

# Towards a Tool for Generating Aspects from MEDL and PEDL Specifications for Runtime Verification

Omar Ochoa, Irbis Gallegos, Steve Roach, Ann Gates

The University of Texas at El Paso, Computer Science Department,  
500 W. University Av. El Paso, TX, 79912, USA  
{omar, irbisg, sroach, agates}@utep.edu

**Abstract:** This paper describes a technique to generate AspectJ aspects from formal specifications written for the Java-MaC runtime verification tool. The aspects are used to instrument programs for runtime verification. To demonstrate the described approach, it is applied to a “benchmark” from formal methods research, i.e., a safety-critical railroad crossing system composed of a train, a gate and a controller. Finally, the results are described on generating Java-MaC’s specification scripts to AspectJ aspects.

**Keywords:** Runtime Verification, Java-MaC, Aspect Oriented Programming, Runtime Monitoring, Software Assurance.

## 1 Introduction

With the ubiquity of software, especially in safety-critical systems, avoiding software failures *must* be emphasized. The failure of safety-critical systems, such as airplane controllers or railroad crossing systems, can result in monetary loss, injury or death. Software testing, the most commonly used verification technique, cannot provide complete test coverage and designing effective comprehensive test suites for complex systems is difficult. A complementary technique is runtime verification, which examines actual execution paths, not possible paths. In this approach, a monitor system observes the behavior of a system and determines if it is consistent with specified properties. A monitor takes a software system and specifications of software properties and checks that the execution meets the properties, i.e., that the properties hold for the given execution [1].

A programming paradigm called Aspect Oriented Programming (AOP) [2] allows developers to encapsulate cross-cutting concerns that are needed to develop effective runtime verification approaches, which require instrumentation of specifications in software code. This paper examines the integration of AOP into the runtime verification approach called Monitoring and Checking (MaC) [3]. MaC was chosen because it supports the separation of monitoring and specification requirement level concerns by using two separate languages, one that deals with details of implementation and another one that deals with requirements.

MaC is a framework for run-time correctness and assurance of real-time systems. Currently a prototype implementation for programs written in the Java language exists (Java-MaC). MaC offers well-defined specification languages based on Linear time Temporal Logic (LTL) [4] in which the underlying structure of time is a totally ordered set  $(S, <)$  isomorphic to the natural numbers with their usual ordering  $(\mathbb{N}, <)$ . Under this definition, time is discrete, has an initial moment with no predecessors, and is infinite into the future.

The goal of the work is to make runtime verification more accessible to developers, thus supporting the ability to detect failures in safety-critical systems. In particular, this paper reports how to extend the MaC runtime verification system to use the instrumentation mechanisms from the AOP paradigm. This is important because it relieves developers from being responsible for maintaining the instrumentation process. In addition, it provides an approach that will allow the runtime verification community to benefit from research advances in AOP.

The paper is divided as follows: Section 2 presents a detailed overview of Java-MaC and AOP. Section 3 describes the proposed approach and an example illustrating it. Section 4 presents the related work. Finally, Section 5 provides concluding remarks.

## 2 Background

In this section, an overview is provided of the most relevant features of Java-MaC and Aspect Oriented Programming.

### 2.1 Java-MaC

The Java-MaC framework allows users to specify system states to be monitored, define high-level events based on run-time system states, and describe correctness properties in terms of high-level events. The framework uses a runtime component called a *filter* to track the collection of probes inserted into the target program and a separate runtime component called an *event recognizer* to detect events from the state information received from the filter.

The Meta-Event Definition Language (MEDL), based on an extension of LTL, is used to express a large subset of safety properties of systems, including real-time properties such as “when a train is crossing, the gate is down.” The Primitive Event Definition Language (PEDL) is used to describe events and conditions in terms of system objects such as methods and variables. PEDL specifications define the events recognized by the event recognizer, and these event definitions are used to automatically instrument the original program. The event recognizer emits event streams to the run-time checker that verifies the sequence of events with respect to the specified MEDL properties [6]. Fig. 1. depicts a dataflow diagram for the Java-MaC framework.

