Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP [1]

Feng Chen, Marcelo d'Amorim, Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign, USA
{fengchen, damorim, grosu}@uiuc.edu

Abstract

Monitoring-oriented programming (MOP) is a software development and analysis technique in which monitoring plays a fundamental role. MOP users can add their favorite or domain-specific requirements specification formalisms into the framework by means of *logic plug-ins*, which essentially comprise monitor synthesis algorithms for properties expressed as formulae. The properties are specified together with declarations stating *where* and *how* to automatically integrate the corresponding monitor into the system, as well as *what* to do if the property is violated or validated. In this paper we present JAVA-MOP, an MOP environment for developing robust JAVA applications. Based upon a carefully designed specification schema and upon several logic plug-ins, JAVA-MOP allows users to specify and monitor properties which can refer not only to the current program state, but also to the entire execution trace of a program, including past and future behaviors.

1 Introduction

It is relatively broadly accepted today that proper usage of assertions and runtime checking can significantly increase the quality and reduce the cost of software development. Most of the systems supporting assertions and online checking, however, tend to focus on contracts between interfaces or on simple checkpoints, providing limited or no support for specifying and checking complex requirements referring, for example, to past or to future events. Moreover, most of the current approaches support, encourage and promote a *unique* underlying requirements specification formalism, assumed by its designers to be sufficiently powerful to express properties of interest. Nevertheless, it is often the case that such "hardwired" property specification formalisms cannot express naturally intuitive properties of certain applications, especially in domain-specific contexts.

Monitoring-oriented programming (MOP) was introduced in [7,9] as a formal framework for software development and analysis, aiming at reducing the gap between formal specification and implementation of software systems. In MOP, monitoring is supported and encouraged as a fundamental principle. Monitors are automatically synthesized from formal specifications and integrated at appropriate

[1] Partly supported by NSF/NASA grant CCR-0234524 and NSF CAREER grant CCF-0448501.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

places in the program, according to user-configurable attributes. Violations and/or validations of specifications can trigger user-defined code, for instance error recovery, outputting/sending of messages, or throwing of exceptions, at various points in the program. MOP allows users to insert their favorite or domain-specific requirements specification formalisms via *logic plug-ins*, which can be essentially regarded as monitor synthesizers for properties expressed as formulae.

Our previous efforts in [7,9] focused on presenting the basic, fundamental principles of MOP in a programming-language-independent manner. In this paper we take a more pragmatic attitude and focus on JAVA-MOP, an instance of MOP whose aim is to allow users to specify and verify at runtime safety properties of JAVA programs. Our current implementation of JAVA-MOP supports most, but not all, of the desired features of MOP. Future versions of the system will gradually incorporate the remaining features by need, driven by practical experiments. JAVA-MOP builds upon our experience with another runtime verification and monitoring system, NASA's Java PathExplorer [15], whose practicality has been testified in the context of NASA applications. JAVA-MOP is the basis for our experiments on code instrumentation, on monitor generation and integration, as well as on the use of MOP in practical applications.

Efforts have been recently invested in making JAVA-MOP a practical tool for monitoring JAVA programs against requirements expressed in various formalisms, with full support for executing user-provided (recovery) code when these requirements are violated or validated. In particular, following the fundamental idea of keeping the three components of monitoring (observation, checking and recovery) decoupled, we have devised a general meta-specification language for adding requirements specifications to JAVA applications without modifying manually the native code. By analyzing such a user-provided meta-specification, JAVA-MOP can *automatically* generate monitors together with corresponding recovery actions, and can then integrate them into the original programs. We have used JAVA-MOP on a non-trivial case study, namely Sun's JAVA CARD API 2.1. Even though the techniques discussed in this paper are specialized to JAVA, we believe that they are general enough to apply to other object-oriented programming languages.

This paper is organized as follows. Section 2 introduces the reader to JAVA-MOP, by means of a simple example showing how one can use MOP to detect and recover from concurrency errors in a JAVA HTTP client application. Section 3 discusses related work. Section 4 presents the JAVA-MOP tool. Section 5 approaches some implementation details, and, finally, Section 6 concludes the paper.

