we have an indexing tree per event parameter set, if a monitor  $m$  can potentially be triggered in the future by some event  $e$  with a parameter set  $(p_1, \dots, p_n)$ , where  $n$  can also be 0, then:

1.  $m$  appears in the indexing tree corresponding to the parameters  $(p_1, \dots, p_n)$ ; that is also because of our assumption/limitation that, when  $m$  is created, all its possible parameters, including  $p_1, \dots, p_n$  but potentially more, were available; when  $m$  was created, it was added to all the indexing trees corresponding to (subsets of) its parameters, including that of  $(p_1, \dots, p_n)$ ; and
2. if  $e$  is ever generated in the future,  $m$  will be referred from the indexing tree for  $(p_1, \dots, p_n)$ . This is because if  $e$  really occurs at some moment in the future, then  $p_1, \dots, p_n$  should all be live objects and thus the mapping in the corresponding indexing tree has not been destroyed.

Therefore, if a future event can ever trigger  $m$ , then  $m$  is not garbage collectible. This guarantees the soundness of our usage of weak references. One interesting corner case here is when  $n$  is 0, i.e., when some event has no parameter. In such case, the corresponding indexing tree (for the empty set of parameters) is actually a list instead of a map. Thus, even if all parameters die, the monitor will still be kept alive because there is a reference to it in that list. But this only happens when at least one of the events in the specification has no parameters.

## 5.2 Optimization: Decentralized Indexing

The centralized-indexing-decentralized-monitor approach above can be regarded as a centralized database of monitors. This solution proves to be acceptable wrt runtime overhead in many of the experiments that we carried out; in particular,

```

List Vector.SafeEnum_v_List = null;
pointcut SafeEnum_updatesource0(Vector v) :
    call(* Vector.add*(..)&& target(v);
after (Vector v) : SafeEnum_updatesource0(v) {
    if (v.SafeEnum_v_List != null) {
        Iterator monitors = (v.SafeEnum_v_List).iterator();
        while (monitors.hasNext()) {
            SafeEnumMonitor monitor=(SafeEnumMonitor)monitors.next();
            monitor_updatesource(v);
            if (monitor.succeeded()) {
                //validation handler
            }
        } //end of while
    }
}
}

```

**Figure 14.** Decentralized indexing monitoring code automatically generated by JavaMOP for `updatesource`

it compares favorably with centralized-monitor approaches (see Figure 15). However, reducing runtime overhead is and will always be a concern in runtime verification. We next propose a further optimization based on decentralizing indexing. This optimization is also implemented in JavaMOP.

In *decentralized indexing*, the indexing trees are piggybacked into states of objects to reduce the lookup overhead. For every distinct subset of parameters that appear as a parameter of some event, JavaMOP automatically chooses one of the parameters as the *master parameter* and uses the other parameters, if any, to build the indexing tree using hash maps as before; the resulting map will then be declared as a new field of the master parameter. For example, for the `updatesource` event in Figure 4, since it has only the `<v>` parameter, `v` is selected as master parameter and a new field will be added to its `Vector` class to accommodate the list of related monitor instances at runtime. Figure 13 shows the decentralized version of the centralized indexing example in Figure 11, and Figure 14 shows the generated decentralized indexing monitoring code for the `updatesource` event.

Comparing Figures 14 and 12, one can see that the major difference between the centralized and the decentralized indexing approaches is that the list of monitors related to `v` can be directly retrieved from `v` when using decentralized indexing; otherwise, we need to look up the list from a hash map. Decentralized indexing thus scatters the indexing over objects in the system and avoids unnecessary lookup operations, reducing both runtime overhead and memory usage. It is worth noting that decentralized indexing does *not* affect the behavior of disposing unnecessary monitor instances as discussed in the previous section: when an object is disposed, all the references to monitor instances based on this object will also be discarded, no matter whether they are stored in maps using weak references or whether they are embedded as fields of the object.

On the negative side, decentralized indexing involves more instrumentation than the centralized approach, sometimes beyond the boundaries of the monitored program, since it needs to modify the original signature of the master parameter: for the monitoring code in Figure 14, the Java library class `Vector` has to be instrumented (add a new field).

This is usually acceptable for testing/debugging purposes, but may not be appropriate if we use MOP as a development paradigm and thus want to leave monitors as part of the released program. If that is the case, then one should use centralized indexing instead, using the attribute `centralized`.