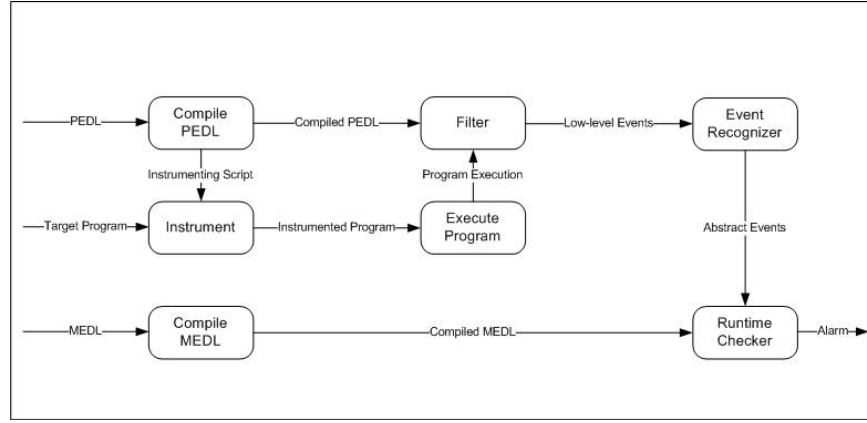


Fig. 1. Java-MaC Framework Data-Flow Diagram

### 2.1.1 Property and Behavior Specifications

Java-MaC defines two types of state information, *events* and *conditions*. *Events* are asserted instantaneously during the system execution, whereas *conditions* represent information that holds true for a duration of time [6].

The distinction between the two determines what the monitor can infer about the execution. The monitor can conclude that an event does not occur at any moment until it receives an update from the filter. By contrast, once the monitor receives a message from the filter, it will determine if a state change has occurred.

In Java-MaC, PEDL is closely related to the target programming language because events are defined using program entities such as variables and methods [7]. Each declaration identifies an object that needs to be monitored. The object resides in a memory location, since the exact memory location of the object is not known during the static phase, this object is specified in a monitoring script as a chain of references that starts in a fixed place in the object graph of the Java program; either a static variable of a class, a local variable of a static method, or the beginnings and endings of methods. When such a chain of references in a monitoring script is specified, it becomes a name for the memory location of the monitored objects [7]. PEDL allows the monitoring of fields and variables of the primitive type, but does not allow objects to be monitored directly to minimize monitoring overhead.

Additionally, domain specific safety requirements are written in MEDL. Primitive events and conditions in MEDL specifications are imported from PEDL specifications. The correctness of the system is described in terms of safety properties and alarms. Safety properties are conditions that must always be true during execution. Alarms are events that must never be raised. Alarms and safety properties are complementary ways of expressing the same thing. The reason for this is that in

some cases it is easier to think about properties in terms of conditions, while in some other cases it is easier to think in terms of alarms.

### 2.1.2 Java-Mac Instrumentation

Java-MaC monitors global primitive variables, local primitive variables, and start and end of methods. The Java-MaC instrumentor detects instructions that update monitored variables or instructions located at the beginning and ending of methods by placing bytecode instructions to overlook updates of these entities of interest.

During the static analysis phase, the Java-MaC instrumentor identifies candidate update instructions for the monitored variables. Once the instrumentor recognizes an update instruction for a monitored variable, the instrumentor inserts a method call to Java-MaC's monitoring methods, acting as a probe at the bytecode level. The probe appears immediately before the instruction and consists of the following methods: *monitorEnter()*, *Filter.lock()* and *sendObjMethod(Object parentAddress, <T> value, String varName)*, where *parentAddress* is an address of an object whose field *varName* is monitored. When *sendObjMethod()* is called at runtime, it checks if the variable being by the probe is a monitored variable by matching *parentAddress* with the address of a monitored object in the address table, i.e., a variable of interest. If the variable is a variable of interest, *sendObjMethod()* sends it to the event recognizer; otherwise, the variable is ignored.

The instrumentor inserts *monitorExit()*, *Filter.lock()* after the variable being monitored. The pair of *monitorEnter()* and *monitorExit()* ensures that the update to a variable and the sending of its new value are executed atomically. For execution points (i.e., calls and exits from methods) the instrumentor inserts probes at the starting point of a method and at the ending points of a method [7].

## 2.2 Aspect-Oriented Programming

AOP aids programmers in the encapsulation of cross-cutting concerns, i.e., specific requirements that span different modules in a system and that cannot be modularized into one component. An aspect is a class that includes constructs to support cross-cutting encapsulation through pointcuts and advice as described in the following paragraph. Aspects can include fields and methods, which are merged with classes by a program called a *weaver*. Aspect weaving can occur at the source code level, at post compilation, or at class-load time [8, 9]. Aspects provide the benefit of good modularity: code simplicity, ease of development and maintenance, and potential for reuse [10].

