Aspect-Oriented Programming Runtime-Enforcement of Temporal Properties in Security-Critical Software

Christian Colombo and Gordon J. Pace

Department of Computer Science, University of Malta
{gordon.pace|ccol002}@um.edu.mt

Abstract. The Aspect-Oriented Programming paradigm has been advocated for modularisation of cross-cutting concerns in large systems. Various applications of this approach have been explored in the literature, one of which is that of runtime-verification based on assertions or temporal properties. Manually weaving temporal properties to ensure correct execution into a large code base is difficult to achieve in a clean, modular fashion, and AOP techniques enable independent specification of the properties to be automatically woven into the code. In this paper, we explore a number of applications of AOP-based runtime-verification with an emphasis on security-critical system development. Apart from weaving properties into existing programs, we show how related techniques can be used to approach security issues separately from the functionality of a module, allowing for better design of the actual system. Also, we explore AOP as a way of automatically ensuring that reusable code in a library is temporally correctly employed. An area in which not much work has yet been done is that of the use of AOP for runtime-verification of real-time properties. In our case studies we explore real-time issues and outline a proposal for automatic translation from real-time properties into code using AOP techniques.

1 Introduction

In designing software systems we would like to separate all different concerns into different modules, making the system easier to understand and maintain, makinf the code more reusable and more manageable for developers [KLM+97,BS06] — all advantages of good modular design. However, a problem sometimes encountered when using commonly used programming paradigms, is that a number of our concerns may not be possible modularised with the rest of the system [KLM+97,KHH+01,MKL97]. For example consider the design of a server application shown in Figure 1.

It is natural to choose to split the server into two main modules: one which validates the incoming connections and one which serves the existing connections. It may also also natural to split further the module serving requests into sub-modules, where each module would be responsible for the servicing of a particular type of requests. One may also envisage a separate module which

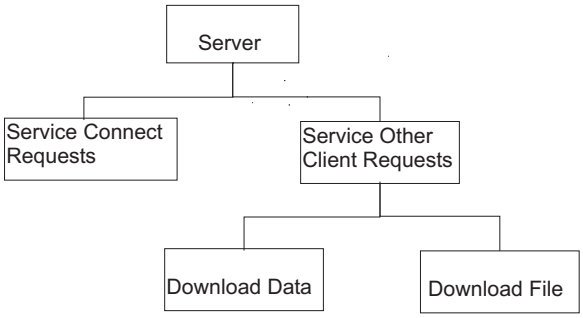

Fig. 1. A server application design.

caters for the accounting of the server application. However, usually, statistics are gathered for accounting from all the modules of the system, spreading out use of the accounting module across the whole code. A more serious concern, is that changes to what information is logged will require maintenance of code across the whole system, and not just the accounting module. Similarly, security concerns usually cut across various of the modules of the system. With security concerns, the situation is worse than with most logging requirements, since one usually wants to identify not single events and function calls, but sequences of events which may originate from separate modules. Hence, one is faced with the same problems of unstructured code. These concerns, called aspects in AOP jargon, are said to *cross-cut* the system's underlying functionality. The motivation behind AOP, lies in providing modular means of introducing such aspects into a system. Subsequently, the aspect code is automatically woven into the other modules at the points at which they cross-cut.

In this paper, we explore applications of AOP-based runtime-verification with an emphasis on security-critical system development. Using a simple server application case study, we show how AOP can be used for explicitly stating the properties of an existing system without modifying the system itself, and running these property monitors automatically together with the system, thus assuring that the properties are adhered to at runtime. Another technique we explore is the separation of security features from the functional core which may assume well-behaved communicating partners. We show how the development of a security-aware system can be split into (i) the development of a naïve system which presumes non-malevolent users; and (ii) the independent identification of security features which are then woven automatically into the functionality of the naïve module, thus allowing for better design of the complete system. Finally, we also explore AOP as a means to automatically ensure that reusable code in a library is employed correctly by having a separate contract specifying the correct use of the code in the library, including temporal dependencies. We start by exploring these different applications looking at temporal dependencies — with properties such as 'method `connect` cannot happen twice in a row with-

out a call to `disconnect` in between'. We then extend this reasoning to deal with real-time properties, such as 'no more than 10 calls to `connect` can appear every minute'. In both cases, we review and propose techniques using which such properties can be automatically incorporated using AOP techniques.

