ICFEM'04. LNCS 3308, pp 357 - 373. 2004

A Formal Monitoring-based Framework for Software Development and Analysis

Feng Chen, Marcelo D'Amorim, and Grigore Roşu

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

Abstract. A formal framework for software development and analysis is presented, which aims at reducing the gap between formal specification and implementation by integrating the two and allowing them *together* to form a system. It is called *monitoring-oriented programming* (MOP), since runtime monitoring is supported and encouraged as a fundamental principle. Monitors are automatically synthesized from formal specifications and integrated at appropriate places in the program, according to user-configurable attributes. Violations and/or validations of specifications can trigger user-defined code at any points in the program, in particular recovery code, outputting/sending messages, or raising exceptions. The major novelty of MOP is its generality w.r.t. logical formalisms: it allows users to insert their favorite or domain-specific specification formalisms via *logic plug-in* modules. A WWW repository has been created, allowing MOP users to download and upload logic plug-ins. An experimental prototype tool, called JAVA-MOP, is also discussed, which currently supports most but not all of the desired MOP features.

1 Introduction

We present a tool-supported software development and analysis framework, aiming at increasing the quality of software through monitoring of formal specifications against running programs. Based on the belief that specification and implementation should *together* form a system, the presented framework technically allows and methodologically encourages designs in which the two *interact* with each other. For this reason, we call it *monitoring-oriented programming* (MOP). In MOP, runtime violations and validations of specifications result in adding functionality to a system by executing user-defined code at user-defined places in the program; executing recovery code, outputting or sending messages, throwing exceptions, etc., are just special cases. Monitors are automatically synthesized from specifications and integrated appropriately within the system.

Practice has shown that there is no "silver bullet" logic to formally express any software requirements. One of the major design decisions of MOP was to keep it independent from any particular specification formalism. Instead, a specification formalism can be modularly added to the MOP framework, provided that it is coded as a *logic plug-in*, that is, a module whose interface respects some rigorously defined and standardized conventions explained in the paper. A logic plug-in essentially incorporates a *monitor synthesis* algorithm, taking formal specifications to corresponding concrete monitoring code that analyzes program execution traces. No other restrictions are imposed on the MOP-admissible specification formalisms, such as to be "executable" or "declarative"; these can range from highly abstract to very detailed. Once a specification formalism is defined as a logic plug-in, any MOP user can include it into her MOP framework. To facilitate the reuse of logic plug-ins, we created a WWW repository where users can download and upload their favorite, or application-specific, logic plug-ins.

Besides monitor synthesis, another major aspect of MOP is *monitor integration.* This is technically very challenging, because some specifications may need to be checked at various places in a program, some of them hard to detect statically. Monitor integration is highly dependent upon the target programming language(s). We have implemented a prototype MOP framework for Java software development and analysis, called Java-MOP, which will be also discussed.

In short, one can understand MOP from at least three perspectives:

(1) As a discipline allowing one to *improve reliability of a system by monitoring* its requirements against its implementation at runtime. By generating and integrating the monitors automatically rather than manually, MOP provides several important advantages, including reduced risk of producing wrong monitors due to misunderstanding of specifications, complete integration of monitors at places that may change as the application evolves, software products more amenable to formal verification and validation, and increased separation of concerns.

(2) As an *extension of programming languages with logics.* One can add logical statements anywhere in the program, referring to past or future states. These can be seen at least like any other boolean expressions, so they give the user a maximum of flexibility in using them: to terminate the program, guide its execution, recover from a bad state, add functionality, throw exceptions, etc. In some sense, MOP can be regarded as a framework potentially enabling fault-tolerant software development methodologies.

(3) As a *lightweight formal method.* While firmly based on logical formalisms and mathematical techniques, MOP aims at avoiding verifying an implementation against its specification before operation, by *not letting it go wrong* at runtime.

The idea of MOP was first introduced in [6] at the *Runtime Verification (RV) workshop*, focusing mainly on the logic independence aspect of it and having minimal tool support. The current paper reports significant advances of both ideological and technical nature, such as: (a) an AOP-based instrumentation package allowing more flexible monitors to be generated as aspects, based on which our current prototype can now support JASS annotations and will soon support JML; (b) several novel configuration attributes besides the in-line and out-line ones in [6], such as on-line, off-line, and synchronous; (c) a WWW repository of logic-plug-ins where users can test, download and upload logic plug-ins, containing already four of them: future time and past time temporal logics, extended regular expressions and JASS; and a new and much improved implementation of Java-MOP which can be configured to generate monitoring code using directly the logic plug-ins in the WWW repository. Since RV has a relatively limited audience, in this paper we do *not* assume the reader already familiar to MOP concepts and terminology. All the necessary notions are introduced as needed.