2 MOP in Java: A Simple Example

In this section we show a simple example where MOP helps to increase the robustness of a software system through online detection of requirements violations caused by "unexpected" thread interleavings; moreover, once a violation is detected, user-provided recovery code is executed, thus reflecting MOP's runtime detect-and-recover capability. Figure 1 shows a JAVA code fragment of an HTTP client taken from [6], which tries to request resources from the server and uses a shared queue to keep track of waiting clients. The client first requests access to the

server. If not granted, it adds itself into a waiting queue (`/*1*/`) and then suspends itself (`/*2*/`), waiting for another client to resume it. If granted, it does its work with the server and then it resumes a waiting client, if there is any waiting (`/*3*/`). The client continuously requests access to the server in a loop. To avoid dataraces, the access to the waiting queue needs to be synchronized.

```
  public class HttpClient extends Thread {
   private static Vector suspendedClients = new Vector();
   ... irrelevant code ...
   public void run() {
    while (true) {
     ... request server access ...
     if (!accessGranted) {
/*1*/   synchronized (suspendedClients) {
          suspendedClients.add(this);
        }
/*2*/   suspend();
     } else {
        ... work with server ...
        synchronized (suspendedClients) {
        if (!suspendedClients.isEmpty()) {
/*3*/     ((HttpClient)suspendedClients.remove(0)).resume();
  }}}}}}
```

Fig. 1. Java code fragment of an Http client.

There are (at least) two subtle concurrency errors in this code. The first is as follows. Suppose that a client's access is denied for some reason and that, right before it adds itself to the `suspendedClients` queue (at /*1*/), the thread scheduler delays it so long that all other clients terminate their job. Our client then continues and adds itself to the waiting queue, but, unfortunately, there is no other client working with the server to ever resume it. So that client will suspend until another client hopefully comes and is granted access, to eventually resume the starved client.

The other concurrency error is as follows. Suppose that a client is denied access, puts itself into the waiting queue, and then right after releasing the lock but before suspending itself (at `/*2*/`) it is delayed long enough to allow another client to remove it from the waiting queue and resume it (`/*3*/`) – resume has no effect if the thread is not suspended. Then the thread regains control and continues to suspend itself. Now there is no information about its suspension in the waiting queue, so no other client will ever resume it: this client is suspended forever.

One could try to fix these concurrency errors by enforcing additional atomicity, such as by synchronizing the check for server access, the waiting queue operation, and/or the suspend action. However, besides the usual efficiency penalties, such additional synchronizations are deadlock prone; in particular, since `suspend` does not release the locks that the corresponding thread holds, its occurrence in a synchronized section is almost equivalent to a deadlock. A better solution could be to reorganize the code to use `wait()` and `notify()` instead.

Both errors are difficult to detect during testing, and even harder to locate their causes. What MOP provides here is a mechanism to detect and recover from these errors *at runtime*. Without even having or understanding a particular implementation of an HTTP client, one can state that a basic natural requirement for using `suspend` and `resume` is that, for any thread, calls to `suspend` and `resume` on

the thread alternate and start with a `suspend`. This can be specified as a regular pattern, namely `(suspend resume)*`, disregarding any other irrelevant events. JAVA-MOP can automatically generate and integrate a runtime checker for this requirement, thus detecting the second bug above, in case it occurs, since it is caused by a mis-ordering of calls: `resume` is called before a corresponding `suspend`.

```
Logic = ERE;
var int flag = -1;
Event suspend: called(void suspend()) {flag = 1;};
Event  resume: called(void  resume()) {flag = 2;};
Formula: (suspend resume)*
Violation Handler:
  if (flag == 2) {
    System.out.println("resume() called before suspend() in HttpClient!
                        Adding client back to queue ...");
    synchronized(suspendedClients){
      suspendedClients.add($this);
    }
  } else {
    System.out.println("suspend() called again before
                        resume() is called in HttpClient!");
  }
```

Fig. 2. JAVA-MOP specification with recovery

Figure 2 shows this requirement specified in JAVA-MOP. Here the underlying formalism is that of extended regular expressions (ERE), and that is stated first using the keyword `Logic`. This way, JAVA-MOP knows which logic plug-in to use for generating the monitoring code. Then the events to monitor are declared, which form the atoms over which the requirements are then formalized as a regular expression. Events declared using `called` are examined within the context of the callee and can also bind the arguments of the called method for further use in warning messages or recovery (not the case here). One can declare local variables, such as `flag`, for use in the generated monitor and can associate actions (any Java code) to events. Here the actions are very simple, they only set the `flag` variable to recall the method-call that occurred last. The violation handler allows one to carry out any task when the requirements are violated; error-reporting and/or exception raising are just simple special cases. Here, for example, the monitor recovers from the error by adding the wrongly resumed thread back to the waiting queue. Thus, JAVA-MOP can not only help to locate errors, but also recover online.

The first concurrency error above, which is likely to self-recover (when another client is granted access), is, however, not fixed by the above JAVA-MOP specification. This error yields a violation of a liveness property, namely that "any suspended client will be eventually resumed". Unfortunately, such unbounded liveness properties are *not monitorable* [27,12]. Nevertheless, one can use metric temporal logic (MTL) [26] (see [35] for a monitor synthesis algorithm) to state bounded liveness properties of the form "any suspended client will be eventually resumed in t seconds". The generated monitor would check if a thread is resumed *t* seconds after it suspends, and the violation handler would resume the starved thread.