In Section 2 we give a brief review of AOP together with an overview of AspectJ as a prominent AOP technology. Furthermore, we give an overview of runtime verification and various flavours of temporal properties. Thus we explain the motivation behind the proposed AOP approaches. In Section 3 we describe three case studies in AspectJ, illustrating the use of AOP techniques for verifying various temporal properties at runtime. Similarly, in Section 4 we give another three case studies but this time with real-time properties. In Section 5 we give an outline of the proposed framework for specifying real-time properties using two examples from the case studies. Finally, in section 6 we conclude the work and propose future work.

2 Background

2.1 Aspect-Oriented Programming

As already briefly mentioned, AOP enables the programming of cross-cutting features in a system in a modular fashion. The AOP modules can be seen simply as other system modules but whose semantics corresponds not to actual code, but essentially as functions transforming, or modifying the rest of the code.

Different variations to AOP have appeared in the literature, depending on the different level of abstraction at which the aspect code is meant to be inserted in the code. For example some applications may need to work on low level information of method calling [MKL97] while in many other cases a higher level abstraction is more appropriate [SB06,KLM+97].

AspectJ is one of the more popular AOP languages [SB06]. It is an aspect-oriented extension to JAVA [KHH+01] originally developed by Xerox PARC to be a general purpose AOP language [SB06,LK98]. AspectJ provides a number of constructs which allow the developer to specify a variety of joinpoints within the code. For example joinpoints (where aspect code can be inserted) can be specified at the beginning or ending of method calls. Similarly, we can specify joinpoints just before or after a field access in a class. There are also other interesting constructs such as *cflow()*. This particular construct allows the developer to program interesting code such as *do something* when a particular recursive method is called but *not* when it is called within itself [KHH+01,SB06].

Of particular interest is that AspectJ offers two types of crosscutting: *dynamic crosscutting* and *static crosscutting*. While dynamic crosscutting allows the user to alter the execution of a system through the use of aspects, static crosscutting allows the changing of the class definitions themselves [KHH+01].

2.2 Runtime-Verification

Model checking has the power to assert that a property holds along any possible execution path of a system (whatever inputs it receives, and whatever

non-deterministic choices are made inside the system) [UT02,ZKTR07], making it a very desirable objective. However, model checking depends on having a decidable domain, and for the algorithm to be tractable when applied to systems of the magnitude one wants to analyse. Various abstraction and reduction techniques have been proposed to scale up model checking, but full verification of large-scale software systems is still largely unattainable [GH05,ZKTR07,FS01]. In contrast with model checking, in runtime-verification, one checks that a given system property holds along a particular execution path [ZKTR07]. This is particularly useful to ensure that at no time during the execution of the system, are any of the system properties violated. Conversely, it can also identify executions paths taken at runtime, and along which the properties to be verified are not satisfied [ZKTR07]. Essentially, runtime-verification links the abstract specification and the actual concrete implementation [LBAK+98,STY03]. Thus, runtime-verification can be used as a protection from potential faults at runtime, by implementing monitors to react to any property violations encountered [GH05].

In general, a runtime-verification specification consists of (i) identification of the properties which are to be satisfied by the system; (ii) identification of the points during the execution when these properties are to hold; and (iii) identification of actions which should be taken when these properties fail.

Clearly, the logic chosen to write such specifications should enable straightforward expression of such properties, so as to avoid potential errors being introduced in the specification. Furthermore, the modification of the code in the concrete system to insert check for such a specification should be largely (or completely) automated, also to avoid potential errors being introduced in translating the properties into actual monitors. Finally, the properties should be cheap to verify at runtime, avoiding heavy overheads which substantially alter the real-time behaviour of a program. These two constraints limit one's choice of logics, with the latter also being motivation for adopting AOP for automatic insertion of monitors in the code. AOP offers the advantage of specifying all such properties in separate modules (aspects) which contain all the logic concerning these properties [UT02]. Thus one avoids having the properties written as dispersed code throughout the system checking for property violations. Furthermore, it has been observed that other runtime-verification approaches for object-oriented systems, usually break the encapsulation principle from object-oriented programming [Bod05].