The choice of the master parameter may significantly affect the runtime overhead. In the specification in Figure 4, since there is a one-to-many relationship between vectors and enumerations, it would be more effective to choose the enumeration as the master parameter of the `create` event. Presently, JavaMOP picks the first parameter encountered in the analysis of the MOP specification as the master parameter for each set of event parameters. Hence, the user can control the choice of the master parameter by putting, for each set of parameters  $P$ , the desired master parameter first in the list of parameters of the first event parametric over  $P$ .

Decentralized indexing is not an entirely new technique for reducing monitoring overhead; it is actually a natural extension to our previous MOP framework [21], which stores monitors' states as fields in objects when checking class invariants, i.e., one-parameter properties. Within the context of multiple parameters (the major concern of this paper), indexing information is embedded into the objects instead of states. A similar idea of piggybacking the indexing information into objects states was proposed in [8], called "inter-type declaration", but it had not been implemented when this paper was written.

## 6. Experiments and Evaluation

We have applied JavaMOP on tens of programs, including several large-scale open source programs, e.g., the DaCapo benchmark suite [14], the Tracematches benchmark suite [8], and Eclipse [25]. Our evaluation mainly focuses on two aspects: the expressivity of the specification language and the runtime overhead of monitoring. The properties used in our experiments come from two sources: properties used in other works (e.g., [27, 38, 8, 16]) and our own formalization of informal descriptions in software documentation.

With the currently supported logic-plugins and the generic support for parameters, JavaMOP is able to formally and concisely express most of the collected properties. One interesting exception is the SQL injection from PQL [38], which we implemented using the raw MOP specification shown in Figure 10. A large portion, nearly half, of the properties that we have tried are recoverable/enforceable. Many violations of properties were revealed in our experiments, although we did not focus on error detection; when violations occurred, we were able to quickly locate their causes using JavaMOP.

The rest of this section focuses on performance evaluation, on discussing some of the detected violations, and on current limitations of our implementation.

### 6.1 Performance Evaluation

The monitoring code generated by JavaMOP caused low runtime overhead, below 10%, in most experiments even



with centralized indexing. By turning on the decentralized indexing, few experiments showed noticeable runtime overhead. In what follows, we evaluate JavaMOP's runtime overhead using the DaCapo benchmark, and also compare JavaMOP with other runtime verification techniques, namely, Tracematches and PQL, using the Tracematches benchmark.

Our experiments were carried out on a machine with 1GB RAM and P4 2.0Gz processor; the Sun Java HotSpot(TM) Client VM (1.5.0\_10) on Windows XP professional was used as the running JVM. All the benchmark programs and properties discussed in this paper can be downloaded from JavaMOP's website [20].

We used the DaCapo benchmark version 2006-10; it contains eleven open source programs [14]: `antlr`, `bloat`, `chart`, `eclipse`, `fop`, `hsqldb`, `jython`, `luindex`, `lusearch`, `pmd`, and `xalan`. The provided default input was used together with the `-converge` option to execute the benchmark multiple times until the execution time falls within a coefficient of variation of 3%. The average execution time is then used to compute the runtime overhead.

### 6.1.1 Properties

The following general properties borrowed from [16] were checked using JavaMOP:

1. `SafeEnum`: Do not update `Vector` while enumerating its elements using the `Enumeration` interface (Figure 4);
2. `SafeIterator`: Do not update a `Collection` when using the `Iterator` interface to iterate its elements;
3. `HashMap`: The hash code of an object should not be changed when the object is used as a key in a hash map;
4. `HasNext`: Always call the `hasNext()` method of an iterator before calling its `next()` method;
5. `LeakingSync`: Only access a `Collection` via its synchronized wrapper once the wrapper is generated by the `Collections.synchronized*` methods;
6. `ClosedReader`: Do not read from a `Reader` if it or its corresponding `InputStream` has been closed;

More properties have been checked in our experiments; we choose these six properties to include in this paper because they generate a comparatively larger runtime overhead. We excluded those with little overhead. Three of these properties are recoverable: `HashMap` (the monitor can maintain a shadow map based on `IdentityHashMap` as backup), `HasNext` (make a call to `hasNext()` before `next()`), and `LeakingSync` (redirect call to the synchronized wrapper).

For every property, we provided three MOP specifications: an ERE formal specification, the same formal specification for centralized indexing, and a (hand-optimized) raw MOP specification. The last one is supposedly the best monitoring code for that property and was used to evaluate the effectiveness of our monitor generation algorithm. The As-

pectJ compiler 1.5.3 (AJC) was used in these experiments to compile the generated monitoring AspectJ code.