Since the properties to check should follow the informal requirements of a system, they are expected to be independent from any implementation details, so they can be provided by the system designers or analyzers even before the implementa-

tion process starts. Programmers then only need to provide the violation handlers, which can contain any recovery code suitable for the particular implementation.

In this example we have only discussed how MOP can detect violations of trace-related properties expressed using regular expressions. It is worth noticing that MOP is *not* limited to regular expressions or to related formalisms. Any specification language supported by a corresponding logic plug-in can be employed.

3 Related Work

There are many existing software development approaches related to MOP that were a major source of inspiration and documentation to us. What makes MOP different is its generality and modularity with respect to the logics underlying specification requirements, which allow it to include other runtime checking approaches as special cases. In this section we mention some approaches more closely related to MOP, and intuitively discuss their relationships to MOP.

Assertion-based runtime checking. The use of runtime assertions in software development is not new. [29] presents an annotation pre-processor for C, named APP, and discusses a classification of assertions. Design by Contract (DBC) [22] was proposed as a software design methodology, well supported in Eiffel [2], in which specifications given as assertions/invariants in programs are compiled into runtime checks. There are DBC extensions proposed for several languages. JASS [5], JCONTRACTOR [3], and JML [21] are DBC approaches for JAVA. MONGEN [14] is another DBC monitoring approach for Java, aiming at checking constraints in design patterns specified as formal contracts [33]. However, MONGEN assumes the monitors are manually coded instead of being automatically generated.

These techniques and tools have shown their strength in practice. However, they can only reason about the *current* program state – they cannot support trace requirements. Trace properties concern the sequence of states rather than only the current state. In particular, safety and liveness are critical requirements in concurrent systems and can only be specified in terms of program traces (see examples in Section 2 and 5.4). Efforts have been made to support advanced properties in DBC-style approaches. JML, for instance, provides *ghost* and *model* variables that can be used to store information from past states, but that essentially requires the user to manually translate the formal specification into programs. This makes the final JML specification hard to understand and error-prone. JASS 2.X provides support for trace assertions in the style of CSP [18], but can only encode strings over the method calls of a program. Moreover, all the above use *fixed* and *different* specification formalisms. However, all these different formalisms fall under the uniform format of logic plug-ins in MOP. For instance, we have already implemented logic plug-ins for significant subsets of JASS and JML (Section 5.3).

Runtime verification (RV) [16,32] aims at providing more rigor in testing. In RV, monitors are automatically synthesized from formal specifications. These monitors can then be deployed *off-line* for debugging, i.e., they analyze the execution trace "post-mortem" by potentially random access to states, or *on-line* for dynamically checking that safety properties are not being violated during system execution. JAVA-MAC [20], JPAX [15], JMPAX [31], and EAGLE [4] are such RV systems.

JAVA-MAC uses a special interval temporal logic as the specification language, while JPAX and JMPAX currently support only linear temporal logic. EAGLE is a finite-trace general logic and tool for runtime verification. TEMPORAL ROVER [13] is a commercial RV system based on metric temporal logic (MTL) [26] specifications.

These systems, unfortunately, also have their specification formalisms fixed. While a fixed formalism to express requirements may seem appealing for a tool designer, experience tells us that there is no "silver bullet" logic whose formulae can naturally express any property of interest in any application. We believe that all the RV systems that we are aware of would become special instances of MOP, provided that appropriate logic plug-ins are defined. In fact, the general ideas and the modular approach underlying MOP are a result of our experience in the area of runtime verification, and were motivated by our strong interest in *unifying* the apparently different RV approaches.

Aspect-oriented programming (AOP) [19] is a software development technique aiming at separation of concerns. An aspect is a module that characterizes the behavior of cross-cutting concerns. Aspects are comprised of three basic elements: join point, point cut, and advice. The first identifies relevant points in the control flow of a program. A point cut represents several join points concisely in a single abstraction, and an advice relates a point cut to an expression that is evaluated when control flow hits the join point. AOP provides a means to define behavior that cross-cuts different abstractions of a program, avoiding scattering code that is related to a single concept throughout the code. One can understand AOP as a language transformation technique that mechanically and transparently instruments the code with advice expressions.