2.3 Expressing Temporal Properties

Most of the properties one would like to verify at runtime are temporal properties, in that they specify properties about the order of events in the underlying system. Examples of such properties are 'ensure that initialisation takes place before other methods execute', or 'ensure that no more than three sequential bad logins are allowed'. Using such properties, the developer can reason about the order of and dependencies in the control flow of a system [SB06]. Unlike atemporal, or global assertions, temporal properties are concerned not only with

the current state of the system but also with its history — the sequence of states which led to the current state [SB06]. The logic to be used should thus enable the expression of properties on sequences of states.

Considerable work has appeared exploring the use of Linear Temporal Logic (LTL) as the way to express such temporal constraints for runtime verification [SB06,Bod05,BS06,SH05]. Bodden and Stolz [SB06,Bod05,BS06], use LTL for the specification, and translate it into AOP code expressing the weaving of monitors of the original LTL property into the concrete system. Sammapun and Sokolsky [SS03] use a similar approach but based on the description of the temporal properties through the use of regular expressions. Alternatives to such logics explored in the literature, are the use automata for the expression of temporal constraints — for example alternating finite state automata have been used in [SB06,Dru06,FS01], and timed automata were used in [Bou06,STY03,FH06] to reason about temporal properties. Although most logics can be naturally translated to and from automata, it is interesting to compare and contrast the complexity of expressing typical properties using the two approaches.

Sometimes, the sort of temporal reasoning one requires goes beyond the the ordering of events, adding operators to enable reasoning about the actual timing of the events. Such specifications would enable expressing properties like 'no more than three bad logins can appear in any 30 minute period during the execution of the system', or 'after 30 minutes of inactivity, the server will automatically disconnect the user'. Certain temporal logics allow for the specification of such real-time properties. Adding real-time constraints to an existing system can be extremely complex, and usually it is much simpler to redevelop the system from scratch rather than adding reactivity, even if the underlying functionality is already there. Gal et al. [GSSP02] show how AOP can be used to allow the developers to separate the functional from the real-time concerns thus making code easier to develop and more reusable.

Another consideration in real-time systems is that upon instrumentation of the aspect code, the timing constraints are not violated through the introduction of the property monitors. Furthermore, one other important aspect one should keep in mind when introducing runtime monitors is memory usage. Excessive memory usage by the verification code may cause undesired effects on the system being considered, including timing violations. However, little research has been done on this issue of guaranteeing memory usage and timing performance after the instrumentation of aspect code.

3 Case Studies

In this section, we will present a number of simple case studies to illustrate the use of AOP techniques for security-critical systems. The technology employed for these case studies is AspectJ. The implemented server is a trivial one for illustration purposes, and receives requests from clients and if the requests are valid, these are served. If the server does not have a profile of a client with a particular ip address, then the client must first notify the server and then login using the

password set during the first notification. Once the client has successfully logged in, other services are available. The client can choose from the available services by specifying a request number, where each number corresponds to a particular type of request. Finally, the user can log out. Once logged out, the user only request a login while any other requests will not be granted. This is the basic specification of the server. However, on top of these properties, we would like to specify an extra property for security reasons: *if a client with a certain ip address sequentially makes a number of invalid requests which exceeds a certain limit, then no more request of this client will be considered.* The first two case studies give a description of two possible approaches of using AOP to secure this property.

3.1 Specifying Temporal Security Properties on an Existing System

In this first case study, we assume that the developer of the server has attempted to cater for the specified security property of blocking a user whose ip address has appeared repeatedly issuing invalid requests. The server code handling this condition is shown in Listing 1.1.

Listing 1.1. Client-blocking in the server code

```
1 if (htBlackList.containsKey(c.ip) && ((Integer)htBlackList.
      get(c.ip)).intValue() > limit)
2     {System.out.println("You are blocked");}
3 else
4     {serve(c,reqNum);}
```

Since one would typically not want to look into the server code to verify that it is correctly implemented, we would like to independently implement the property as an aspect, which when woven into the server code will provide runtime verification of the specified property. The purpose of the aspect in this case is to act as a double check that the system is correctly implemented. Typically, in practice, the property would be concisely expressed in a temporal logic and automatically translated into an aspect. One would thus trust the property more than the underlying code.

During implementation, it was noted that if the implemented system was not well structured, then it would have been much more difficult to implement using AOP, since AOP code needs to be injected at specified joinpoints. A frequently used joinpoint is the method call. Therefore, if the system code is not well structured into methods, there would be many less jointpoints available. The main advice which can block a client is shown in Listing 1.2.

