Verify Your Runs

Klaus Havelund and Allen Goldberg

Kestrel Technology, Palo Alto, California, USA
{havelund,goldberg}@kestreltechnology.com
http://www.kestreltechnology.com

1 Introduction

A program verifier determines whether a program satisfies a specification. Ideally verification is achieved by static analysis without executing the code. However, program verification is unsolvable in general. The interactive approach, for example with a human guiding a theorem prover, does not in practice scale to large software systems. Some restricted kinds of specifications can, however, be checked automatically, for example type definitions. Also static analysis of properties such as un-initialized variables, null-pointer de-referencing, and array-bound violations scales to production programs on the order of hundreds of thousands of lines of code. Even concurrency-related problems such as data races and deadlocks can to some extent be checked statically, although often resulting in false positives. However, going beyond these simple properties to arbitrarily complex behavior specification and scaling to ever-growing production program size is undoubtedly a challenge, and in our opinion we cannot expect regular economic use of program verification of arbitrary properties to be fully achieved within the 15 year time horizon of the challenge.

Hence, we will probably have to accept that parts of the verification task will remain as proof obligations. It is reasonable to not throw such proof obligations away, but to monitor them during program testing, or in the operations phase. In the latter case, one can program reactions to property violations to achieve some form of fault protection. We call the scientific discipline that studies the monitoring of properties during program execution *runtime verification* [1]. Much work has been done in this area within recent years.

In this paper we shall outline and classify some current approaches to runtime verification and describe our contributions. We shall describe how we intend to further contribute to this work in the framework of the Grand Verification Challenge. The paper does not address the topic of test *case generation* although runtime verification is a part of this subject. That is, an effectful test case generation framework needs to support the generation of test cases, where a test case consists of inputs to the program together with an oracle that will inspect the output of the program (including inspection of its internal behavior) when executed on that input. Generation of the oracle is the runtime verification part. We believe that runtime verification is a rich subject on its own.

2 Specification-Based Runtime Verification

Specification-based runtime verification consists of monitoring a program's execution against a *user-provided specification* of intended program behavior. The many approaches to program specification logics have lead to differing styles of runtime verification. One can consider a spectrum of monitoring approaches, ranging from monitoring of predicate assertions stating properties about a single state at a single program location, to monitoring of temporal assertions stating properties about temporally separated states at multiple program locations identified by automated program instrumentation. We shall discuss the techniques along four dimensions:

- *Location quantification:* whether the logic allows to quantify over locations in the program to be monitored. Monitors evaluate when certain program locations are reached during program execution. If the monitoring code is executing *in-line*, these locations will contain the monitoring code itself. If monitoring is *off-line*, the locations will contain event generators, that will send events to the monitors that run in some more or less loose form of synchronization with the code. Locations can either be specified individually, by identifying each of them explicitly, or they can be *quantified* over, as in Aspect Oriented Programming, covering many locations with one declaration, for example "before every call of any method defined in class C, evaluate monitor M".
- *Temporal quantification:* whether the logic allows quantification over time points. For example, whether one can express properties of the form: "whenever a call of method **close** occurs then in the past there has been a call of the **open** method". Some temporal logics only allow to state ordering relationships, while others go further and allow to state relative or absolute time values.
- *Data quantification:* whether the logic allows binding and referral (forward or backward in time) to values across states. For example, "whenever a call of method **close**(f) occurs, with a file argument f, then in the past there has been a call of **open**(f) of the same file f." Obviously, data quantification presumes temporal quantification.
- *Abstract data specification:* whether abstract states mapping variable names to values can be defined together with an abstraction that relates concrete program states to abstract specification states.

In the following we shall classify a collection of monitoring approaches along these dimensions.

Assertions Runtime checks are assertions inserted at specific locations in the code. Assertions were introduced in Java 1.4 such that the programmer can write assertions of the form **assert** ψ for a Java predicate ψ, at explicit program locations. Assertions do not directly support location, temporal or data quantification, nor abstract data specification.

Pre-Post Conditions Pre-post conditions is an extension of the assert statement, where the programmer explicitly indicates where checks should be performed, namely before and after method calls, hence not supporting location quantification. However, a post condition typically relates the value of variables at the start and the end of the method. Thus this is a restricted form of temporal and data quantification. The Eiffel language [25] has long embodied this idea, and recently so has JML [41], the Java Modeling Language. In Eiffel there is no provision for abstract data specification, while in JML there is. The Larch Shared Language approach [44] supports abstract (axiomatic) data specification in combination with pre-post conditions.