Related Work. There are many approaches related to MOP that were a great source of inspiration for us. What makes MOP different is its generality and modularity with respect to specification languages, allowing it to include other approaches as special cases. It is a major part of our efforts to capture these relationships explicitly, by providing specific logic plug-ins.

Design by Contract (DBC) [20] is a software design methodology particularly well supported in Eiffel [11]. DBC allows specifications to be associated with programs as assertions and invariants, which are compiled into runtime checks. There are DBC extensions proposed for several languages, such as JASS [4] and jContractor [1] for Java. Java 1.4 introduces simple assertions as part of the language, which are then compiled into runtime checks. DBC approaches fall under the uniform format of logic plug-ins, so MOP can naturally support DBC variants as special methodological cases. To strengthen this claim, we have implemented a logic plug-in for JASS. However, MOP also allows monitors to be synthesized *out-line* (See section 2.1 for more details), which is crucial in assuring high reliability of software and is not provided by any DBC approach.

Java Modeling Language (JML) [18] is a behavioral specification language for Java. It can be used to specify designs for Java classes and interfaces, and provides the basis for further analysis, e.g., runtime debugging [7] and static analysis [12]. JML assertions are presented as special annotations, using a fixed logical formalism, so JML can be easily incorporated into Java-MOP.

Runtime Verification (RV) is an area [14, 15, 26] dedicated to providing more rigor in testing. In RV, monitors are automatically synthesized from formal specifications, and can 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 properties are not being violated during execution. Java-MaC [17], JPaX [24], and Temporal Rover [9] are examples of such RV systems; however, they also have their requirements logics hardwired, and their generated monitors are synchronous. The MOP paradigm supports both in-line and out-line, both on-line and off-line, as well as both synchronous and asynchronous monitoring; these modes will be explained in Section 2. Although our current Java-MOP prototype does not support all these modes yet, as argued in Section 4, it is not hard to incorporate them. MOP also allows any logical specification formalism to be modularly added to the system. It is expected that all the RV systems that we are aware of will fall under the general MOP architecture, provided that appropriate logic plug-ins are defined.

Aspect Oriented Programming (AOP) [16] is a software development technique aiming at separation of concerns [27]. An *aspect* is a module that characterizes the behavior of cross-cutting concerns. 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 at different points of the program, thus aiming for maintenance and extensibility of the code. Similarly to AOP, MOP requires instrumentation to integrate monitors into the code; however, instrumentation is a small piece of MOP, namely a subpart of its *monitor integration* capability. MOP's most challenging part is indeed to *synthesize monitors* from user defined logical formulae. In AOP, the behavior of each aspect is left to be defined by the programmer. Actually, 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. Our current MOP prototype uses AspectJ [2] as an instrumentation infrastructure. However, AspectJ does not provide support for several important MOP features, such as asynchronous and/or out-line monitoring, as well as strong runtime support needed to check a property at each change of the state of an object, which is a must in the context of safety critical software.

2 Overview of MOP

The general idea underlying MOP is that specifications are inserted in programs via *annotations*. Actual monitoring code is automatically synthesized from these annotations and integrated at appropriate places in the program, according to user-defined configuration attributes of the monitor. Before we present the general principles of MOP, we first present a simple example. Figure 1 shows a Java-MOP annotation, where a traffic light controller is monitored against the safety property "yellow after green" expressed as a future time linear temporal logic (FTLTL) formula. Java-MOP takes such an annotated code and generates the corresponding monitor as plain Java code, which is shown in Figure 2. More details on the syntax of annotations are given in Section 2.1; [5] presents temporal logic and its corresponding monitor synthesis algorithm.

```
... (Java code A) ...
/*@ FTLTL
   Predicate red : tlc.state.getColor() == 1;
   Predicate green : tlc.state.getColor() == 2;
   Predicate yellow : tlc.state.getColor() == 3;
   // yellow after green
   Formula : [](green -> (! red U yellow));
   Violation handler : ... (Java "recovery" code) ...
@*/
... (Java code B) ...
```

Fig. 1. Java-MOP specification.

```
... (Java code A) ...
switch(FTLTL_1_state) {
case 1:
   FTLTL_1_state = (tlc.state.getColor() == 3) ? 1 :
     (tlc.state.getColor() == 2) ? (tlc.state.getColor() == 1) ? -2 : 2 : 1; break;
case 2:
   FTLTL_1_state = (tlc.state.getColor() == 3) ? 1 :
     (tlc.state.getColor() == 1) ? -2 : 2; break ;
}
if (FTLTL_1_state == -2) { ...(Violation Handler)... }
// Validation Handler is empty
... (Java code B) ...
```

Fig. 2. Monitor generated for the MOP specification in Figure 1.