### 6.1.2 Statistics and Results of the Evaluation

Tables 1 and 2 show the instrumentation and monitoring statistics for monitoring the above properties in DaCapo: Table 1 gives the number of points statically instrumented for monitoring each of the properties; Table 2 gives the number of events and the number of monitor instances generated at runtime using centralized indexing. Both these numbers are collected from a single execution of the benchmark. The first row in each table gives the names of the properties, and the first column in Table 2 gives the programs. We do not split the static instrumentation points by different programs because they are merged together in the benchmark suite; some of them even share common packages. Decentralized indexing does not change the number of generated events or monitor instances; it only affects the monitor indexing.

These two tables show that the properties selected in our experiments imposed heavy runtime monitoring on the programs: a large number of points, ranging from one thousand to twelve thousand, in the original programs were instrumented to insert the monitoring code. The monitoring code was frequently triggered during the execution, especially for those properties involving the Java `Collection` classes, e.g., `SafeIterator`, `HashMap`, and `HasNext`. Some properties generated numerous runtime checks but only a few, even zero, monitor instances were created (e.g., `SafeEnum` and `LeakingSync`). The reason is that these properties observe some frequently visited methods, but the events that we allowed to create monitor instances rarely or never occurred. For example, `LeakingSync` checks all the method calls on the `Collection` interface, but no calls to `Collections.synchronized*` methods happened in these experiments, so no monitor-initialization events were created. Such experiments are particularly useful to evaluate the effectiveness of the generated monitoring code to filter dynamically irrelevant events, i.e., events that have no effect on the current monitor states. Also, a big difference between the number of events and the number of created monitor instances (e.g., `jython-SafeEnum` and `bloat-Leakingsync`) indicates a real potential for static analysis optimizations.

Table 3 summarizes the runtime overhead measured in our experiments, represented as a *slowdown percentage* of the monitored program over the original program. For every property-program combination, three monitoring runtime overhead numbers are given: with centralized indexing, with decentralized indexing, and using a hand-optimized raw MOP specification. Among all 66 experiments (recall that we already excluded some results with little overhead), only 11 (bold) caused more than 10% slow-down with centralized indexing; for the decentralized indexing version, this number reduces to 4. Except for the 4 worst cases, with decentralized indexing JavaMOP generates monitoring code *almost as efficient as the hand-optimized code*.



	SafeEnum	SafeIterator	HashMap	HasNext	LeakingSync	ClosedReader
DaCapo	1147	5663	1729	2639	12855	2966

**Table 1.** Instrumentation statistics: instrumentation points in the DaCapo benchmark

	SafeEnum		SafeIterator		HashMap		HasNext		LeakingSync		ClosedReader	
antlr	10K	0	1K	0	0	0	0	0	233K	0	3M	1K
bloat	0	0	90M	1M	391K	46K	155M	1M	6M	0	11K	0
chart	57	0	569K	815	8K	3K	6K	815	653K	0	208	2
eclipse	16K	0	38K	31	31K	19K	1K	31	230K	0	29K	165
fop	7	1	49K	79	17K	6K	277	79	3M	0	1K	3
hsqldb	174	0	0	0	0	0	0	0	686	0	0	0
jython	50K	0	174K	50	443	439	106	50	16M	0	1M	114
luindex	457K	14K	82K	8K	9K	9K	28K	8K	3M	0	19K	0
lusearch	335K	0	405K	0	416	416	0	0	1M	0	2M	0
pmd	717	0	25M	1M	11K	105	46M	8M	26M	0	28K	4
xalan	5K	0	119K	0	124K	78K	0	0	682K	0	98K	1K

**Table 2.** Monitoring statistics: generated events(left column) and monitor instances(right column). K =  $\times 10^3$ , M =  $\times 10^6$ 