Invariants Invariants, as found in for example Eiffel and JML, express properties about a single state, and are required to hold at *all* locations where data consistency can be expected, specifically at the completion of method calls, and at the limit after every variable update. Hence this is an example of a logic supporting location quantification. Since an invariant asserts a property of just the current state it does not support temporal or data quantification.

State Machine Notations and Process Algebras In state machines/automata and process algebras, the specification is an *abstract program*, and runtime verification dynamically checks that the executing program is a *refinement* of the abstract program. That is, the instrumented program locations correspond to abstract program states, and monitoring checks the required state sequencing and that each concrete program state satisfies (via an abstraction function) the properties of a corresponding abstract state. This form of specification supports temporal quantification. The Jass system [40] monitors a combination of JML and CSP process algebra. Alternating automata, supporting AND as well as OR states, have shown to be particularly convenient for monitoring logics as demonstrated in [22, 28, 27]. Also state charts [32] offer this combination of AND and OR states. An example illustrating the use of state machines for monitoring is the TLChart system [24], that monitors a combination of temporal logic and state machines. An extension of simple state machines are timed automata, where time constraints can be put on states (one can only be in a state for a certain time period) and on transitions. One such system for runtime verification of timed automata is T-UPPAAL [56] and another similar system is described in [13]. T-UPPAAL also generates test cases. All the systems mentioned above monitor finite traces against finite trace automata. In [20] is described a technique for monitoring against Omega automata: automata that normally accept infinite traces. This is specifically useful for monitoring automata generated from specifications originally targeted for model checkers such as SPIN [54]. In [30] is described an algorithm for synthesizing finite trace monitoring algorithms from LTL specifications, inspired by similar algorithms used for synthesizing infinite trace Omega automata from LTL specifications.

Temporal Assertions While automata and process algebras are operational in nature, temporal logics are declarative. *Temporal logics* have operators that re-

late arbitrary states, and hence support full temporal quantification, and can in many cases allow more succinct specifications. Pre-post conditions support a simple form of temporal quantification by relating two states (the pre-state and the post-state). In the commercial Temporal Rover system [22], one writes past time and future time temporal logic formulas at specific program locations, that get evaluated whenever that program location is reached. This tool hence supports temporal quantification but not location quantification. The MaC system [43] supports temporal past time assertions and location quantification by allowing instrumentation of method calls and variable updates. MaC also allows abstract data specifications referenced as the propositions of the temporal logic. An interesting logic is the future time temporal logic PSL [46] adopted by the hardware industry. In metric temporal logics one can state properties about time. Several such systems have been developed, for example [22, 57]. Regular expressions, and extended regular expressions allowing negation, appear to be very useful for writing certain properties that in temporal logic would become more complicated to state. Such a system is described in [51]. A generalization of metric logics are data logics supporting data quantification, where one can reason about data values existing at different time points. Such systems are described in [23, 27, 21]. Temporal logics are often mapped to automata, although other interpretations are possible, such as for example described in [49, 36], where rewriting is used to interpret temporal logic for monitoring.

General Purpose Specification Languages A monitoring language may be a complete formal specification language, in the style of ASML [7], Maude [17], PVS [47], VDM [60], RAISE [48] or Specware [53]. This is the approach taken at Microsoft where ASML (Abstract State Machine Language) [7] is used for runtime verification as part of a general test case generation framework. Clearly such an approach supports abstract data specification. These full specification languages usually have executable subsets which resemble a programming language, be it functional or state-based. This observation can be exploited by having the specification language be an extension of the programming language, an approach taken in Spec# [9], Microsoft's extension of the work in [7].

3 Predictive Runtime Verification

As with testing, the effectiveness of runtime verification depends on the choice of test suite. For concurrent systems this becomes even more serious because this is compounded by the many possible execution paths of a non-deterministic program. This raises the question of whether there are properties that can be checked on one or a small number of execution traces and still identify bugs with high probability (if such exist). The answer is affirmative due to recent work on what we call *predictive runtime verification*.

In predictive runtime verification a property P to be monitored is replaced with a stronger property Q, i.e. for all inputs x, $Q(x) \rightarrow P(x)$. Furthermore if $\forall x P(x)$ then $Q(x)$ for most x (few false positives) but if $\exists x \neg P(x)$ then $\neg Q(x)$

for most x (good detection). It turns out that for certain problems finding such Q is possible.