One way to regard MOP is as an extension of programming languages with special statements which can refer to current, past or future states of the program. To effectively handle these special statements, a standardized annotation language is needed. Annotations are automatically converted into monitors, which can have several configurable attributes. The monitors may need meta-information about the program's state, such as the name of the current method or class, so a standardized communication interface between the program and annotations is also needed. Logics should be easy to incorporate as modules, called logic plug-ins, so another standardized interface is necessary for these.

2.1 Extending Programming Languages with Logical Annotations

Most works on extending languages with annotations mainly focus on detecting program errors. MOP is built on the belief that logical annotations can play a deeper, active role in the execution of a program. For example, Figure 3 shows how one can use MOP to guarantee authentication before access to protected resources, by simply adding an appropriate safety policy together with an action. The safety policy, expressed in negative form using extended regular expressions (ERE; see [5]), states that "if one accesses a resource, it must be the case that at some moment in the past the end of the authentication process has been seen". This policy is violated iff the ERE formula `[^end(a)]* start(m)`, stating that the end of the authentication has not occurred before the resource started to be managed, is validated. If that is the case then, instead of reporting an error, one simply enforces the authentication. Note that this specification has the attribute `class` (explained below), so the generated monitor will check the safety policy at each state change of the current object.

```
/*@ ERE {class}
   Event a : authenticate() ;
   Event m : manageResource() ;
   // If the resource is accessed without authentication
   Formula : (^ end(a))* start(m)
   // enforce the authentication
   Validation Handler: authenticate();
@*/
```

Fig. 3. A class monitor to assure authentication.

It is often the case that it is easier to express a safety policy by what should *not* happen. This is one of the reasons for which MOP accepts both a violation and a validation handler; another reason comes from the fact that there are situations where one may want to take different actions depending on whether a policy is violated or validated. Note that violation and validation are not complementary properties. Hence, by combining specifications expressed using appropriate underlying logical formalisms and code, and by synthesizing and integrating automatically corresponding monitoring code, critical properties of a program can be expressed and implemented rigorously and compactly. Moreover, the obtained program is more amenable to formal verification and validation, because significant parts of it are directly generated from its specification.

MOP Annotations. Specifications are introduced as comments in the host language, having the structure in Figure 4. They begin with an optional name,

```
/*@ [Annotation Name] <Logic Name> [{attributes}]
   ... Specification Body ...
   [Violation Handler: ... code handling the violation ...]
   [Validation Hander: ... code to trigger when validated ...
@*/
```

Fig. 4. The structure of MOP annotations.

followed by a required name of its underlying logic, e.g., FTLTL or ERE in the examples in Figures 1 or 3, respectively, followed by optional monitor configuration attributes (which will be discussed shortly), followed by the main body of the annotation, whose format is logic-specific. In case of FTLTL, e.g., the body of the specification has two parts: one containing declarations of predicates based on program states, and the other containing the FTLTL formula built on these.

The last part of the annotation contains user-defined actions, including a *violation handler* and a *validation handler*. Catching the violation is a programmer's main concern in most cases, and then common actions are to throw exceptions or produce error logs; MOP gives one full freedom in how to continue the execution of the program. One may also want to take an action when the specification is fulfilled at runtime. For example, a requirement "eventually F", which is validated when F first holds, may trigger a certain action. All these suggest a strong sense of programming with logics in MOP.

Monitor Configuration Attributes. Monitors generated from specifications can be used in different ways and can be integrated at different places, depending on specific needs of the application under consideration. For example, some specifications need to be checked at only one point (e.g., pre- or post-conditions), while others must be monitored at any point in the program (e.g., class invariants). In some applications, one may want generated monitors to use the same resources as the rest of the program, in others one may want to run the monitors as different processes. To give the software developers maximum flexibility, MOP allows one to configure the monitors using attributes in annotations.

The ***uniqueness*** attribute, denoted `!`. The default monitor configuration behavior is to create a monitor instance for each object of the corresponding class. However, if a monitor is specified as *unique*, it will only have one instance at runtime, regardless of the number of objects of that class.

There are two running mode attributes, stating how the generated monitoring code is executed. One is ***in-line*** versus ***out-line*** monitoring. Under *in-line* monitoring, which is the default, the monitor runs using the same resource space as the program. The monitor is inserted as one or more pieces of code into the monitored program. In the *out-line* mode, the monitoring code is executed as a different process, potentially on a different machine or CPU. The in-line monitor can often be more efficient because it does not need inter-process communication and can take advantage of compiler optimizations. However, an in-line monitor cannot detect whether the program deadlocks or stops unexpectedly. Out-line monitoring has the advantage that it allows, but does not enforce, a centralized computation model, that is, one monitor server can be used to monitor multiple programs. In order to reduce the runtime overhead of out-line monitoring in certain applications, one can define communication strategies; for example, the monitored program can send out the concrete values of the relevant variables and the out-line monitor evaluates the state predicates, or, alternatively, the monitored program sends out directly the boolean values of the state predicates.