	SafeEnum			SafeIterator			HashMap			HasNext			LeakingSync			ClosedReader		
antlr	0.0	0.0	1.5	0.0	0.0	0.0	0.0	0.0	1.1	0.0	0.4	0.0	2.7	0.0	0.0	<b>22.1</b>	<b>5.8</b>	<b>0.0</b>
bloat	2.4	0.0	0.0	<b>385</b>	<b>176</b>	<b>24.2</b>	2.4	1.8	1.4	<b>323</b>	<b>154</b>	<b>36.3</b>	<b>13.5</b>	<b>3.2</b>	<b>2.2</b>	0.1	0.0	2.3
chart	0.0	0.0	0.0	0.3	0.0	0.0	4.8	3.6	4.8	0.0	0.0	0.0	0.1	0.5	0.0	0.0	0.0	0.0
eclipse	2.4	4.1	0.8	0.0	0.0	1.4	3.6	3.7	0.5	0.0	3.8	1.5	0.8	3.0	3.1	0.6	2.2	2.4
fop	0.4	1.2	0.6	1.7	1.5	0.0	0.0	0.0	0.0	1.7	0.8	1.5	<b>14.7</b>	<b>0.5</b>	<b>1.0</b>	1.9	0.0	0.0
hsqldb	0.0	3.3	0.0	0.0	0.9	1.2	0.0	0.0	2.1	0.0	0.8	0.0	1.1	1.4	1.4	1.6	0.0	0.0
jython	0.5	0.6	0.0	1.6	0.8	0.5	0.7	0.2	0.3	1.3	0.0	0.6	<b>30.2</b>	<b>0.0</b>	<b>2.3</b>	0.7	0.4	0.2
luindex	2.6	1.6	0.2	3.2	1.9	0.5	0.6	1.2	1.8	0.9	0.3	0.0	4.3	3.2	2.2	1.1	1.7	1.1
lusearch	6.6	0.5	0.0	9.5	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.0	<b>32.4</b>	<b>1.1</b>	<b>0.6</b>	<b>75.7</b>	<b>0.0</b>	<b>0.1</b>
pmd	0.0	0.0	0.0	<b>272</b>	<b>44.8</b>	<b>11.3</b>	0.5	0.0	0.0	<b>353</b>	<b>25.4</b>	<b>13.7</b>	<b>34.3</b>	<b>5.4</b>	<b>8.0</b>	0.0	0.0	0.0
xalan	0.0	3.5	4.4	4.8	6.7	5.4	7.2	4.7	6.5	4.6	0.0	2.8	3.0	1.5	1.7	8.5	2.2	4.5

**Table 3.** Runtime overhead (in percentage; e.g., 14.7 means 14.7% slower) of JavaMOP: centralized | decentralized | raw

Analyzing Tables 3 and 2, one can see that decentralized indexing handles the dynamically irrelevant events much better than centralized indexing, e.g., when checking the LeakingSync property. This is caused by the fact that, when there is no related monitor instance, decentralized indexing only checks an object field, while centralized indexing needs to make an expensive hash map lookup. The runtime overhead is determined not only by the frequency of reaching monitoring code, but also by the execution time of the monitored action. For example, HashMap required quite heavy monitoring on many programs but did not cause any noticeable performance impact. This is because the methods checked for HashMap, including put, remove, and contains, are relatively slow. On the other hand, checking bloat and pmd against SafeIterator and HasNext is as bad as it can be: the monitored actions take very little time to execute (e.g., the hasNext and next methods of Iterator) and they are used very intensively during the execution (indicated by the massive numbers in Table 2). Even for such extreme cases, the monitoring code generated by JavaMOP with decentralized indexing may be considered acceptable: slowdown between 2 and 3 times. However, one can always

choose to implement a hand-optimized raw MOP specification for the property of interest; in our case, the raw MOP specification reduced the runtime overhead to only 20-30%.

### 6.1.3 Comparing JavaMOP, Tracematches, and PQL

Attempts have also been made to compare JavaMOP with other existing trace monitoring tools. However, some of them are not publicly available, others have limitations that prevented us from using them in our experiments. Consequently, we only succeeded to compare JavaMOP thoroughly with Tracematches and partially with PQL.

As shown in [6], Tracematches is one of the most efficient and mature trace monitoring tools to date. A benchmark for trace monitoring tools and experiments has been proposed by the Tracematches team in [8] (a March 2006 version provided with ABC 1.2.1<sup>2</sup>), containing eight property-program combinations. Detailed explanations of these properties and programs can be found in [8, 9]; one of them had 0 runtime overhead and apparently was not intended to measure run-

<sup>2</sup>The Tracematches numbers in Table 4 were actually obtained using a recent experimental Tracematches implementation provided to us by its developers before the final version of this paper.

