

# Java-MOP: A Monitoring Oriented Programming Environment for Java

Feng Chen and Grigore Roşu

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

**Abstract.** A Java-based tool-supported software development and analysis framework is presented, where monitoring is a foundational principle. Expressive requirements specification formalisms can be included into the framework via logic plug-ins, allowing one to refer not only to the current state, but also to both past and future states.

## 1 Introduction

This paper presents a monitoring oriented programming (MOP) software development and analysis environment for Java, named JAVA-MOP. Based on the belief that specification and implementation should *together* form a system and interact with each other by design, we proposed the MOP framework [2], aiming at increasing the quality of software through monitoring of formal specifications against running programs.

There are several software development approaches in the literature based on the very basic idea of monitoring. *Design by Contract* (DBC) [10] related approaches, e.g., JASS [3] and JML [8], allows specifications to be associated with classes as assertions and invariants, which are compiled into runtime checks. Runtime verification (RV) [5] is an expanding area dedicated to provide more rigor in testing, essentially as a complementary approach to model checking software systems. There are several RV systems, including JAVA-MAC [7], TEMPORAL ROVER and its follower DB ROVER [4], JPAX [6] and its followers EAGLE [1] and JMPAX [12].

What distinguishes the MOP from these approaches is its ability to be extended with new logics and to support self-recovery at violation. Practice has shown that there is no “silver bullet” logic to formally express any requirements. Some can be best expressed using a certain logical formalism, for example temporal logics, while others can be best expressed using other logics, like that of JML, or domain-specific logics. On the other hand, programming languages are intended to be universal. For these reasons, MOP provides the capability of adding logics on top of a target programming language via logic plugins.

Monitoring can also provide a strong foundation for increasing the quality, robustness, and confidence in the correctness of complex software systems. The Simplex [13] architecture shows an example to smoothly upgrade control systems based on monitoring. Therefore, MOP supports the user to define violation and/or validation handlers along with specifications, which can be highly complicated recovery actions. These handlers will be automatically triggered at runtime when the specification is violated or validated, in order to recover the program from unsafe states.

## 2 Overview of JAVA-MOP

JAVA-MOP is a development tool for Java, supporting the MOP paradigm. It provides both GUI and command-line interfaces for editing and processing specifications. Algorithms to synthesize optimized monitoring code for different logics have been implemented in order to incorporate useful specification languages into JAVA-MOP. Moreover, users are able to easily incorporate new formalisms which can be used later in specifications via logic plugins. JAVA-MOP does not only aim at specifying and monitoring system behaviors, but also gives users the ability to recover from errors at runtime. Here we only briefly present major features of JAVA-MOP. Interested readers can refer to our website [11] for the distribution package and related documents.

**Extensible architecture.** JAVA-MOP follows a distributed architecture in design as shown in figure 1. This design facilitates extending the framework with logic formalisms added to the system as new components, which we simply call *logic plug-ins*. These components are usually comprised of two modules, namely a *logic engine* and a *language shell*. For example, the logic engine for ERE and the Java shell for ERE form the logic plugin for extended regular expressions. Logic engines translate formulae into efficient monitors, presented in some abstract representation (pseudocode). Then language shells transform abstract monitors into code for a specific language, e.g., Java. The output interface of the logic plugin is standardized. This way, if a new logical formalism is needed to specify the requirement of a certain application, then one can develop a synthesis algorithm for the specific logic, wrap the algorithm as a logic plugin for Java, and add the plugin into the JAVA-MOP. For some simple specification languages, or for programming-language-specific formalisms, such as Jass, the logic engine is unnecessary and the language shell only is sufficient to generate the monitor.

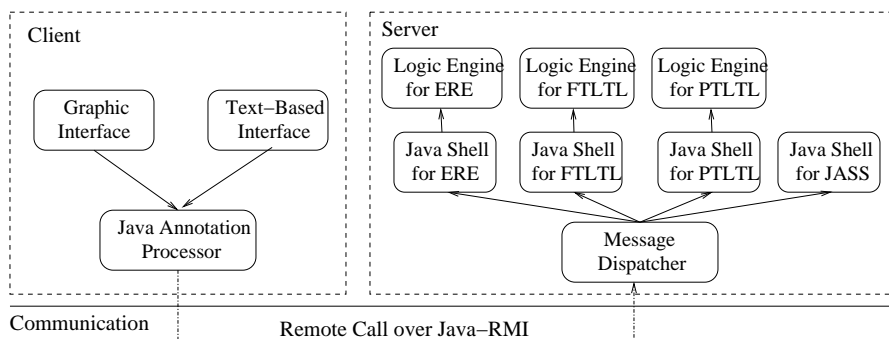


Fig. 1. The Architecture of JAVA-MOP

The client part contains the Java annotation processors, which integrate the monitoring code generated by the server into the system, according to configuration attributes of the monitor. In addition, the client part is also in charge of instrumenting the code to generate events to be monitored. Currently, JAVA-MOP is using ASPECTJ as the instrumentation mechanism. ASPECTJ aspects