The other running mode attribute is ***on-line*** versus ***off-line***. Under *on-line* monitoring, which is the default, specifications are checked against the execution of the program dynamically and run-time actions are taken as the specifications are violated or validated. The *off-line* mode is mostly used for debugging purposes: the program is instrumented to log an appropriate execution trace in a user-specified file and a program is generated which can analyze the execution trace. The advantage of off-line monitoring is that the monitor has random access to the execution trace. Indeed, there are common logics for which on-line monitoring is exponential while off-line monitoring is linear [24].

Note that the two running mode attributes are orthogonal to each other. In particular, an in-line off-line configuration inserts instrumentation code into the original program for generating and logging the relevant states as the program executes. Alternatively, an out-line off-line configuration generates an observer process which receives events from the running program and generates and then logs the states relevant to the corresponding specification. The latter may be desirable, for example, when specifications involve many atomic predicates based on a relatively small number of program variables; in this case, the runtime overhead may be significantly reduced if the program is instrumented to just send those variables to the observer and let the latter evaluate the predicates.

In the case of on-line monitoring, a further attribute, ***synchronization***, is possible. This attribute states whether the execution of the monitor should block the execution of the monitored program or not. In in-line monitoring, for example, a synchronized monitor is executed within the same thread as the surrounding code, while an asynchronous one may create a new execution thread and thus reduce the runtime overhead on multi-threaded/multi-processor platforms. Synchronization also plays a role in synthesizing code from logic formulae, because, for some logics, asynchronous monitoring is more efficient than synchronous monitoring. Consider, for example, the formula "next F and next not F" which is obviously not satisfiable: a synchronous monitor must report a violation right away, while an asynchronous one can wait one more event, derive the formula to "F and not F", and then easily detect the violation by boolean simplification. Note that synchronous monitoring requires running a satisfiability test, which for most logics is very expensive (PSPACE-complete or worse).

The ***effective scope*** attribute, which can be *class*, *method*, *block* or *checkpoint*, specifies the points at which the monitoring code is merged with the monitored program. As indicated by name, the class attribute specifies an invariant of a class, so it should be checked whenever the class state changes. The method one refers to a specific method, which can be further divided into three subcategories: pre-method (before the method), post-method (after the method), and exceptional-method (when the method raises exceptions). The annotation can also be associated to a block, e.g., as a loop invariant/variant. Similar to the method annotation, it is divided into three subtypes, namely, pre-block (before the block), post-block (after the block), and exceptional-block (when the block throws exceptions). Finally, the checkpoint attribute, which is the default, states that the monitoring code should be integrated at the exact current place in the program. Section 2.2 describes in more depth the monitor integration problem. ***Retrieving Program Information.*** Since specification and implementation live together in MOP, the specification needs information about the program in order to define and synthesize its runtime behavior. The general principle is to use the elements of the program to construct the elements of the specification, such as the atomic predicates which are defined on the state of the program. Three other pieces of information about the program, besides its state, have turned out to be practically important in MOP to enhance its expressibility and functionality, namely, *positions*, *results* and *historic references*. Figure 5 shows an example of how a JASS specification in Java-MOP can use such information.

```
/*@ JASS {post-method}
   @ ensures x == Old.x ;
   @ ensures Result == a * a ;
   Violation Handler:
   throw new Exception("Method "+@FNCT_NAME_+" violated property "+ @ANNT_Name);
@*/
int square(int a) { ... (Java Code) ... }
```

Fig. 5. Specification retrieving program information.

The *position* information is extracted via special variable names, such as `@METHOD_NAME_` for the name of the method that contains the current annotation. Position information is useful to generate effective messages to users. Many other special position variables are available, including ones for class and file names, line numbers, etc. The use of predefined variables allows certain specifications to be independent of their position and of changes of names of methods, classes, etc. The *result* and *historic references* are useful to state certain properties, such as pre- and post-conditions of methods. The specification in Figure 5 states that after each execution of the `square` method, the value of the class field `x` stays unchanged and the result of the method is indeed the square of its input.

2.2 Extensible Logical Framework

To allow specifications using potentially different underlying formalisms, support for attaching new logical frameworks to an MOP environment is needed. The layered MOP architecture in Figure 6 is currently adopted. It is especially tuned to facilitate extending the MOP framework with new logical formalisms added to the system as new components, which we simply call *logic plug-ins*. More specifically, a logic plugin is usually composed of two modules in the architecture, namely a *logic engine* and a *language shell*. By standardizing the protocols between layers, new modules can be easily and independently added, and modules on a lower layer can be reused by those on the adjacent upper layer.