One of the earliest successes was the Eraser algorithm [50] for detecting data races, that was implemented in Compaq's Visual Threads tool [33]. This algorithm checks a single execution trace in order to determine whether there are any *potentials* for data races: the situation where two threads access a shared variable simultaneously. This work has later been extended to cover other forms of data races, such as higher level data races [5] and atomicity violations [6, 29, 63]. Also deadlocks of the dining philosopher format can be checked in this manner [15]. A generalized predictive analysis framework is presented in [52]. In most of the above mentioned systems, the properties are programmed directly as algorithms in a traditional programming language. Attempts have, however, been made to express the properties in logic [10]. These are often data oriented properties that are best expressed in a monitoring logic appropriate for expressing data quantification and location quantification.

Concurrent target systems may be modified by inclusion of wait statements or modifications to schedulers, so that a fuller range of non-deterministic behaviors are exhibited during testing. Such modifications can be combined with predictive analysis. This is discussed in the overview paper [26] and in [14].

4 Instrumentation

Instrumentation is the modification of the target system with additional code that informs the monitor of events and data values relevant to the monitored properties, such as the taking of a lock, the entry into a method, or the update of a variable. This can be achieved through source code instrumentation, for example using Aspect Oriented Programming (AOP) as supported by AspectJ [8]; through byte-code instrumentation, BCEL [12] being an example byte-code instrumentation tool; or through object code instrumentation, with Valgrind [58] being an example. The Java-MOP system described in [16] is a generalized framework for instrumenting Java programs specifically for runtime verification. Instrumentation can, however, also be done through debugging interfaces, modification of the runtime system or virtual machine, or through operating system or middleware services. In our work we have used byte-code instrumentation and aspect oriented programming to instrument code.

It is worth noting that runtime verification techniques are starting to appear within the aspect oriented programming community. In a traditional AOP language such as AspectJ [42], an aspect specifies augmentations/modifications to a program, that may add functionality or specify a correctness property and appropriate actions to be executed when the property is violated. Traditionally actions are weaved into the program at program points, specified by so called pointcuts. A recent trend is to augment the pointcut language to include predicates on the execution trace. Solutions have been offered for augmenting AOP with regular expressions [2], future time linear temporal logic [55, 19], state machines [59], and grammars [62].

Naive instrumentation can cause significant degradation of performance and is a significant concern for most systems, especially real time systems. Static analysis can be used as a technique for optimizing runtime monitoring. This is a dual but equivalent view to that presented in the introduction, namely that runtime monitoring is used to verify residual properties that remain unverified by (static) program verification.

5 Our Previous Work

In this section we briefly outline our own and close colleague's work in runtime verification. Some of our early work [34] was done in predictive runtime verification of concurrent Java programs and resulted in a tool for performing predictive deadlock and Eraser-like data race analysis on Java programs, guiding the Java PathFinder (JPF) model checker [61] to confirm the warnings discovered by the much faster predicative analysis. Instrumentation was done by modifying the Java Virtual Machine of JPF. The work on predictive runtime verification was later re-implemented and elaborated in the Java PathExplorer (JPaX) tool [37, 35]. Specifically the deadlock analysis algorithm was improved to yield fewer false warnings [15]. In [14] is described an approach where such deadlock warnings are fed into a testing framework, where detected potential deadlock cycles are used to control the execution of the program in an attempt to confirm the deadlocks. Other recent results on predictive runtime verification includes work that goes beyond low-level data races on single variables, and includes detection of high-level data races on collections of variables [5], and detection of out-dated copies of shared variables [6]. JPaX also supports specification-based runtime verification. The Maude rewriting system [17] is used to define new logics [49, 36]. This has proved extremely elegant since Maude is well suited for defining the syntax and semantics of a logic. In [38, 39] we describe how to synthesize very efficient algorithms based on dynamic programming for monitoring past time logic.