### 2.2.1 AspectJ

AspectJ [11] is an AOP implementation for the Java programming language. A *join point* is a place in the code where additional behavior is required. A *pointcut* is a specification of a set of join points. There are two types of pointcuts: primitive and user defined. User-defined pointcuts are boolean combinations of primitive pointcuts.

Pointcuts may match a method invocation at either the call site or the method site, at an assignment or read from a field, or at a point where some condition holds. For example, one could verify if variable  $x$  is updated by using the construct: *pointcut checkx() : set(int Class.x)*, where *checkx()* identifies the aspect, *set()* recognizes when the specified non-private field is updated, and *int Class.x* specifies field  $x$  in class *Class* as the field of interest. The behavior of the program can be changed at each join point by specifying a construct called *advice*, which is code to be executed at a join point. The constructs *before()*, *after()*, direct when the *advice* is going to be executed, either before entering the join point or after exiting the join point, respectively. Additionally, the construct *around()* executes before entering the join point like a *before()* and optionally using a *proceed()* to execute the join point or to return and not execute the join point.

### 3 Proposed Approach

This section provides a description of the proposed approach and an example of how the approach can be used to provide assurance to a safety-critical system.

#### 3.1 Description

Primitive events in PEDL correspond to transfer of control between methods or assignments to variables. PEDL events describe join points in a program. MEDL properties correspond to safety and liveness requirements; therefore, the advice at each pointcut is checked against the MEDL specification. As mentioned earlier, an aspect is composed of pointcut declarations and advice associated with each pointcut. The following paragraphs describe the generation of an aspect with respect to PEDL and MEDL.

PEDL states what variables and method calls are going to be used to generate events as well as the conditions that will be used for monitoring. As a result, for each variable and method, a pointcut must be specified. Events that are described by the PEDL keywords *update*, *IoM start*, *end*, *startM*, *endM*, are mapped to corresponding AspectJ constructs (see Table 1). For example, *startM* and *endM* correspond to the start of a method and the end of a method, which map to the *before()* and *after()* directives given to the advice modifier.

Conditions are a combination of variables and booleans that when true emit an event. This is handled by creating a method that will check its condition. If the condition is true, then a respective aspect field will contain the boolean value of the condition. In this way, if the condition is true, the aspect field will be true.

MEDL states the conditions on high-level events that must be checked on the low-level events and conditions given by a PEDL specification. Because PEDL field methods are created for conditions and low-level events, MEDL specifications will be expressed as a method inside the aspect. Such methods are called on the advice corresponding to the pointcuts extracted from PEDL, i.e., when this method gets called, it will check the boolean value of the aspect fields that represent the conditions described by PEDL. Fig. 2 uses a dataflow diagram to show the proposed

modification to Java-MaC to accommodate aspects.

Table 1. Mapping from Java-MaC to AspectJ

Java-MaC	Aspect J
update(x)	set(x)
startM()	pointcut: before()
endM()	pointcut: after()
IoM()	pointcut: around()

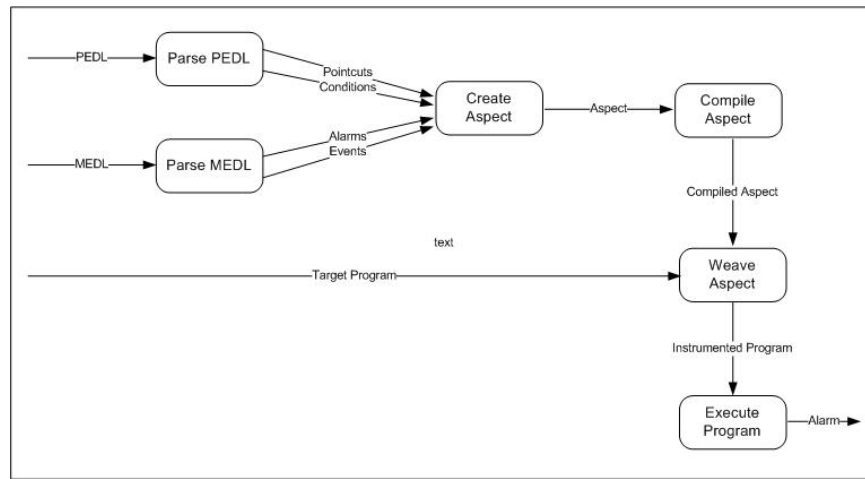


Fig. 2. Modified Java-MaC Framework Data-Flow Diagram using AOP

### 3.2 Example

The described approach is applied to the simulation of railroad crossing gates [12]. The safety property used was: when the train crosses the railroad crossing, the gates must be down. The railroad-crossing example was the modified example that is provided with the distribution of Java-MaC, which included the MEDL and PEDL specification files. The purpose of this example is to demonstrate instrumentation using the generated AspectJ aspect and to show that the specified properties, as captured by the aspects, are verified at runtime.