Listing 1.2. Client-blocking in AOP

```
1 void around (Client cl, int r):(execution(* Server.serve(..))
      && args(cl,r)) {
2     if (ht.containsKey(cl.ip)) {
3         if (((Integer)ht.get(cl.ip)).intValue() > limit)
4             System.out.println(cl+" :: Property violation
                  detected. User will be blocked.");
```

```
5           else proceed(cl,r);
6       } else proceed(cl,r);
7   }
```

The *around* advice employed in this case is a construct which allows the developer to receive control before the actual method is executed, thus enabling him or her to do anything before and/or after the actual method is called. This includes changing the parameters by which it is called and even stopping it from being called at all. In this example, the method *serve* is only instructed to go ahead (using the *proceed* construct) if the client is not blocked.

3.2 Specifying Temporal Security Properties as a Separate Module

A different approach from the previous scenario would be to separate the underlying functionality in the case of non-malicious users completely from the code handling security concerns. The problem in doing so using typical modularity offered by programming paradigms other than AOP is that the security features and the code expressing the 'correct' functionality are tightly knit together, passing control back and forth across the code. In the approach we present here, the implementation of the security features appear as AOP code, reaching into the design of the naïve server and weaving the features. Modularity is achieved since through AOP directives we can actually modify, as opposed to simply use, other code. The blocking mechanism in this case is not simply a *double check,* but the *actual check* itself.

The advantage of such a configuration is that the server code is left totally clean without any additional checks for ensuring security properties. Furthermore, all the security related code (part of which is shown in Listing 1.2) is now in one module (aspect) while in the previous case, there were two classes (apart from the AOP aspect) which contained code related to the security checks. Adding additional properties requires adding them to one module without having to understand the potentially complex logic of the server code riddled with runtime checks. Furthermore, one could imagine scenarios where the properties may be applied to different vanilla-servers, thus enabling reuse of the runtime properties.

During the design of these case studies it was noted that the decision of which properties are part of the actual server and which other properties are to be considered as extraneous, is totally arbitrary (for the developer to decide). For example, we could have decided that checking that the login is correct is not a part of the server's functionality and implemented it as a separate aspect.

3.3 Specifying Temporal Security Properties on a Code Library

In the third case study we will present the use of AOP for runtime-verification in a library to stop (or warn) users of library in the case of wrong usage. Such inappropriate use of the code may lead to undesirable faults in the user's applications. We use AOP techniques to weave temporal properties inside library

code, to handle incorrect usage going beyond traditional pre- and post-condition checking. The advantage of this approach is that the properties specified will not only hold for the currently implemented methods in the library, but also for extensions which can possibly be added in the future. In the simple example suggested here, we consider a scenario where the library should be initialised before any other method is used. Furthermore, once initialised, the library can be reset to return the library to an uninitialised state. The code which blocks any method call before initialisation is shown in Listing 1.3.

Listing 1.3. Library checking advice

```
1  Object around():(execution (* Library.*(..)) && !execution(*
       Library.initialization(..))) {
2      if (initialized)
3          return proceed();
4      else
5          return "LIBRARY: Library must be initialized.";
6      }
```

Using the "*" wildcard, we have managed to check for initialisation before the execution of any possible method apart from the one whose name is *initialization*. One should note the efficiency of implementing such logic in a few line of code rather than inserting a condition at the start of all the methods in the library. Furthermore, adding further methods to the library does not necessitate any modifications to the code handling the property.

4 Case Studies with Real-Time Security Properties

In this section we will reconsider the previous examples with additional real-time constraints.

4.1 Specifying Real-Time Security Properties on an Existing System

In the case of the server, the added constraint was that after a certain period of time, a blocked client will now be unblocked once more and allowed to make requests. Therefore, the time at which a client was blocked is stored in a hash table. Then, each time a request is received from a blocked client, the time elapsed (since the denial of service) is checked and is unblocked if the time limit was exceeded. Given that our server implementation is very simply it was relatively easy to add the necessary logic for the newly introduced real-time constraint. However, in a real scenario, the consequence of adding new code in the actual server implementation may prove to be much more cumbersome. The AOP aspect was also updated to cater for the new real-time constraint. This was done by storing the time at which each client was blocked from the system. Subsequently, when a client sends a request, the current time is compared to the stored time so that if enough time has elapsed, the client is unblocked. This is shown in Listing 1.4.