Although MOP's most challenging part is the synthesis of monitors and instrumentation code from high-level specifications, the importance of a powerful mechanism to facilitate the integration of monitors into the implementation cannot be overstated. AOP provides such a mechanism. Our current implementation of JAVA-MOP uses ASPECTJ [1] as an instrumentation infrastructure: synthesized monitoring code is wrapped as advices and then ASPECTJ is invoked to finish the integration work. From an AOP perspective, one can understand MOP as a *synthesizer of* AOP *advices*. However, it is important to note that MOP and AOP are intended to solve different problems. MOP is tuned and optimized to *merge* specification and implementation via monitoring, while AOP aims at *separation* of concerns. Even though ASPECTJ provides JAVA-MOP with an elegant and rapid mechanism to integrate monitors into an implementation, it does not provide everything a powerful MOP environment needs: in particular, ASPECTJ does not provide support for some MOP features such as atomicity of actions associated to events, or property checks at every state change of a particular object.

4 JAVA-MOP

JAVA-MOP is an MOP development tool for Java. The major purpose of JAVA-MOP is to provide an infrastructure for combining formal specification and implementation by automatic generation of monitoring code and instrumentation of monitored programs for JAVA. To accommodate the underlying pluggable logic frame-

work, JAVA-MOP provides a general and extensible specification schema, allowing users to specify properties using different formalisms and to optionally state how to steer the behavior of the system when requirements are violated or validated. This schema is devised to fit JAVA, but is general enough to easily support other object-oriented languages. This section focuses on the specification schema of JAVA-MOP, leaving the implementation details to the next section.

4.1 Standalone Specifications v.s. Annotations

We encourage users to provide JAVA-MOP specifications in separate files. However, for users' convenience, we also allow specifications to be added as code annotations. This makes MOP look similar in spirit to other DBC-like tools, e.g., JASS or JML. When annotations are used, the JAVA-MOP front end generates a separate specification file from the annotated source file.

The current tool supports only properties within the scope of a class. Therefore, each JAVA-MOP specification file corresponds to a JAVA class, containing all the properties concerning that class. Each property is formally given as a JAVA-MOP specification that will be further turned into a monitor. Figure 3 shows the format of a JAVA-MOP specification. Note that JAVA comments are allowed.

4.2 Specification Schema

```
/************** Heading starts ****************/
[attribute]* <Type> <Name> Logic=<Logic Name> {
  /************** Body starts ********************/
  ... Specification Body ...
  /************** Handler starts *****************/
  [Violation Handler: ...handling the violation...]
  [Validation Hander: ...handling the validated...]
}
```

Fig. 3. Syntax of the JAVA-MOP Specification

The design of the JAVA-MOP specifications is mainly driven by the following factors: uniformity in the use of various logics, ability to control monitor behaviors, and compatibility with existing tools such as those based on DBC. A formal specification consists of three parts: the heading, the body and the handlers.

The ***heading*** is composed of optional attributes, type, name of the specification, as well as the name of the underlying logic (the unique name identifying the corresponding logic plug-in). We next discuss each of these in more depth.

Attributes are used to configure the monitor with different installation capabilities. They are orthogonal to the actual monitor generation. One important attribute is `static`, which states that the specification is related to the class instead of the object. For a static monitor, only one instance is generated at runtime and is shared by all the objects of the associated class. By default, monitors are non-static, meaning that every object will be monitored individually. The `asyn` attribute requires the monitor to run asynchronously. When omitted, the monitor runs in synchronized mode, forcing the system to wait until the monitor finishes its work.

The ***type*** defines points in the execution where properties are checked. Four types are available: `class-inv`, `interface-constr`, `method`, and `check-point`. The type `class-inv` states that the property is a class invariant and should be checked whenever the referred fields are updated or the referred methods

are called. `interface-constr` denotes a constraint on the interface. It should be checked at every observable state change, specifically on boundaries of public method calls. It is similar to a class invariant in JML [21]. The `method` specification is used to specify pre, post, and exceptional conditions for a method. `checkpoint` specifications are placed inside the code and checked whenever they are hit during the execution. If the `checkpoint` specification is written in a separate file, the programmer may place a reference to the name of the specification, `//@ <specification name>`, at the appropriate positions in the source code.

The ***logic name*** used in the specification, e.g. JML or ERE, is needed in order for JAVA-MOP to generate the monitor using the appropriate logic plug-in.

The ***body*** of the specification formally specifies the desired property. Its syntax varies with the underlying logic. For JML and JASS specifications, we adopt their original syntax except for the format of comments. So one can translate JML and JASS specifications into JAVA-MOP simply by changing their headings and providing violation handlers. Properties written in logics that express requirements over traces of the program, such as ERE and LTL, need a different structure of the specification body, like the one discussed in Section 4.3.