Property	Program	LOC	Original (seconds)	Hand Optimized	JavaMOP	JavaMOP-CI	TM	PQL
Listener	ajHotDraw	21.1K	1.3	0.0	6.7*	139.3	354.0	2193.3
SafeEnum	jHotDraw	9.5K	3.3	0.1	136.3	509.1	1509.1	7084.8
NullTrack	CertRevSim	1.4K	0.12	210.1	-	232.0	452.7	n/a
Hashtable	Weka	9.9K	2.8	3.3	3.3	6.7	15.2	n/a
HashSet	Aprove	438.7K	560.0	21.2	23.9	51.9	124.3	n/a
Reweave	ABC	51.2K	7.0	11.1	-	20.2	63.5	n/a

**Table 4.** Runtime overhead (in %) for JavaMOP, Tracematches, and PQL on the Tracematches Benchmark. (\*: A class-scoped specification was used. -: Decentralized indexing monitoring cannot be applied for these properties because it requires instrumentation on non-modifiable classes in Java, e.g., `Object` and `String`.)

time overhead, and another makes use of an ABC-specific feature (stack access, not provided by AJC) that we decided not to support in order to keep JavaMOP compiler-independent. Table 4 shows the results that we obtained for the other six property-program combinations. These experiments were run on the same machine mentioned above. Each program was executed five times and the average running time was used in computing the runtime overhead.

The first two columns in Table 4 list the properties and the programs; the third column gives the sizes of the programs; the fourth shows the running time of the original, unmonitored program; the remaining columns give the runtime overhead caused by hand-optimized monitors, (decentralized indexing) JavaMOP monitors, centralized indexing JavaMOP monitors, Tracematches monitors, and PQL monitors. We take no credit for the hand-optimized code: it was implemented by Tracematches developers using AspectJ and offered with the benchmark. The Tracematches properties were also contained in the benchmark package. To achieve a direct comparison, all the MOP specifications used the ERE logic-plugin and were essentially identical to the Tracematches specifications. The Tracematches specifications and the monitoring code generated by JavaMOP were compiled with the ABC compiler [7] for direct comparison. Due to implementation limitations of PQL, only two properties could be specified using PQL; we could not apply the static analyzer in the PQL distribution due to lack of documentation.

Two properties in these experiments, `NullTrack` and `Reweave`, involve non-modifiable Java classes (`Object` and `String`) as parameters. Consequently, the generated decentralized indexing code cannot be weaved into the original programs. In such cases, we applied only centralized indexing monitoring. Also, for the `Listener` property, we used a class invariant for the decentralized mode equivalent to the two-parameter specification proposed by Tracematches (and checked in the centralized mode), resulting in a significant performance difference between the two modes. This strengthens our belief that one should be provided the freedom to choose what suits one’s needs most, in particular one should not have to write parametric specifications if the property is essentially a class invariant.

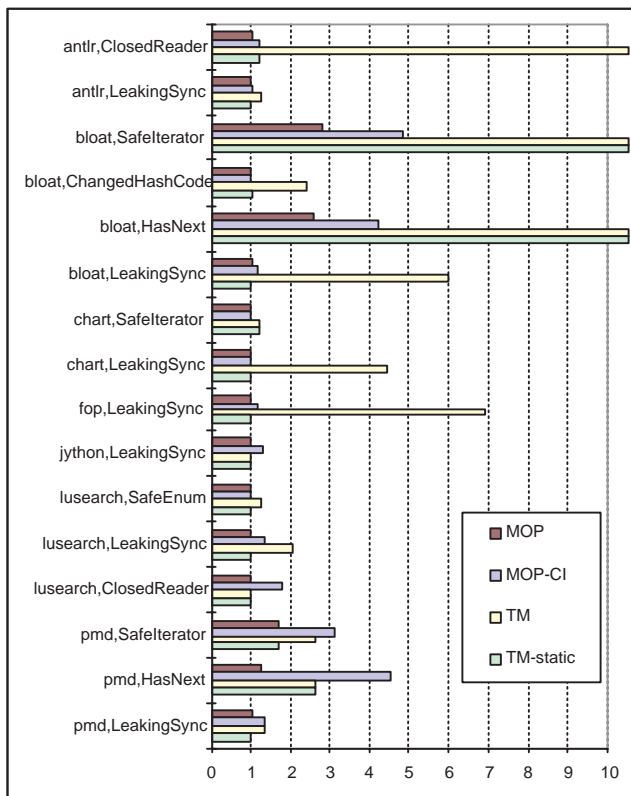
Table 4 shows that JavaMOP generates more efficient monitoring code than Tracematches and PQL, often close to the hand-optimized code when using decentralized indexing. Since JavaMOP generates *standard* AspectJ code, it gives us the freedom to choose off-the-shelf compilers. In our experiments, ABC tended to take more time to compile the code than AJC, e.g., it took ABC nearly an hour to compile Aprove but AJC needed only a few minutes.