The first layer provides a friendly user interface, via which one can define, process, and debug MOP annotations. The second layer contains *annotation processors*, each specialized on a specific target programming language. It consists essentially of a program scanner, which isolates annotations to be processed by specialized modules and report errors if these do not follow the expected structure. Annotation processors extract the specifications and dispatch them to the

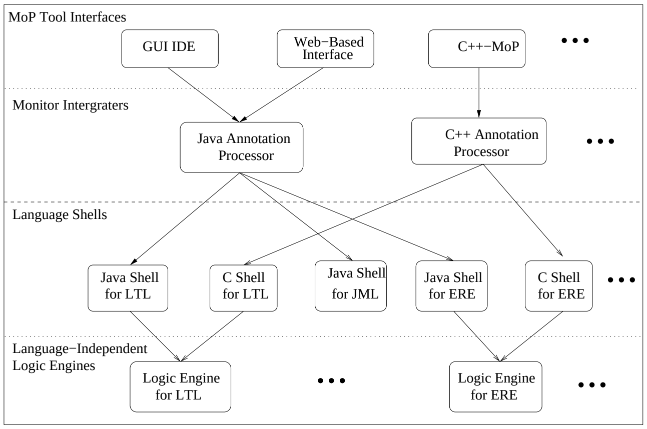

Fig. 6. MOP Architecture.

corresponding logic plug-ins. Then they collect the generated monitoring code and integrate it within the monitored program according to the configuration attributes. The two upper levels are not expected to change or update frequently.

The two lower levels contain the *logic plug-ins*, which may be added, removed, or updated on a frequent basis. To employ a new specification language, the user needs to develop a new logic plug-in, or obtain one from others or the provided WWW repository (see Section 3), and add it to her MOP framework. Then, annotations using the new formalism can be processed in a "push button" manner. Monitors generated for a specific programming language and specification formalism are generated on the third layer, by modules called *language shells*, acting as intermediaries between annotation processors and logic engines. In some simple cases, such as DBC or JML, the shell can directly generate monitors without the need of a logic engine. Modules on the bottom layer are the *logic engines*, forming a core and distinguished feature of MOP. These are generic translators from logic formulae into efficient monitors. Their output uses abstract pseudocode, which is further translated into specific target code by language shells. Therefore, logic engines can be reused across different languages.

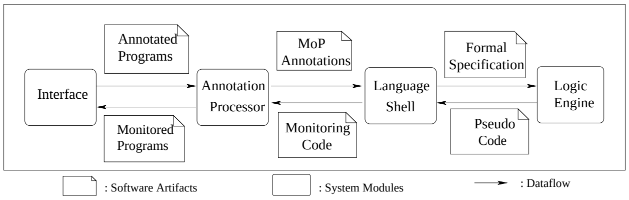

Fig. 7. Program transformation flow in MOP.

The protocols between adjacent layers are based exclusively on ASCII text, to achieve maximum flexibility, extensibility, and to ease debugging and testing. Basically, components in this architecture are implemented as individual programs receiving ASCII text from the standard input and outputting ASCII text to the standard output. Hence, the modules can be implemented by different roles using different languages, and can run on different platforms. Figure 7 shows how an MOP supporting tool transforms an annotated program.

Interface of Logic Plug-ins. An important interface is the one between the upper two layers and the lower ones, allowing MOP supporting tools to interact with the logic plug-ins. The input of a logic plug-in is clearly a formal specification, but the format of its output is less obvious. We next discuss five dimensions that the output of any logic plug-in must consider in order to be attachable to an MOP supporting tool, regardless of its internal monitor synthesis technique. Several concrete logic plug-ins that have already been experimentally evaluated using our JAVA-MOP prototype are discussed in [5].

(1) Declarations. Variables storing the state of the monitor. These will be automatically inserted at appropriate places in the program, depending upon the target programming language and the configuration attributes of the monitor.

(2) Initialization. This phase prepares the variables for starting the monitoring and is executed only once, the first time the monitoring breakpoint is encountered during the execution of the program.

(3) Monitoring body. This is the main part of the monitor, executed whenever the monitor breakpoint is reached.

(4) Success condition. This gives the condition stating that the monitoring requirement has been fulfilled, so that there is no reason to monitor it anymore. For example, for a formula "eventually F" in future time temporal logic, this condition becomes true when F holds the first time.

(5) Failure condition. This gives the condition that states when the trace violates the requirements. When this condition becomes true, user-provided recovery code will be executed. "Recovery" should be taken with a grain of salt here, because such code can not only throw an exception or put the system in a safe state, but also attach (any) new functionality to the program.