The ***handlers*** are provided by the user at the end of the specification to handle the violation and validation of the property. It is worth noting that violation and validation of a formula are *not* complimentary to each other. For example, a property stating "event A eventually leads to event B" would never be violated or validated. To provide better support for error handling, JAVA-MOP pre-defines some variables which can be used in handlers. These give the handler the ability to retrieve environment information, such as the current object reference (`$this`), arguments and the return value of a method call, as well as other information to locate the violation, such as the name of the monitored specification, and so on.

4.3 Specifying Trace Properties

Trace logics, such as ERE and LTL, give users the ability to specify safety properties concerning the entire execution trace. Specifying such trace properties requires a different structure of the specification body from that used in contract-based formalisms such as JML and JASS. Based on experience with runtime verification of temporal properties, we devised a typical structure for the body of the trace specification, which consists of two parts, as Figure 4 shows. The first specifies how to extract the abstract trace of the program, by declaring predicates and events building the trace, along with some assistant variables. The second is a formula specifying the property, whose syntax is specific to the underlying logic. It is worth noting that, although we believe this structure is suitable for many trace logics, the user can devise her own syntax for the logic that she adds to the JAVA-MOP tool.

There are two important aspects regarding the abstraction of the execution trace. One is to define the *observation points* and the other is to extract the necessary *state information*, i.e., the *abstract state*. Most types of MOP specifications have their observation points fixed by design. But for the class invariant, the observation points are implicitly determined by the specification, e.g., by the declaration of predicates and events in the above example. Specifically, this specification should

```
For CruiseController

class-inv CruiseControlBehavior Logic=FTLTL {
  /******** specification body (declarations) *******/
  var double x = 0;
  event ControlOn: called(void On()) {x = $this.speed;}
  event ControlOff: called(void Off());
  predicate isNormal: ($this.speed < x+5) && ($this.speed > x-5);
  /******** specification body (formulae) **********/
  /* After the cruise control is turned on, the speed should be    */
  /* kept within the cruise speed +/- 5 until  the control is off. */
  formula : ControlOn -> (isNormal U ControlOff);
  /*********** Handlers begin *********************/
  Violation Handler: {
    // if violated, try to restore the speed.
    if ($this.speed > x) $this.brake() else $this.accelerate();
  }
}
```

Fig. 4. A cruise control specification using future-time linear temporal logic(FTLTL).

be checked at the end of calls to methods `on()` and `off()` in the `CruiseController` class (stated by the `For` keyword at the top of Figure 4) as indicated by events, as well as at every update of the `speed` field since it is referred from the `isNormal` predicate.

The declaration of events follows the format: `event <name> : <event type> [&& <boolean expression>] [action]`. The following event types are supported: `update(<field>)`, `called(<method>)`, `begin(<method>)`, `end(<method>)`, and `exception(<method>)`. The semantics of event types is as follows. For `update`, an event is sent right after the corresponding field assignment; for `called`, an event is generated right before the method is called in the caller's context (this may be necessary because sometimes the source code of the method is unavailable, e.g., methods of the superclass that come from a library); for `begin`, an event is generated right before the beginning of the method execution; for `end`, when the execution of a method ends but has not returned; for `exception`, when the method throws an exception and exits and before the exception is caught. One can put additional constraints on the event. This is realized by attaching a boolean expression that must be true when the event is triggered.

JAVA-MOP also allows users to associate actions, declared within curly brackets, to monitored events as Figures 2 and 4 show. This strengthens the expressiveness and effectiveness of the specification language. For example, like in Figure 2, we may associate counters with events together with a regular expression and thus specify properties like "the trace contains as many A's as B's", which are beyond the expressive power of regular languages. In general, developers can associate any action to events. This allows one to create orthogonal data-structures that can be used to smoothly "wrap" an application and "observe" each of its execution steps. In the example in Figure 4, `x` is updated to the current speed when the cruise control is turned on. This way we maintain the *monitor state*, a necessary feature in the support of parametric events.

Predicates are declared with the keyword `predicate` and follow the format: `predicate <name> : <boolean expression>`. As discussed, for class invariants, predicates also indicate the observation points, namely that class invari-

ants should be checked every time when any of the variables used in the defining predicates are updated. Currently, due to limitations of the AOP framework that we use, only fields of primitive types are allowed in the class invariant in JAVA-MOP.

Predicates and events are then used in the context of a formula as atomic propositions. When the observation point of interest is encountered, the corresponding monitor will evaluate these propositions based on the program state and the event that it received, and then use their values to evaluate the formula.

5 Implementation

JAVA-MOP provides both GUI and command-line interfaces for editing and processing specifications. The tool can be obtained from our website [23]. A web-based interface is also available for the interested user to experiment online before having to go through the installation process.