PQL and Tracematches have their own strengths and the above comparison should not be interpreted as an argument against them. PQL provides a general specification formalism extending context-free grammars; it is therefore not surprising that it generates a larger runtime overhead. Tracematches implements a sound and specialized algorithm to support universally quantified regular patterns. The parametric framework discussed in our paper is generic and logic-independent, therefore the present JavaMOP implementation does not provide any logic-specific optimizations or specializations like those in Tracematches.

The paper [9] discusses in some depth differences between JavaMOP[ERE,inline,validation,centralized] and Tracematches, and also shows the results of their performance comparison between the two systems on the same property-program pairs as ours. However, their numbers are rather different. Briefly, the results in [9] show that JavaMOP-CI and Tracematches overall have comparable runtime overheads, while our experiments show that JavaMOP-CI produces significantly less overhead than Tracematches. The explanation derived from discussions with the Tracematches developers is that these apparently conflicting results are mainly caused by the different JVM implementations on Linux and Windows. However, it also appears that a new feature, “frequency”, supported by an experimental, not-yet-public implementation of Tracematches was utilized in their experiments. This feature allows one to manually tune a specification to reduce the runtime overhead. Because the Tracematches benchmark suite does not make use of this feature, because of its limited benefit (it only helps in the case of `SafeEnum`), and ultimately because it and an example using it were made available to us only the day before the deadline for the final version of this paper, we have not considered it in our experiments yet.

### 6.1.4 Comparing JavaMOP and Statically Optimized Tracematches

An important advantage of building a runtime verification tool on top of an instrumentation package, like Tracematches and PQL do, is that one can have more control over instrumentation and thus facilitate the use of static analysis. A static analyzer has been recently proposed for Tracematches in [16]<sup>3</sup> and it was also evaluated on the property-program combinations using the DaCapo benchmark. This allows for another comparison, between JavaMOP and Tracematches with static analysis. The results are shown in Figure 15.



**Figure 15.** Runtime overhead of JavaMOP and Tracematches on DaCapo (CI: centralized indexing; TM: Tracematches; TM-static: TM with static analysis). The runtime overhead is represented as the ratio of monitored execution over non-monitored execution; e.g., 1 means no overhead and 10 means ten times slower.

Figure 15 compares those examples with more than 10% overhead in Table 3 or more than 20% for Tracematches without static analysis according to [16]. We did not repeat the 16 experiments for Tracematches in our environment, and all the numbers for Tracematches are taken from [16]. Without using static analysis, Tracematches caused less overhead than centralized JavaMOP monitors in four cases (Jython-LeakingSync, lusearch-ClosedReader, pmd-SafeIterator, and pmd-HasNext), and it was always less

efficient than decentralized JavaMOP monitors. After using static analysis to eliminate unnecessary instrumentation points, there are still three cases (bloat-SafeIterator, bloat-HasNext, and pmd-HasNext) in which JavaMOP outperformed Tracematches, while for the others both systems had similar performance.

We are not arguing against static analysis; on the contrary, we believe that static analysis can and should be combined with MOP to further reduce the runtime overhead, but that is out of the scope of this paper.

### 6.2 Violation Detection

As mentioned, error detection was not the main focus in our experiments; we consider that, for error detection, runtime verification needs to be combined with test case generation. However, we still encountered unexpectedly many violations during the evaluation of JavaMOP. One reason is that many safety properties in our experiments were devised for checking performance, and are therefore not strictly required to hold in all programs. Consequently, many violations do not lead to actual errors in the program. For example, violations of the `hasNext` property were found in some Java library classes, e.g., `AbstractCollection` and `TreeMap`. It turned out that these implementations use the size of the collection instead of the `hasNext` method to guard the iteration of elements. We also found violations indicating possible semantic problems of programs, which are subtle and thus difficult to find by ordinary testing. We next discuss some of these.

#### 6.2.1 Potential Errors.

There is a known problem in `jHotDraw` about using objects of `Enumeration`: one can edit a drawing, which may update a vector in the program, while making the animation for the drawing, which uses an enumerator of the vector. As expected, JavaMOP was able to find this problem.

We also found violations of some *interface contracts*, i.e., rules to use interfaces, in Eclipse. These can lead to resource leaks as pointed out in [27] and [38]. Three kinds of properties were checked in our Eclipse experiments:

1. The `dispose` method needs to be called to release acquired resources before a GUI widget is finalized.
2. The `remove*Listener` should be called by a host object to notify its listeners (registered by calling `add*Listener`) to release resources before it is finalized. \* represents the name of the listener.
3. Eclipse uses Lucene [37] as its search engine; in Lucene, it is required that, before a `Dir` object is closed (by calling its `close` method), all the file readers created by the `Dir` object should be closed.