Logic Plug-ins Development. Designing and implementing a monitoring logic plug-in is a highly nontrivial task, requiring a careful analysis of trade-offs among various existing possibilities, or even developing entirely new algorithms. Two or more logic plug-ins can be possible for a given logic, each with different advantages and disadvantages. One typical situation is for a logic to admit a logic plug-in in which all the states of the monitor are explicitly generated, such as an automaton, and one in which only one current state of the monitor is stored, but a means to generate the next state on-the-fly is provided. A major benefit of the modular MOP architecture is that users are encouraged to develop such logic plug-ins *once and for all*, and then to post them in WWW repositories. We have devised monitor synthesis algorithms for future time, past time, metric and epistemic temporal logics, extended regular expressions and JASS (Section 3); and we are currently developing logic plug-ins for other logics, including real-time logic (RTL) [23] and EAGLE [3]. A logic plug-in contains two sub-modules, a *logic engine* and a *programming language shell*. The former generates monitoring pseudocode from the input specification, and then the later turns that pseudocode into target programming language code. This way, a mathematically skilled developer of the specific logic engine does not need to know the target programming language, and the logic engine can be reused for different programming languages. The language shells are usually straightforward.

Monitor Integration. Once a monitoring code is generated, an MOP supporting tool needs to integrate it into the original program according to the configuration attributes. Specifically, under the in-line mode the monitoring code is placed at proper positions within the monitored program, while under the out-line mode the MOP tool firstly constructs a stand-alone monitor program and then generates corresponding communication code which is placed at proper positions into the monitored program. For the off-line case, code that logs the relevant execution trace is inserted at specific places in the program and a stand-alone analysis program is also generated, but its role is to detect errors in the log rather than to influence the execution of the program at runtime.

The effective scope attribute says where the monitoring, communication or logging code needs to be placed. The method-related monitor is placed at the beginning of the method or at the end of it, and the monitor associated to the block is handled similarly, but with a smaller scope. The checkpoint monitor is easy to integrate by replacing the annotation with the generated code.

Class-scoped monitors, i.e., those generated from annotations with the configuration attribute `class`, are the most difficult to integrate; JAVA-MOP currently supports only a partial integration of these monitors. One approach towards a complete solution would be to insert the monitoring code at all places where relevant events can take place, e.g., before or after method calls, or after variable assignments. However, this simplistic approach can add an unacceptable runtime overhead and may raise some additional technical difficulties because of aliasing. JASS provides a tempting trade-off to reduce the complexity of this problem, by requiring that the specification holds only in those states that can be acknowledged by the client of a library, e.g., before or after the invocation to a method. We are currently adopting this trade-off in our JAVA-MOP prototype, which may admittedly allow safety leaks in critical systems.

Monitor integration gives MOP a flavor of AOP, and indeed, our current prototype uses ASPECTJ to realize the instrumentation. However, MOP's major purpose is to extend programming languages with logics as a means to combine formal specification and implementation, rather than to explicitly address crosscutting concerns. The essential and most challenging parts in the development of a framework supporting MOP are to set up the general logic-extensible infrastructure and to develop the individual logic plug-ins. AOP techniques are only employed to facilitate the final monitoring code integration. Besides, MOP supports local specifications, such as block properties and/or checkpoint, which are not considered in AOP. Moreover, MOP is concerned with the runtime behavior of the program rather than with its static structure. As previously mentioned, our current prototype can not support rigorous class invariants monitoring, because the required runtime support is beyond the current abilities of AOP.

3 Java-MOP

We next present JAVA-MOP, an MOP prototype framework. It consists of an integrated collection of specialized programs, allowing one to easily, naturally and automatically process annotations in Java programs, via a user-friendly interface. It is also available for download from the MOP website [21].

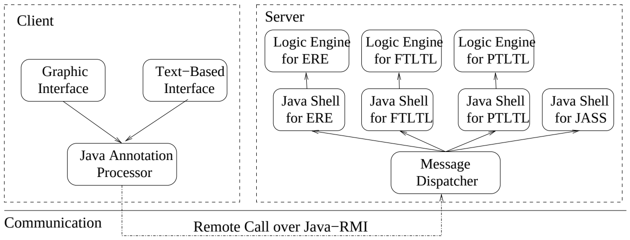

Fig. 8. The Architecture of Java-MOP.

Overview. Java-MOP is essentially a distributed client-server application, whose architecture is shown in Figure 8 and whose implementation is discussed in more depth in [5]. Users of Java-MOP only need to download and install the client, which is implemented entirely in Java and is platform independent. One can also download and start the server locally and then reconfigure the client to request monitor synthesis services locally, but this is not necessary.