The JAVA-MOP tool incorporates two functionalities: code generation and monitor integration. Code generation is encoded within logic plug-ins, and ASPECTJ is used as an integration (instrumentation) mechanism. Specifically, the tool generates ASPECTJ aspects for specifications which are not `checkpoint`, and invokes the ASPECTJ compiler to instrument the original program. In this section, we briefly discuss the design and implementation of JAVA-MOP.

5.1 Architecture

To provide the extensiblity of MOP, we employ a client-server architecture style. The client includes the interface modules and the JAVA-MOP specification processor, while the server contains a message dispatcher and logic plug-ins for JAVA. The message dispatcher takes charge of the communication between the client and the server, dispatching requests to corresponding logic plug-ins. The communication can be either local or remote, based on the installation of the server. The advantage of this architecture is that one logic server can provide monitor generation services, which can require intensive computation and/or search through already processed formulae (for efficiency), to multiple clients. Besides, the client is implemented purely in JAVA and thus can run on different platforms, while some of the logic engines, namely those for linear temporal logics and ERE, are implemented in Maude [11], an efficient meta-logic development tool which runs best under Linux. This architecture provides a more portable tool, since the client and the server are allowed, but not enforced, to run on different platforms.

The client provides both a command-line and a graphical user interface. The command-line interface takes as input argument either a sequence of JAVA files and specification files, or a folder path that contains these. Then it processes all the JAVA file(s) found in the input path, generating files in which monitors are synthesized and integrated appropriately into the original source code. Currently the GUI can only handle annotated JAVA files. It is based on the ECLIPSE platform [1]. We also implemented a Web-based interface [23] through which one can try JAVA-MOP online without having to install it locally.

JAVA-MOP currently uses ASPECTJ for code instrumentation. For `checkpoint` specifications, monitoring code is inserted where the annotations were defined. ASPECTJ aspects are produced for all other kinds of specifications. However, note that

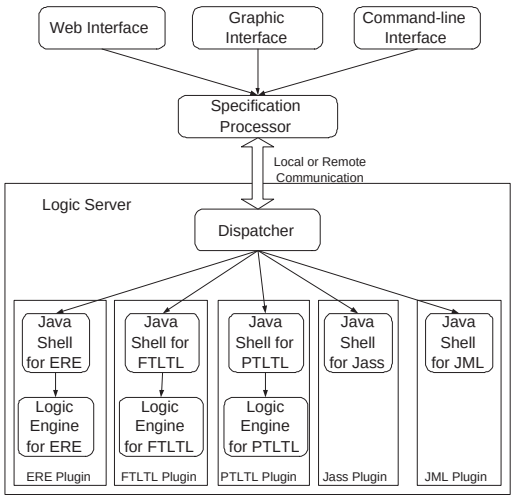

Fig. 5. The Architecture of JAVA-MOP.

ASPECTJ performs *static* code instrumentation, while monitoring is *dynamic*. This may be inconvenient in some applications. For example, for a class invariant, one may need to monitor every update of a field on a *specific instance* of a class, instead of monitoring all the updates to the field in all the objects of that class.

5.2 Interfaces to Logic Plug-Ins

One important feature of the MOP framework is its extensibility, which allows the user to add new specification formalisms by providing logic plug-ins. In order to support this feature, the input and output to a logic plug-in should be in a standard format. In JAVA-MOP, the input to the logic plug-in is simply the body of the specification, while its output is composed of the following:

Monitored variables. Fields in the class, whose updates should be monitored.

Monitored events. Events to monitor along with associated actions, following the syntax "`<eventName> [event definition] <actions>`". The event definition can be one of `set(variable)`, `called(method signature)` or `execution(method signature)`.

Declarations. Variables to maintain relevant state information, needed for the next monitoring step. These variables will be declared as new fields in the class.

Initialization. A segment of program to prepare the monitor to start monitoring.

Monitoring body. The main part of the monitor, which is executed any time the observation point is reached.

Intermediate declaration. Temporary variables needed by the monitor during the verification process.

Success condition. Says when the monitored requirement has been fulfilled. When this becomes true, the user-provided validation handler will be executed.

Failure condition. This gives the condition that shows when the trace violates the requirements. When this condition becomes true, the user-provided "recovery" code (given as the violation handler) will be executed.