are produced for specifications to be monitored. ASPECTJ, however, also imposes some limitation in our implementation. The integration made by ASPECTJ is static while the monitoring is dynamic. This brings difficulties and inefficiencies when monitoring dynamic entities. For example, for a class invariant, one may need to monitor every update of `anObject.aField` instead of `afield` of any object whose class is the same as `anObject`.

**Monitor synthesis.** Every logic plugin essentially encodes an algorithm to synthesize monitoring code for a specific formalism. We have devised monitor synthesis algorithms for future time and past time temporal logics, as well as for extended regular expressions, JASS, and JML.

- *JML and Jass.* These DBC-based approaches follow the idea of including specifications into the code and then pre-compiling them into runtime checks. So we are able to smoothly include them in JAVA-MOP. The original syntax of JML and JASS annotations has been slightly modified, to fit the uniform, logic-independent syntactic conventions in JAVA-MOP.
- *Temporal Logics.* Temporal logics have proved to be indispensable and expressive formalisms in the field of formal specification and verification of systems [9]. Since MOP can be regarded at some extent as a complementary, but still related, approach to formal verification, we provide logic plug-ins to support past and future time variants of temporal logics.
- *Extended Regular Expression.* Regular expressions provide an elegant and powerful specification language for monitoring requirements, because an execution trace of a program is in fact a string of states. Extended regular expressions (ERE) add complementation to regular expressions, which gives one the power to express patterns on traces non-elementarily more compactly. A logic plugin for ERE has been incorporated into JAVA-MOP.

**Steering Behaviors of Monitors.** Besides adding more rigor to testing, MOP is especially intended to be a monitoring tool to assure correctness during program execution. To support runtime recovery, MOP provides a the capability to steer the execution of the program when requirements are violated or validated.

Users can provide handlers for the violation or validation of monitored properties. These handlers can not only report errors or throw exceptions, but also execute complicated actions, e.g., resetting states or rebooting the system. Therefore, critical monitors can be automatically integrated into the final system to correct the system at runtime.

MOP monitors can have different running scope. It can be a class invariant, which is checked at every change of the class state. Or it can be a interface constraint, which is checked when a client invokes the interface of the class. JAVA-MOP also supports method pre-/post- conditions and checkpoint assertions. Besides, users can also choose if the system needs to wait for the checking result or not. The keywords *synchronized* and *asynchronized* are used for this purpose. The motivation behind asynchronous monitors is that some properties are not critical and the system does not have to react to the violation. In such cases, asynchronous mode can avoid unnecessary waiting and reduce the runtime overhead. Besides, some logics, e.g., context-free languages, may require the generated monitor to wait until the next events to proceed.

### 3 Conclusion

This paper presents a development and analysis environment for Java, which supports the MOP paradigm. Monitors will be generated from formal specifications and then used to verify the execution of the system. Users can define self-recovery actions for the violation of specifications. More logic plugins for useful specification languages will be added into JAVA-MOP in order to support different domain requirements.

### References

1. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. In *Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD'04) (Satellite workshop of IPDPS'04)*, Santa Fe, New Mexico, USA, April 2004. IEEE digital library.
2. F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume to appear of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
3. M. M. Detlef Bartetzko, Clemens Fischer and H. Wehrheim. Jass - java with assertions. In *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.
4. D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *Proc. of CAV'03: Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 114–118, Boulder, Colorado, USA, 2003. Springer-Verlag.
5. 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.
6. K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.
7. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
8. 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*, pages 105–106, 2000.
9. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.
10. B. Meyer. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, Upper Saddle River, New Jersey, 2000.
11. Mop website. <http://fsl.cs.uiuc.edu/mop>.
12. K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of 4th joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, (ESEC/FSE'03)*. ACM, 2003.
13. L. Sha, R. Rajkumar, and M. Gagliardi. The simplex architecture: An approach to build evolving industrial co mputing systems. In *Proceedings of The ISSAT Conference on Reliability*, 1994.