In more recent work we decided to develop a runtime verification framework for Java *in Java*. Eagle [11] is a powerful temporal kernel language supporting temporal quantification and capable of modeling all of the temporal logics and most of the specification paradigms mentioned in this overview. Eagle is an extension of propositional logic with three temporal kernel operators, recursion, and parameterization over formulas in the logic as well as over data values. Formula parameterization allows the user to define new temporal combinators, and hence new temporal logics. The language therefore directly supports the definition of new specification patterns of the kind illustrated in [45]. Data parameterization allows to define properties relating data values from different points in time, hence supporting data quantification. Due to these constructs Eagle can define various forms of past and future time linear temporal logics, real-time logics, interval logics, extended regular expressions and state machines. Eagle furthermore supports abstract data specification in that formulas are interpreted on an abstract state defined as a Java class, and referred to as the

Eagle state. In principle there is a stratification of the propositional language and the logic proper so that Java may be replaced by a high-level specification language. The user must define an abstraction mapping from concrete program states to abstract Eagle states. At each instrumentation location in the monitored target system, a method representing the abstraction function is called to update the Eagle state. Noting that Eagle supports the definition of state machines, we see that Eagle hence supports both data refinement and control refinement. A recent extension of Eagle supports automated program instrumentation [19], hence location quantification, using the aspect oriented programming tool AspectJ [8]. In previous work we developed the jSpy tool [31], which instruments Java byte-code. A jSpy instrumentation specification consists of a set of rules, each of which consists of a condition on byte-code and an instrumentation action stating what to report when byte-codes satisfying the condition are executed. The reported events are then picked up by the monitors that in turn check for various user provided properties. Eagle has been used within a test-case framework as described in [3, 4].

6 Future Work

As a scientific discipline specification-based runtime verification does not face the same difficult problems as, say, model checking or theorem proving, and is likely closer to become part of practical software development environments. However, the discipline faces unsolved problems concerned with choice of specification notations, monitoring algorithms, code instrumentation, as well as social issues such as the usual resistance amongst software developers to write formal specifications in addition to the code itself. We feel that predictive runtime verification should be part of any development system since it is very effective, fully automated, requires no specifications, and essentially imposes only minor cost to the programmer. The challenge is to identify other problems that lend themselves to this form of analysis. Concerning specification-based runtime verification, choosing the right specification formalism is critical to the success of the approach. The formalism must be simple, yet powerful, and/or, it could be an already accepted notation, such as UML. We will continue experimenting with Eagle, but we will also investigate other formalisms in order to achieve the optimal balance between simplicity, efficiency and effectiveness. Amongst work not mentioned is that on generating specifications from runs [18]. We intend to extend our work in this direction.

References

1. *1st – 5th Workshops on Runtime Verification (RV'01 - RV'05), 2001–2005,* `http://www.runtime-verification.org`, volume 55(2), 70(4), 89(2), 113, to be published, of *ENTCS*. Elsevier Science Direct.
2. C. Allan, P. Avgustinov, S. Kuzins, O. de Moor, D. Sereni, G. Sittamplan, J. Tibble, A. S. Christensen, L. Hendren, and O. Lhoták. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA'05*, 2005.

3. C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining Test-Case Generation and Runtime Verification. *Theoretical Computer Science*, 336(2–3):209–234, May 2005. Extended version of [4].
4. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, and W. Visser. Experiments with Test Case Generation and Runtime Analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines (ASM'03)*, volume 2589 of *LNCS*, pages 87–107. Springer, March 2003.
5. C. Artho, K. Havelund, and A. Biere. High-Level Data Races. *Software Testing, Verification and Reliability*, 13(4), 2004.
6. C. Artho, K. Havelund, and A. Biere. Using Block-Local Atomicity to Detect Stale-Value Concurrency Errors. In *2nd International Symposium on Automated Technology for Verification and Analysis, Taiwan*, October–November 2004.
7. ASML. `http://research.microsoft.com/fse/asml`.
8. AspectJ. `http://eclipse.org/aspectj`.
9. M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS'04*, volume 3362 of *LNCS*, Marseille, France, March 2004. Springer.
10. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program Monitoring with LTL in Eagle. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD'04)*, April 2004. Santa Fee, New Mexico, USA.
11. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Fifth International VMCAI conference (VMCAI'04)*, volume 2937 of *LNCS*. Springer, January 2004.
12. BCEL. `http://jakarta.apache.org/bcel`.
13. S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis. Testing Conformance of Real-Time Applications by Automatic Generation of Observers. In *Proceedings of the 4th International Workshop on Runtime Verification (RV'04)* [1], pages 19–38.
14. S. Bensalem, J.-C. Fernandez, K. Havelund, and L. Mounier. Confirmation of Deadlock Potentials Detected by Runtime Analysis. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD'06), Portland, Maine, USA*, July 2006.
15. S. Bensalem and K. Havelund. Scalable Dynamic Deadlock Analysis of Multi-Threaded Programs. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD - 3), IBM Verification Conference, Haifa, Israel*, November 2005. To be published in LNCS.
16. F. Chen, M. D'Amorim, and G. Roşu. Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP. In *Proceedings of the 5th International Workshop on Runtime Verification (RV'05)* [1].
17. M. Clavel, F. J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude System. In Paliath Narendran and Michaël Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag. System Description.
18. Daikon. `http://pag.csail.mit.edu/daikon`.
19. M. D'Amorim and K. Havelund. Runtime Verification for Java. In *Workshop on Dynamic Program Analysis (WODA'05)*, March 2005.
20. M. D'Amorim and G. Rosu. Efficient Monitoring of Omega-Languages. In *CAV'05*, LNCS. Springer-Verlag, 2005.