Figure 6 shows the output of the FTLTL logic plug-in used in Figure 4. The JAVA-MOP specification processor will further translate this output into ASPECTJ aspects.

```
//Monitored Variables
$this.speed;x;
//Monitored Events
ControlOn[ called(void On()) ]:{
event0=true; x = $this.speed;}
ControlOff[ called(void Off())]:{
event1=true;}
//Declaration
int $state = 1;
local boolean event0=false, event1=false;
//Intermediate Declaration
boolean isNormal= ($this.speed < x+5) && ($this.speed > x-5);
boolean ControlOn=#event0;
boolean ControlOff=#event1;
//Monitoring body
switch ($state) {
case 1:
 $state =  ControlOff ? -1 : ControlOn ? isNormal ? 2 : -2 : -1; break ;
case 2:
 $state =  ControlOff ? -1 : isNormal ? 2 : -2; break ;
}
//Success condition
$state == -1
//Failure condition
$state == -2
```

Fig. 6. Output of the FTLTL plug-in for the specification in Figure 4.

5.3 Supported Specification Languages

Two kinds of specification languages are currently supported in MOP: DBC-like runtime checking languages such as JML and JASS, and trace languages like ERE and LTL. We next introduce them informally. Interested readers can refer to our technical report [8] for more technical details, including corresponding algorithms.

JML and Jass. Both JML and JASS are Java specific, using Java syntax inside specifications. This makes translation from specification to checking code straightforward; separate logic engines are unnecessary in such cases. Therefore, the logic plug-ins for JML and JASS consist only of language shells. JASS has been defined in a plug-in supporting state assertions. Most features of JASS, except trace assertions in JASS 2.x, are supported. For trace properties, we prefer to use ERE and LTL as specification languages. JML provides a comprehensive modeling language with some features that are difficult, sometimes almost impossible, to monitor, for example, the `assignable` clause [21]. We therefore focused on defining those features supported by the JML runtime checker in [10], including method specifications, type invariants, and historic constrains. We do not support abstract specifications, i.e., *ghost* variables and *model* fields, but note that declaring and using variables inside specifications is supported in a more general fashion in MOP.

ERE. Regular expressions provide an elegant and powerful specification language for monitoring requirements, because an execution trace of a terminating program is in fact a string of states. The advantage of regular expressions over many other logics is that they are a standard form of notation to which many people have already been exposed. Extended regular expressions (ERE) add complementation, or negation, to regular expressions, allowing one to specify patterns that must *not* occur during an execution. Complementation gives one the power to express more compactly patterns on traces. However, complementation leads to a

non-elementary exponential explosion in the number of states of the corresponding automaton if naive ERE monitoring algorithms are used. Preliminary efforts in [28,30] show how to generate efficient monitoring algorithms for ERE. A logic engine for ERE and its corresponding JAVA plug-in have been implemented.

LTL. Temporal logics and its variations prove to be favorite formalisms for formal specification and verification of systems [24]. Safety properties can be naturally expressed using temporal logics, so these logics can also be very useful in MOP. Based on work in [17] and [27], we implemented logic engines and corresponding language shells for JAVA to support variants of temporal logics.

5.4 Case Study

We evaluated the effectiveness of JAVA-MOP on SUN's standard JAVA CARD API 2.1 informal specification and the reference implementation [34]. Our case study [2] was initially motivated by an already existing JML specification [25]. The analysis carried out in this case study illustrates the strength of combining specification formalisms, such as JML, ERE and LTL in this case. The resulting specification is more comprehensive and more concise when appropriate formalisms are used for different properties. Another interesting observation in this case study is that, while monitoring contracts of classes is quite heavy and may greatly impact the performance in many cases, monitoring temporal properties, e.g. safety properties about orders of method calls, is usually relatively little costly because it requires few observation points as well as simple processing actions, most of which just simple state transitions. Moreover, violations of temporal properties are very likely to be corrected at runtime by proper usage of handlers. The complete case study report can be found on our website [23]. Here we only present some conclusions.

JAVA CARD API 2.1 consists of four packages, namely, `java.lang` with 12 classes, `javacard.security` with 17 classes and interfaces, `javacard.framework` with 18 classes and interfaces, and an optional `javacardx.crypto` package. The corresponding specification from [25] presents an informal description of requirements for the implementation. As in [25], our study focuses on the APIs constraints, putting aside the functional specifications and properties related to lower level details. In addition to straightforward pre-conditions, post-conditions, and exceptional conditions, our review reports around 40 critical properties, most of which are history-related constraints on the method calls (30 out of 40). Therefore, allowing the use of logics such as ERE or LTL provides a more concise and dependable way to formally specify and check time-related properties at runtime, significantly improving the expressiveness of the specification.