Listing 1.4. AOP checking advice with real-time constraint

```
1  pointcut request(Client cl, int r):execution(* Server.serve
       (..)) && args(cl,r);
2  void around (Client cl, int r) :request(cl,r) {
3      if (htBlacklist.containsKey(cl.ip)) {
4          long currentTime = System.currentTimeMillis();
5          if (currentTime - ((Long)htBlacklist.get(cl.ip)).get
               () < release)
6              System.out.println(cl+" :: You are BLOCKED");
7          else {
8              System.out.println(cl+" :: You are UNBLOCKED");
9              htBlacklist.remove(cl.ip);
10             if (ht.containsKey(cl.ip)) {
11                 ht.remove(cl.ip);
12                 ht.put(cl.ip, new Integer(0));
13             }
14             proceed(cl,r);
15         }
16     }
17     else proceed(cl,r);
18 }
```

4.2 Specifying Real-Time Security Properties as a Separate Module

Adding the additional real-time constraint to the system with security as a separate module proved to be much easier than the previous scenario. Basically, the server code remained intact while the extra logic (the same as that in Listing 1.4) was simply added in the single module (aspect) which handles the security.

4.3 Specifying Real-Time Security Properties on a Code Library

The added real-time constraint in the library scenario was that after a certain time period the initialised library automatically returns to an uninitialised state. The implementation was done by storing the time at which the library was initialised and adding an extra check (Line 3 in Listing 1.5) to ensure that the initialisation is still valid when a user requires a method execution from the library.

Listing 1.5. Library checking advice including real-time constraint

```
1  Object around():(execution (* Library.*(..)) && !execution(*
       Library.initialization(..))) {
2      long currentClock = System.currentTimeMillis();
3      if (initialized && currentClock-clock < limit)
4          return proceed();
5      else if (initialized) {
6          initialized = false;
7          return "LIBRARY: Time expired";
```

```
 8       }
 9       else
10           return "LIBRARY: First you must initialize";
11   }
```

5 Proposed Framework for Real-Time Properties

Although demonstrated in the case studies, the use of AOP may seem straightforward, in other cases it may be much more difficult to translate from the conceptual security property into the actual aspect code. We are currently exploring the use of different formal techniques for specifying real-time security properties in a natural way. The aim is to explore the use of decidable temporal notations, such as timed automata to describe properties which would then be automatically compiled to AOP code.

Timed automata, use timers, which can be used to specify when particular actions which will be performed if the system is within a certain state. To illustrate the approach we are exploring, we will present the real-time properties presented in the case studies and show how they would be represented using timed automata.

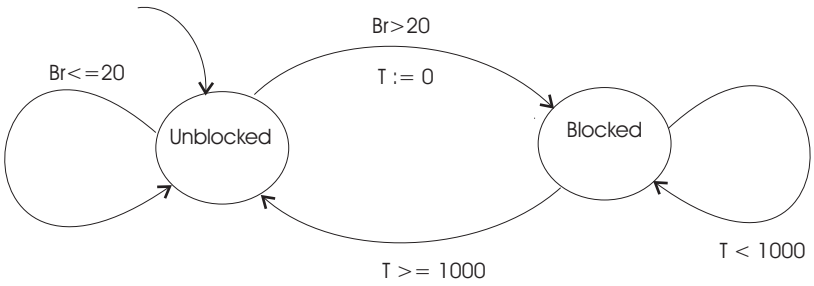

Fig. 2. A timed automaton representing the real-time property implemented in the server case study.

The server property is shown in Figure 2. The process of keeping count of the number of bad requests per client is hidden and the condition to enter into blocking mode is represented by the boolean input *Br*. Once the system enters into blocking mode, the timer is reset to zero (represented by *T:=0*). Eventually, the system only returns to unblocked mode when the timer reaches the desired amount (in this case 1000).