The PEDL file for this example contains the following conditions: *startIC* denotes a train reaching the rail road crossing; *endIC* denotes the train passing the crossing;

*startGD* denotes the gate closing; and *endGD* denotes the gate starting to rise. The MEDL file contains the conditions *IC*, which denotes “a train is crossing,” and *GD*, which denotes “a gate is down.” These conditions are represented as *Cond IC= [startIC, endIC]* and *Cond GD= [startGD, endGD]*, with the safety property *safeRRC= !IC // GD*. The aspect generated from this specification consists of the safety condition *safeRRC= !IC // GD*. Two pointcuts are generated to monitor each part of the condition. Pointcut *IC* is triggered when *train\_x + train\_length > cross\_x && train\_x <= cross\_x + cross\_length*, which represents the train crossing. Pointcut *GD* is triggered after *Gate.gd()* is executed, but before *Gate.gu()* is called, which represent the gate going down or up, respectively.

The aspect monitors the safety condition and, if it is violated, an alarm is raised. Once the aspect was generated, it is woven into the railroad application. The simple aspect-instrumented version detected the same violations as the Java-MaC-monitored version.

The following PEDL code excerpt provides the pointcuts for the AspectJ aspect. The *MonVarDcl* heading indicates that the variable declarations following the heading are to be monitored. So in this case, the variables: *train\_x*, *train\_length*, *cross\_x* and *cross\_length* in the class *RRC* are of interest. Similarly, the *MoMethodDcl* heading indicates that the methods following the heading are to be monitored, which in this instance are the methods: *gd()* and *gu()* in the class *Gate*.

PEDL Code excerpt used to provide the pointcuts for the AspectJ Aspect

```
MonVarDcl:
    float RRC.train_x;
    int RRC.train_length;
    int RRC.cross_x;
    int RRC.cross_length;

MonMethodDcl:
    Gate.gd();
    Gate.gu();
```

The MEDL code excerpt below was used to provide the advice for the AspectJ aspect. In this excerpt, four events are imported, *startIC*, *endIC*, *startGD*, *endGD*, which are the events that will be monitored based upon the properties defined later in the MEDL script. The *CondDef* heading defines the conditions to be associated with the events. In this case, the *InCrossing* heading defines the condition to specify that there is a train in the crossing. Similarly, *GateDown* defines the condition to specify that the gate in the crossing is down. Once the conditions have been defined, safety properties using these conditions are defined by using the *SafePropDef* heading. For this property it is the case that whenever a train is in the crossing (*InCrossing*), it must always be the case that the gate is down (*GateDown*).

MEDL Code excerpt used to provide the advice for the AspectJ aspect

```
ReqSpec RailroadCrossing

import event startIC, endIC, startGD, endGD;

CondDef:
  Cond InCrossing = [startIC, endIC];
  Cond GateDown = [startGD, endGD];

SafePropDef:
  SafeProp safeRRC = InCrossing -> GateDown;

End
```

After providing the tool with the MEDL and PEDL specifications for the program, the tool successfully generated the AspectJ aspect matching the MEDL and PEDL specifications as presented in the following code. Once generated, this aspect was successfully used to provide the same assurance as with Java-MaC.

AspectJ Aspect matching the MEDL and PEDL specifications

```
//PEDL equivalent section
before(RRC trgt) : set (float RRC.train_x) &&
                    target(trgt)

before() : call(void Gate.gd(int))

after() : call(void Gate.gd(int)){
  GD=true;
  monitor();
}
before() : call(void Gate.gd(int)){
  GD=false;
  monitor();
}

//MEDL equivalent section
IC = aRRC.train_x + aRRC.train_length > aRRC.cross_x &&
aRRC.train_x <= aRRC.cross_x + aRRC.cross_length;
```

## 4 Related Work

Several approaches have attempted to exploit the cross-cutting and instrumentation mechanisms of Aspect Oriented Monitoring to provide runtime verification or fault recovery capabilities. This section provides a short description of some of these efforts and how they differ to the described approach.