We instrumented the GUI package of Eclipse with these three properties and also the JDT package with the second property (note that there are many different `add*Listener`-`remove*Listener` pairs in these two packages). Then we

<sup>3</sup> a newer static analysis was proposed [17] after our paper was finalized

used the instrumented Eclipse in our development work (no noticeable slow-down was experienced during the evaluation). More than 30 violations were detected in the GUI package, while none was found in the JDT package – this may indicate the importance of the second property. In summary, the GUI package, which is more complex and harder to test, seems less reliable w.r.t. to memory leaks.

### 6.2.2 Inappropriate Programming Practice

Several unexpected violations were encountered during our experiments. For example, we ran into some violations in Xalan [42] when checking a simple property about the `Writer` class in Java: no writes can occur after the writer is closed (by calling the `close` method). This is, according to the Java documentation which states that an exception should be raised, a must-have property. Despite these violations, no errors occurred in Xalan. Using JavaMOP, we located the places causing the violations without much insight of the program and a quick review showed that a pool of writer instances is used in Xalan to avoid unnecessary recreations, but the writer can be closed before it is returned to the pool. However, the program uses `StringWriter`, whose `close` method happens to have no effect. Although it is not an error in this implementation, we believe that it is inappropriate programming practice: the writer should be cleared instead of closed when returned to the pool.

### 6.3 Limitations of MOP and JavaMOP

The current MOP logic-plugins encapsulate monitor synthesis algorithms only for non-parametric trace logics. Even though the new MOP specification language allows universal parameters to be added to any of these logics, there is no way to add nested parameters, or existential ones. We intend to soon add a logic-plugin for Eagle [12], a “super-logic” generalizing both ERE and LTL, and also allowing arbitrary quantification and negation, but do not expect it to have a stimulating runtime overhead.

Our current JavaMOP implementation assumes that, in a parametric specification, the events marked by the logic-plugin to create monitor instances contain all the parameters of the specification. This limitation can be avoided by implementing a more complicated monitor creation strategy; however, we were not motivated to it because all the properties that we have checked so far fall under this restriction.

The gap between dynamic events for monitoring and static monitor integration based on AOP can lead to some limitations of MOP tools. Ideally, for variable update events, the MOP tool should instrument all the updates of involved variables. But, statically locating all such updates requires precise alias analysis. Therefore, JavaMOP only allows update events for variables of primitive types. In addition, static instrumentation may cause extra performance penalty of monitoring. For the specification in Figure 4, one can see that the monitor is not “interested” in next events after `create` until an `updatesource` event is encountered.

But since we instrument the program statically, the monitor keeps receiving next events even when they are not needed. These limitations may be relaxed by utilizing dynamic AOP tools, but more discussion on this direction is out of the scope of this paper. However, since MOP can also be used to add new functionality to a program, one may not want to miss any related event: some action may be executed even when the event does not affect the monitor state.

## 7. Conclusion

We presented a generic, logic-independent approach to support parametric specifications in Monitoring-Oriented Programming (MOP). A novel optimization technique, called decentralized indexing, was proposed to reduce the runtime overhead of monitoring parametric properties. A new, enriched MOP specification language was also proposed, that supports parameters and raw specifications; one can use raw MOP specifications to fully implement and control the desired monitoring process using the target programming language. An extensive evaluation of JavaMOP and comparisons with other runtime verification tools have been carried out; results are encouraging: less than 8% experiments showed more than 10% runtime overhead, and JavaMOP generated overall more efficient monitoring code than other runtime verification tools.

The techniques presented in this paper are purely dynamic. Although we showed that runtime verification is feasible, we also believe that static analysis can and should be used to further reduce the runtime overhead of monitoring: by statically analyzing the program against the desired property, one can eliminate irrelevant instrumentation points. Since static analysis is closely related to the particular logic-plugin, to add static analysis to MOP we will probably need *static logic-plugins*. Also, MOP can be combined with test generation techniques to provide an effective testing framework for safety properties.

**Acknowledgments.** We would like to warmly thank the current Tracematches team, namely Pavel Avgustinov, Julian Tibble, and Oege de Moor, for developing and making public the challenging Tracematches benchmark, as well as for a series of insightful discussions while preparing the final versions of our papers. We would also like to thank Eric Bodden, Laurie Hendren and Ondrej Lhotak for their contributions on the Dacapo-based benchmark suite that made the comprehensive evaluation of JavaMOP possible.