Figure 3 depicts the real-time property implemented in the library case study. In this example, the timer is reset when the initial input is set to true. Subsequently, when the timer reaches the threshold (also set to 1000), the system reverts back to an uninitialised state. However, this time, the system will also

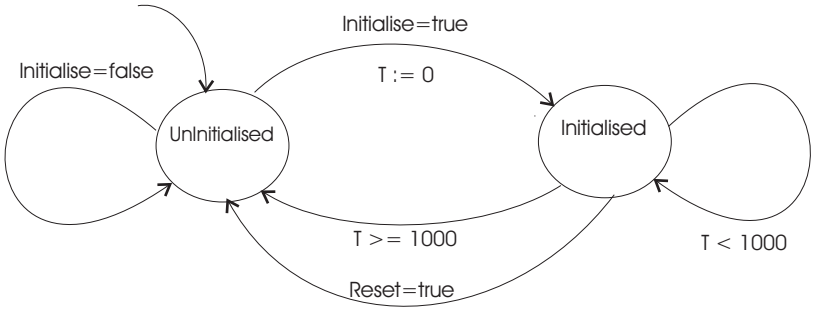

Fig. 3. A timed automaton representing the real-time property implemented in the library case study.

return to an uninitialised state if the reset input becomes true while in the initialised state.

6 Conclusion and Future Work

In this paper we have presented ways of employing AOP for ensuring temporal security properties in a modular and succinct manner. Furthermore, we have shown how the same framework can be further modified to additionally include real-time properties. To allow for such real-time properties to be specified more easily we proposed a framework based on time automata.

We are currently exploring the use of variants of symbolic timed automata for the expression of real-time properties which can be automatically translated into aspects. In particular, we would like to explore their use, on real-time security properties in security-critical systems. One interesting challenge to address, is memory usage of the monitoring code. One approach we believe could be fruitful, is the use of techniques developed for reactive systems such as Lustre [HCRP91].

References

[Bod05] Eric Bodden. Efficient and expressive runtime verification for Java. Grand Finals of the ACM Student Research Competition, 2004/2005.

[Bou06] Patricia Bouyer. Weighted timed automata: Model-checking and games. *Electr. Notes Theor. Comput. Sci.*, 158:3–17, 2006.

[BS06] Eric Bodden and Volker Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In *Software Composition*, pages 147–162, 2006.

[Dru06] D. Drusinsky. On-line monitoring of metric temporal logic with time-series constraints using alternating finite automata. *J. UCS*, 12(5):482–498, 2006.

[FH06] P. Fradet and S. Hong Tuan Ha. Systèmes de gestion de ressource et aspects de disponibilité. *Revue francophone l'objet*, 12(2), 2006.

[FS01] B. Finkbeiner and H. Sipma. Checking finite traces using alternating automata. In *Proceedings of the First International Workshop on Runtime*

Verification, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[GH05] Allen Goldberg and Klaus Havelund. Automated runtime verification with eagle. In *MSVVEIS*, 2005.

[GSSP02] Andreas Gal, Olaf Spinczyk, and Wolfgang Schrder-Preikschat. On aspect-orientation in distributed real-time dependable systems. In *WORDS*, pages 261–270, 2002.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming, 11th European Conference, LNCS 1241*, 1997.

[LBAK+98] I. Lee, H. Ben-Abdallah, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. A monitoring and checking framework for run-time correctness assurance. In *Korea-U.S. Technical Conference on Strategic Technologies*, 1998.

[LK98] C. Lopes and G. Kiczales. Recent developments in AspectJ. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP 98)*, 1998.

[MKL97] A. Mendhekar, G. Kiczales, and J. Lamping. Rg: A case-study for aspect-oriented programming. Technical Report SPL97-009P9710044, Xerox PARC, 1997.

[SB06] Volker Stolz and Eric Bodden. Temporal assertions using aspectj. *Electr. Notes Theor. Comput. Sci.*, 144(4):109–124, 2006.

[SH05] Volker Stolz and Frank Huch. Runtime verification of concurrent Haskell programs. *Electr. Notes Theor. Comput. Sci.*, 113:201–216, 2005.

[SS03] Usa Sammapun and Oleg Sokolsky. Regular expressions for run-time verification. In *Proceedings of the 1st International Workshop on Automated Technology for Verification and Analysis (ATVA'03)*, 2003.

[STY03] J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. *Proceedings of the IEEE*, 91:100–111, 2003.

[UT02] Naoyasu Ubayashi and Tetsuo Tamai. Aspect-oriented programming with model checking. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 148–154, New York, NY, USA, 2002. ACM Press.

[ZKTR07] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin Rinard. Runtime checking for program verification. In *Workshop on Workshop on Runtime Verification (collocated with AOSD)*, 2007.