### 4.1 Monitoring Oriented Programming (MOP)

The University of Illinois at Urbana-Champaign's Monitoring Oriented Programming (MOP) framework [13] allows specifiers to check conformance of implementation to specification at runtime. It is an AOP-based instrumentation package allowing flexible monitors to be generated as AspectJ aspects. In MOP, runtime violations and specification validations result in adding functionality to a system by executing user-defined code at user-defined places. MOP also extends programming languages (Java for instance) with logics so that logical statements can be added anywhere in the program. So far, MOP supports logic plug-ins for future time and past time temporal logics, extended regular expressions and JASS, all available through a web repository. The difference between MOP and this approach lies in that MEDL and PEDL are used as specification languages. It has been shown that [14] Future Interval Logic (FIL) can be converted into MEDL, thus a translation a formal language such as FIL to a specification language can be done mechanically. Additionally, in MOP, for every specification a separate monitor needs to be generated. This generation process can require intensive computation power for large sets of specifications. With this approach one monitor is generated as an aspect for all specifications.

### 4.2 TRAP/J

TRAP/J [15] from Florida International University is a software tool that enables autonomic computing in existing Java programs by generating adapt-ready (programs that behavior can be managed at runtime) versions of the original programs at compile time. The generation process is transparent to the original program source code, so there is no need to modify the source code manually. In TRAP/J new behavior can be introduced to the adapt-ready programs at runtime using the wrapper- and meta-language classes. First, the adapt-ready application is loaded by the JVM. At the time each meta-object (entities of interest in the code) is instantiated, it registers itself with the Java RMI registry using a unique ID. Next, if an adaptation is required, the composer (entity that requests an adaptation to the code) dynamically adds new code to the adapt-ready application at runtime, using Java RMI to interact with the meta-objects. As part of the behavioral reflection provided in the adaptation infrastructure, a meta-object protocol is supported in TRAP/J that allows interception and reification of method invocations targeted to objects of the classes selected at compile time to be adaptable. To reduce overhead, TRAP/J enables the developer to select, at compile time, a subset of classes, constituting an existing program, to be adaptive at runtime.

To support dynamic adaptation in existing Java programs, TRAP/J benefits from aspect-oriented programming (AspectJ) and the ability of programs to reason and alter their own behavior (behavioral reflection). TRAP/J generates specific aspects and reflective classes associated with the selected classes. A case study is presented in which TRAP/J was used to enable an existing audio-streaming application to perform self-optimization in a wireless network environment by adapting to changing conditions automatically. Because TRAP/J's goal is not to support runtime monitoring but to provide fault recovery capabilities, it would have to be integrated with a system to specify runtime verification properties. Additionally, this system would have to use a formal language to describe the runtime verification specifications.

### **4.3 Temporal Assertion using AspectJ**

Stolz and Bodden [16] present Java Logical Observer (JLO), a runtime verification framework for Java programs. In their approach, properties can be specified in Linear-time Temporal Logic (LTL) over AspectJ pointcuts. These properties are checked during program-execution by an automaton-based approach where transitions are triggered through aspects, detecting violations by entering error-states.

No Java source code is necessary since AspectJ works on the byte-code level, thus even allowing instrumentation of third-party applications. The current implementation supports the full formalism, yet without access to runtime state. Improvements to parameterized formulae are discussed to overcome the issue. The difference between JLO and this approach is that with JLO AspectJ is used to aid in the tracing of execution through an automaton that represents valid and invalid error states. Compared to this approach, AspectJ is used to instrument the program based on the specification scripts provided.

### **4.4 jMonitor**

jMonitor [17] is a pure Java library and runtime utility which allows programmers to specify event patterns to monitor runtime execution of legacy Java applications. That works by overloading the dynamic class loader. The jMonitor class loader instruments the class bytecodes of the monitored Java program on the fly according to the externally specified event patterns and event monitors.

During the execution of an instrumented application, each Java bytecode instruction that matches any of the specified event patterns triggers the call of one or more associated monitor methods. The monitor methods get called with the following runtime context information regarding the triggering event: the type of event, its target object, the call stack representing the method in which the event occurred, and the arguments to the method which collectively defines the full call context when the event occurred.

jMonitor events correspond to fundamental Java programming abstractions such as reading or writing of a field in a class, method invocation, method return or throw of an exception, and creation of a new object or array. Each event is also qualified with

a Java application context such as the name of the field or the method and the names of the class and method context. The names are specified as strings representing POSIX compliant regular expressions.