The client provides both a textual and a graphical user interface. The textual one consists of a simple command taking one or more names of annotated Java files as input and generating corresponding files in which monitors are synthesized and integrated appropriately. The textual interface is typically used for batch processing, which is recommended when lots of monitors need to be generated and/or when they involve complex logical formulae which need a long time to be processed.

A friendly GUI interface is also implemented and available for download, based on the Eclipse platform [10]. The Java-MOP tool provides an editor plug-in for Eclipse, called Annotated Java Editor, which can be used to process MOP annotations in a interactive way.

Logic plug-ins. We have developed several logic plug-ins so far, which are currently supported by Java-MOP. The discussion here is rather informal, but [5] gives formal definitions and monitor synthesis algorithms. The interested reader is encouraged to try all the examples in this paper, as well as many others, at our WWW plug-in repository [21], where one can generate and visualize monitors via an HTML client.

Design by Contract - JASS***.*** Since the various DBC approaches are based on the idea of compiling special annotations into runtime checks, MOP can smoothly include these approaches, provided that appropriate logic plug-ins are developed. To test and evaluate this hypothesis, we have implemented a Java logic plug-in for JASS. The original syntax of JASS annotations has been slightly modified to fit the uniform, logic-independent syntactic conventions in Java-MOP. JASS supports the following types of assertions: method pre-conditions and post-conditions, loop variants and invariants, and class invariants. Pre-conditions must be satisfied by the caller but the callee is in charge of checking it. Post-conditions need to be satisfied at the exit of a method. Loop variants, used to check termination of loops, are integer expressions that must decrease at every iteration and whose evaluation should never reach 0. Loop invariants are checked at the boundaries of a loop, and class invariants are properties over the fields of an object and are checked on the boundaries of methods. JASS' logic is relatively simple, so its monitor generation is straightforward.

Temporal Logics. Temporal logics have proved to be indispensable expressive formalisms in the field of formal specification and verification of systems [22, 8]. Many practical safety properties can be naturally expressed in temporal logics, so these logics are also useful as specification formalisms in our MOP framework. We implemented plug-ins for both future and past time temporal logics. In future time linear temporal logic (FTLTL), one can state properties about the future execution of a system. Consider the example in Figure 1, which formally states in FTLTL a safety property of a traffic light controller: “if the green light is on then the red light should not be turned on until the yellow has been turned on”. Figure 2 already showed the final monitoring code monitor generated from this specification. Figure 9 shows the output of the FTLTL logic plug-in, before it is merged within the original code by the annotation processor, as well as the automaton generated by the HTML client to the FTLTL logic plug-in. Note that the additional variables in the generated code have generic names, starting with the `$` symbol; the annotation processor will replace these by appropriate fresh names, to avoid conflicts with other variable names already occurring in the program. Dually, in past time linear temporal logic (PTLTL), one can specify properties referring to the past states. PTLTL has been shown to have the same expressiveness as FTLTL [13]. However, it is exponentially more succinct than FTLTL [19] and often more convenient to specify safety policies.

```
// Declaration
   int $state = 1;
// Monitoring body
   switch($state) {
   case 1: $state = (tlc.state.getColor() == 3) ? 1 :
             (tlc.state.getColor() == 2) ? (tlc.state.getColor() == 1) ? -2 : 2 : 1; break;
   case 2: $state = (tlc.state.getColor() == 3) ? 1 :
             (tlc.state.getColor() == 1) ? -2 : 2; break;
   }
// Failure condition
   $state == -2
```

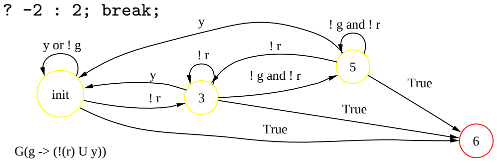

Fig. 9. Generated monitor for Figure 1 and its graphical visualization.

Extended Regular Expressions. Software engineers and programmers understand easily regular patterns, as shown by the interest in and the success of scripting languages like PERL. We believe that regular expressions provide an elegant and powerful specification language also for monitoring requirements, because an execution trace of a program is a string of states. Extended regular expressions (ERE) add complementation to regular expressions, allowing one to specify patterns that must *not* occur. Complementation gives one the power to express patterns on traces non-elementarily more compactly. However, complementation leads to a non-elementary exponential explosion in the number of states of the corresponding automaton, so naive ERE monitoring algorithms may be impractical. Preliminary efforts in [25] show how to generate simply exponential monitoring algorithms for ERE. A logic engine for ERE and a corresponding logic plug-in incorporating these algorithms has been implemented.