## References

- [1] P. Abercrombie and M. Karaorman. jContractor: Bytecode instrumentation techniques for implementing DBC in Java. In *RV'02*, volume 70.4 of *ENTCS*, 2002.
- [2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*, 2005.

- [3] C. Anley. Advanced SQL injection in SQL server applications. *NGSSoftware*, 2002.
- [4] AspectC++. <http://www.aspectc.org/>.
- [5] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In *ASM'03*, volume 2589 of *LNCS*, pages 87–107, 2003.
- [6] P. Avgustinov, E. Bodden, E. Hajiyev, L. Hendren, O. Lhotak, O. de Moor, N. Ongkingco, D. Sereni, G. Sittampalam, J. Tibble, and M. Verbaere. Aspects for trace monitoring. In *FATES/RV'06*, volume 4262 of *LNCS*, pages 20–39, 2006.
- [7] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. ABC: an extensible AspectJ compiler. In *AOSD'05*, 2005.
- [8] P. Avgustinov, J. Tibble, E. Bodden, O. Lhotak, L. Hendren, O. de Moor, N. Ongkingco, and G. Sittampalam. Efficient Trace Monitoring. Technical Report abc-2006-1, Oxford University, 2006.
- [9] P. Avgustinov, J. Tibble, and O. de Moor. Making Trace Monitors Feasible. In *OOPSLA'07*, 2007. this volume.
- [10] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS'04*, volume 3362 of *LNCS*, pages 49–69, 2004.
- [11] H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma. *Runtime Verification (RV'05)*. Elsevier, 2005. ENTCS 144.
- [12] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *VMCAI'04*, volume 2937 of *LNCS*, pages 44–57, 2004.
- [13] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with Assertions. In *RV'01*, volume 55.2 of *ENTCS*, 2001.
- [14] 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 *OOPSLA'06*, 2006.
- [15] E. Bodden. J-lo, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.
- [16] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP'07*, volume 4609 of *LNCS*, pages 525–549, 2007.
- [17] E. Bodden, P. Lam, and L. Hendren. Flow-sensitive static optimizations for runtime monitors. Technical Report abc-2007-3, Oxford University, 2007.
- [18] F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *ICFEM'04*, volume 3308 of *LNCS*, pages 357 – 373, 2004.
- [19] F. Chen, M. D'Amorim, and G. Roşu. Checking and correcting behaviors of Java programs at runtime with JavaMOP. In *RV'05*, volume 144(4) of *ENTCS*, 2005.
- [20] F. Chen and G. Roşu. JavaMOP. <http://fsl.cs.uiuc.edu/javamop>.
- [21] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *RV'03*, volume 89(2) of *ENTCS*, 2003.
- [22] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS'05*, volume 3440 of *LNCS*, pages 546–550, 2005.
- [23] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *International Workshop on Dynamic analysis (WODA'05)*, 2005.
- [24] D. Drusinsky. Temporal Rover. <http://www.time-rover.com>.
- [25] Eclipse. <http://eclipse.org>.
- [26] Eiffel Language. <http://www.eiffel.com/>.
- [27] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA'05*, 2005.
- [28] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *RV'01*, volume 55.2 of *ENTCS*, 2001.
- [29] K. Havelund and G. Roşu. *Runtime Verification (RV'01, RV'02, RV'04)*. Elsevier, 2001, 2002, 2004. ENTCS 55, 70, 113.
- [30] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall Intl., New York, 1985.
- [31] JBoss. <http://www.jboss.org>.
- [32] jHotdraw. <http://www.jhotdraw.org>.
- [33] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, volume 1241 of *LNCS*, pages 220–242, 1997.
- [34] G. Kiczales, J. D. Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [35] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Runtime Assurance Tool for Java. In *RV'01*, volume 55.2 of *ENTCS*, 2001.
- [36] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA'00*, 2000.
- [37] Lucene. <http://lucene.apache.org>.
- [38] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA'05*, 2005.
- [39] B. Meyer. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, New Jersey, 2000.
- [40] M. Rinard. Acceptability-oriented computing. In *Onward! Track, OOPSLA'03*, 2003.
- [41] O. Sokolsky and M. Viswanathan. *Runtime Verification (RV'03)*. Elsevier, 2003. ENTCS 89.
- [42] Xalan. <http://xml.apache.org/xalan-j/>.