Several distinct event monitors can be associated with any event. jMonitor instruments applications to capture the call context and call the monitor function with this information. Each monitoring function is called before, after or instead of the associated event depending on the event specification.

jMonitor differs with this approach in that an existing runtime verification system is being enhanced by removing the instrumentation and replacing it with an AOP approach. This benefits the MaC system because the AOP community can maintain, improve and port the instrumentor to other languages.

## 5 Summary

This work presents an extension to Java-MaC that converts MEDL and PEDL files to AspectJ aspects. The results confirm the applicability of AspectJ as an instrumentor. Code instrumentation for runtime verification is a natural AOP challenge because instrumentation is a cross-cutting concern. Much effort is being put into the development of next-generation aspect weavers, and the natural next step is to investigate how these new methods can be used for Java-MaC instrumentation. Some of the benefits of adapting AspectJ as the instrumentation mechanism include: MaC developers do not have to maintain the instrumentor; the AOP community is developing efficient ways to weave/instrument codes, which can be exploited to improve MaC instrumentors; and by using AOP, MaC will be easily extendable to other AOP-supported languages and perhaps other architectures.

Future work includes support for all features in the MEDL and PEDL languages and demonstration of the equivalence of the weaving process of AspectJ to Java-MaC instrumentation. Other future work includes determining whether this technique can be applied to runtime verification of SOA services implemented in a variety of AOP supported languages.

## References

- [1] N. Delgado, A. Gates, and S. Roach, "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools". in *Proc. IEEE Transactions on Software Engineering*, 30(12), pp.859-872, December 2004.
- [2] G. Kiczales, J. Lamping, and A. Mendhekar, "Aspect-Oriented Programming". in *Proc. European Conference on Object-Oriented Programming 1997*, 1997, volume 1241 of LNCS, pp. 220-242.
- [3] MaC. "Run-time Monitoring and Checking (MaC)". [Online] Available <http://www.cis.upenn.edu/~rtg/mac/index.php3>, November 23, 2006.

- [4] Emerson, E. A. 1990. Temporal and modal logic. In Handbook of theoretical Computer Science (Vol. B): Formal Models and Semantics, J. van Leeuwen, Ed. MIT Press, Cambridge, MA, 995-1072.
- [5] Palo Alto Research Center. "The AspectJ Programming Guide". [Online] Available <http://www.eclipse.org/aspectj/doc/released/progguide/index.html>, October 1, 2006.
- [6] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. "Java-MaC: A Run-time Assurance Tool for Java". in *Proc. 1st International Workshop on Run-time Verification*. 2001.
- [7] Java-MaC: A Run-time Assurance Approach for Java Programs, Moonjoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, Oleg Sokolsky, Formal Methods in System Design, Volume 24, Issue 2, March 2004, pages 129-155.
- [8] E. Hilsdale, and J. Hugunin, "Advice Weaving in AspectJ", in *Proc. Aspect-oriented Software Development 2004*, 2004, pp. 26-35.
- [9] Palo Alto Research Center. "The AspectJ Programming Guide". [Online] Available <http://www.eclipse.org/aspectj/doc/released/progguide/index.html>, October 1, 2006.
- [10] G. Kiczales, J. Lamping, and A. Mendhekar, "Aspect-Oriented Programming". in *Proc. European Conference on Object-Oriented Programming 1997*, 1997, volume 1241 of LNCS, pp. 220-242.
- [11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. in *Proc. of the European Conference on Object-Oriented Programming 2001*, 2001.
- [12] C. Heitmeyer and D. Mandrioli, "Eds. Formal Methods for Real-Time Systems". *Number 5 in Trends in Software*. John Wiley & Sons, 1996.
- [13] F. Chen, M. D'Amorim, G. Rosu. "A Formal Monitoring-based Framework for Software Development and Analysis". *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, 2004.
- [14] O. Mondragon, A. Q. Gates, S. Roach, H. Mendoza, O. Sokolsky. "Generating Properties for Runtime Monitoring from Software Specification Patterns". *International Journal of Software Engineering and Knowledge Engineering 17*, 2007. pp. 107-126.
- [15] S. Sadjadi, P. K. McKinley, B. H.C. Cheng, and R.E. Stirewalt. TRAP/J: Transparent generation of adaptable Java programs. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, Agia Napa, Cyprus, October 2004.
- [16] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Fifth Workshop on Runtime Verification (RV'05)*. Elsevier, 2005.

- [17] M. Karaorman and J. Freeman. “jMonitor: Java Runtime Event Specification and Monitoring Library”. *Electronic Notes in Theoretical Computer Science, Volume 113, 3 January 2005*. 2005, pp. 181-200.