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

Omar Ochoa, Irbis Gallegos, Steve Roach, Ann Gates
Department of Computer Science
The University of Texas at El Paso

In this paper, we describe an Aspect Oriented extension to the verification tool Java Monitoring and Checking (Java-MaC) [1]. This approach generates AspectJ aspects from Java-MaC specifications. We then use these aspects to monitor the program during execution [2]. To demonstrate the described approach, we apply it to a "benchmark" from formal methods research [3], a safety-critical railroad crossing system composed of a train, a gate and a controller. In this system, the gate must be down while the train is crossing and up when no train is crossing.

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) is based on an extension of linear-time temporal logic and 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, which verifies the sequence of events with respect to the specified properties [4].

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. 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 [5, 6]. Aspects provide the benefit of good modularity: code simplicity, ease of development and maintenance, and potential for reuse [7]. AspectJ [8] 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.

Since primitive events in PEDL correspond to transfer of control between methods or

assignments to variables, PEDL events represent pointcuts in a program. MEDL properties correspond to safety requirements, or the advice for each pointcut. While Java-MaC provides for runtime verification of stand-alone applications, it requires full access to the source code of the application. Aspect orientation provides a way to weave aspects without having to access the source code, thus providing a black box approach to instrumentation and monitoring.

Our goal is to automatically generate AspectJ aspects from MEDL and PEDL specifications. In order to generate aspects, we define a one-way mapping from MEDL and PEDL grammars to the AspectJ grammar. MEDL and PEDL specifications are used to identify properties to be monitored and instrumentation locations. An aspect is created and is woven into either the source code or the byte code. Depending on the needs of the user, the aspect-enhanced code may monitor and detect violations, it may emit data to the event recognizer, or it may emit an event stream to the runtime checker. This allows the Java-MaC architecture to be used in emerging technologies such as web and grid services.

The MEDL and PEDL files for the railroad crossing example were defined as described in [4]. The PEDL file contained the following events: *startIC* occurs when a train reaches the crossing; *endIC* occurs when the last train passes the crossing; *startGD* occurs when the gate is closed; and *endGD* occurs when the gate starts to rise. The MEDL file contained the properties *IC*, which means a train is crossing, and *GD*, which means a gate is down. These conditions are represented as *Cond IC= [startIC, endIC]* and *Cond GD= [startGD, endGD]*, with the safety condition *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 aspect generation is not yet fully automated. In the near future, we anticipate being able to support the MEDL and PEDL languages in their entirety.

References:

[1] Java MaC, “Run-time Monitoring and Checking (MaC)”. [Online] Available http://www.cis.upenn.edu/~rtg/mac/index.php3, November 23, 2006.

[2] 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.

[3] C. Heitmeyer and D. Mandrioli, “Eds. Formal Methods for Real-Time Systems”. *Number 5 in Trends in Software*. John Wiley & Sons, 1996.

[4] 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.

[5] E. Hilsdale, and J. Hugunin, “Advice Weaving in AspectJ”, *in Proc. Aspect-oriented Software Development 2004*, 2004, pp. 26-35.

[6] Palo Alto Research Center. “The AspectJ Programming Guide”. [Online] Available http://www.eclipse.org/aspectj/doc/released/progguide/index.html, October 1, 2006.

[7] 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.

[8] 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.