Online logic plug-in repository. Due to the standardized interface of MOP components, once a logic plug-in is developed, which in most cases is expected to be a highly nontrivial and time-consuming task, potentially any other MOP user can use it. To facilitate the reuse of logic plug-ins and to encourage users to propose, design, implement, and share them, we set up an online WWW repository of logic plug-ins [21]. The MOP user can try a logic plug-in online, via a provided HTML client, before she decides to download and install it. This repository is also implemented so that it can serve as a remote logic plug-in server, as discussed in Section 3. Thus, it is possible for the user to install few or no logic plug-ins on her machine, but to retrieve them by need from the repository. This facilitates the usage of the MOP framework, because the user does not need to search, download, compile and install all wanted plug-ins on her own server.

4 Conclusion and Future Work

We presented monitoring-oriented programming (MOP) as a general software development and analysis framework, aiming at increasing the quality of software by merging formal specifications into programs via monitors synthesized and integrated automatically. Specification formalisms can be easily added to and removed from an MOP supporting system, due to logic plug-ins. A WWW repository of logic plug-ins allows MOP users to download or upload them, thus avoiding the admittedly heavy task of developing their own logic plug-ins. The logic-independence and the broad spectrum of generated monitor integration capabilities allow MOP to paradigmatically capture other related techniques as special important cases. We also introduced JAVA-MOP, a prototype system supporting most of the desired features of MOP.

There is much future work to be done, part of it already started. We intend to extend our prototype to support all the running modes soon, to incorporate other related approaches, such as JML, and especially to evaluate the MOP paradigm on students and in practical large size applications. Advanced techniques to dynamically "observe" the execution of a program in as much depth as needed are imperiously needed; the current AOP capabilities that we are using are rather static and limited. We believe that MOP may also lead to novel programming techniques, and more work will be dedicated to methodologies for developing MOP-based high quality software.

References

1. P. Abercrombie and M. Karaorman. jContractor: Bytecode Instrumentation Techniques for Implementing DBC in Java. In *ENTCS*, volume 70. Elsevier, 2002.
2. Aspectj project. http://eclipse.org/aspectj/.

3. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Intl. Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, LNCS, 2004.
4. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with Assertions. In *ENTCS*, volume 55. Elsevier, 2001.
5. F. Chen, M. D'Amorim, and G. Roşu. Monitoring-Oriented Programming. Technical Report UIUCDCS-R-2004-2420, Univ. of Illinois Urbana-Champaign, 2004.
6. F. Chen and G. Roşu. Towards Monitoring-Oriented Programming: A paradigm combining specification and implementation. In *Workshop on Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 106–125. Elsevier, 2003.
7. Y. Cheon and G. T. Leavens. A Runtime Assertion Checker for JML. In *Software Engineering Research and Practice (SERP'02)*. CSREA Press, 2002.
8. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
9. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LCNS*, pages 323–330. Springer, 2000.
10. Eclipse project. http://www.eclipse.org.
11. Eiffel Software. The Eiffel Language. http://www.eiffel.com/.
12. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'02*, pages 234–245, 2002.
13. D. M. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In *Conference on Temporal Logic in Specification*, volume 398 of *LNCS*, pages 409–448. Springer, 1989.
14. K. Havelund and G. Roşu. *Workshops on Runtime Verification (RV'01, RV'02, RV'04)*, volume 55, 70(4), to appear of *ENTCS*. Elsevier, 2001, 2002, 2004.
15. K. Havelund and G. Roşu. Runtime verification. *Formal Methods in System Design*, 24(2), 2004. Special issue dedicated to RV'01.
16. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
17. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Runtime Assurance Tool for Java. In *Runtime Verification (RV'01)*, ENTCS. Elsevier, 2001. vol. 55.
18. 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.
19. N. Markey. Temporal Logic with Past is Exponentially more Succinct. *EATCS Bull*, 79:122–128, 2003.
20. B. Meyer. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, Upper Saddle River, New Jersey, 2000.
21. Mop website. http://fsl.cs.uiuc.edu/mop.
22. A. Pnueli. The Temporal Logic of Programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
23. R. Alur and T.A. Henzinger. Real-Time Logics: Complexity and Expressiveness. In *IEEE Symposium on Logic in Computer Science*, pages 390–401. IEEE, 1990.
24. G. Roşu and K. Havelund. Rewriting-based Techniques for Runtime Verification. *Automated Software Engineering*, to appear in 2005.
25. K. Sen and G. Roşu. Generating Optimal Monitors for Extended Regular Expressions. In *Workshop on Runtime Verification (RV'03)*, ENTCS, 2003. vol. 89.
26. O. Sokolsky and M. Viswanathan. *Workshop on Runtime Verification (RV'03)*, volume 89 of *ENTCS*. Elsevier, 2003. Computer Aided Verification (CAV'03).
27. P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton. N Degrees of Separation: multi-dimensional separation of concerns. In *ICSE'99*, pages 107–119, 1999.