6 Conclusion and Future Work

A software development tool supporting monitoring oriented programming (MOP) in the context of JAVA has been presented, called JAVA-MOP. Following the general philosophy of MOP, our tool supports several requirement specification formalisms and can easily be extended with new ones, provided that corresponding logic plugins are supplied. Several examples were discussed, showing the practical feasibility of the approach in spite of its generality. Interesting future work includes support

[2] We warmly thank Sophie Quinton of her help with this case study.

for specifying global (cross class) properties, which is needed for many applications including the JAVA CARD case study. Another interesting avenue for further investigation is to use static analysis as a means to reduce the runtime overhead: if a specification, or part of it, can be proved statically, then one does not need to generate the corresponding monitor, or can generate a more efficient one.

References

[1] Eclipse and aspectj project. http://www.eclipse.org.

[2] Eiffel language. http://www.eiffel.com/.

[3] P. Abercrombie and M. Karaorman. jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java. In *Proceedings of RV'02*, volume 70 of *ENTCS*. Elsevier Science, 2002.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of VMCAI'04*, volume 2937 of *LNCS*. Springer, 2004.

[5] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with Assertions. In *Proceedings of RV'03*, volume 55 of *ENTCS*. Elsevier Science, 2001.

[6] Concurrency bugs. http://qp.research.ibm.com/QuickPlace/concurrency_testing.

[7] F. Chen, M. d'Amorim, and G. Roşu. A Formal Monitoring-Based Framework for Software Development and Analysis. In *Proceedings of ICFEM'04*, volume 3308 of *LNCS*, pages 357–372. Springer, 2004.

[8] F. Chen, M. d'Amorim, and G. Roşu. Monitoring Oriented Programming. Technical Report UIUCDCS-R-2004-2420, Univ. of Illinois Urbana-Champaign, March 2004.

[9] F. Chen and G. Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In *Proceedings of RV'03*, volume 89 of *ENTCS*. Elsevier Science, 2003.

[10] Y. Cheon and G. T. Leavens. A Runtime Assertion Checker for the Java Modeling Language. In *Proceedings of SERP'02*, pages 322–328. CSREA Press, 2002.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual. June 2003. http://maude.cs.uiuc.edu/download.

[12] M. d'Amorim and G. Roşu. Efficient Monitoring of ω-Languages. In *Computer Aided Verification (CAV)*, 2005.

[13] D. Drusinsky. Temporal Rover. http://www.time-rover.com.

[14] J. Hallstrom, N. Soundarajan, and B. Tyler. Monitoring Design Pattern Contracts. Technical Report 04-09, IOWA State University, November 2004.

[15] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of RV'01*, volume 55 of *ENTCS*. Elsevier Science, 2001.

[16] K. Havelund and G. Roşu. *Workshops on Runtime Verification (RV'01, RV'02, RV'04)*. 2001, 2002, 2004.

[17] K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In *Proceedings of TACAS'02*, volume 2280 of *LNCS*. Springer, 2002.

[18] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[19] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[20] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of RV'01*, volume 55 of *ENTCS*. Elsevier Science, 2001.

[21] 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 2000 Companion*, 2000.

[22] B. Meyer. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, 2000.

[23] MOP Website. http://fsl.cs.uiuc.edu/mop.

[24] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of FOCS'77*, pages 46–57. IEEE, 1977.

[25] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In *Proceedings of CARDIS'00*. Kluwer Academic Publishers, 2000.

[26] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *RealTime Systems*, 2(4):255–299, 1990.

[27] G. Roşu and K. Havelund. Rewriting-Based Techniques for Runtime Verification. *Automated Software Engineering*, 12(2):151–197, 2005.

[28] G. Roşu and M. Viswanathan. Testing Extended Regular Language Membership Incrementally by Rewriting. In *Proceedings of RTA'03*, volume 2706 of *LNCS*, pages 499–514. Springer-Verlag, 2003.

[29] D. S. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[30] K. Sen and G. Roşu. Generating Optimal Monitors for Extended Regular Expressions. In *Proceedings of RV'03*, volume 89 of *ENTCS*. Elsevier Science, 2003.

[31] K. Sen, G. Roşu, and G. Agha. Online Efficient Predictive Safety Analysis of Multithreaded Programs. In *TACAS'04*, volume 2988 of *LNCS*. Springer, 2004.

[32] O. Sokolsky and M. Viswanathan. *Runtime Verification 2003*, volume 89 of *ENTCS*. Elsevier Science, 2003. Proceedings of CAV'03 satellite workshop.

[33] N. Soundarajan and J. O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *Proceedings of ICSE '04*, pages 666–675. IEEE Computer Society, 2004.

[34] Sun. Java Card 2.1 API. http://java.sun.com /products /javacard/.

[35] P. Thati and G. Roşu. Monitoring Algorithms for Metric Temporal Logic. In *Proceedings of RV'04*, volume 113 of *ENTCS*. Elsevier Science, 2004.