21. B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime Monitoring of Synchronous Systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174, 2005.
22. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
23. D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *CAV'03*, volume 2725 of *LNCS*, pages 114–118. Springer-Verlag, 2003.
24. D. Drusinsky. Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions. In *Proceedings of the 4th International Workshop on Runtime Verification (RV'04)* [1], pages 2–18.
25. Eiffel. `http://www.eiffel.com`.
26. Y. Eytani, K. Havelund, S. Stoller, and S. Ur. Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs. *Concurrency and Computation: Practice and Experience*, 2005. To appear.
27. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting Statistics over Runtime Executions. In *Proceedings of the 2nd International Workshop on Runtime Verification (RV'02)* [1], pages 36–55.
28. B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1], pages 44–60.
29. C. Flanagan and S. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. *SIGPLAN Not.*, 39(1):256–267, 2004.
30. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. ENTCS, 2001. Coronado Island, California.
31. A. Goldberg and K. Havelund. Instrumentation of Java Bytecode for Runtime Analysis. In *Fifth ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP'03)*, July 2003. Darmstadt, Germany.
32. D. Harel. Statecharts: A Visual Formalism For Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
33. J. Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 331–342. Springer, 2000.
34. K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.
35. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1], pages 97–114.
36. K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
37. K. Havelund and G. Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, 24(2), March 2004.
38. K. Havelund and G. Roşu. Efficient Monitoring of Safety Properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.

39. K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002. Best paper award.
40. Jass. `http://csd.informatik.uni-oldenburg.de/~jass`.
41. JML. `http://www.cs.iastate.edu/~leavens/JML`.
42. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *European Conference on Object-oriented Programming, volume 2072 of Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.
43. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1].
44. Larch. `http://www.cs.iastate.edu/larch-faq-webboy.html`.
45. Patterns. `http://patterns.projects.cis.ksu.edu`.
46. PSL/Sugar. `http://www.pslsugar.org`.
47. PVS. `http://pvs.csl.sri.com`.
48. RAISE. `http://spd-web.terma.com/Projects/RAISE`.
49. G. Roşu and K. Havelund. A Rewriting-based Approach to Trace Analysis. *Automated Software Engineering*, 12(2):151–197, 2005.
50. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
51. K. Sen and G. Roşu. Generating Optimal Monitors for Extended Regular Expressions. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV'03)* [1], pages 162–181.
52. K. Sen, G. Roşu, and G. Agha. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In M. Steffen and G. Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005*, volume 3535 of *LNCS*, Athens, Greece, June 2005. Springer.
53. Specware. `http://www.specware.org`.
54. SPIN. `http://spinroot.com`.
55. V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Fifth Workshop on Runtime Verification (RV05), Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers*, 2005.
56. T-UPPAAL. `http://www.cs.aau.dk/~marius/tuppaal`.
57. P. Thati and G. Rosu. Monitoring Algorithms for Metric Temporal Logic Specifications. In *Proceedings of the 4th International Workshop on Runtime Verification (RV'04)* [1], pages 131–147.
58. Valgrind. `http://valgrind.org`.
59. W. Vanderperren, D. Suvé, M. Augustina Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. In *Workshop on Software Composition, ETAPS 2005*, 2005.
60. VDM. `http://www.csr.ncl.ac.uk/vdm`.
61. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'00: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.
62. R.J. Walker and K. Viggers. Implementing Protocols via Declarative Event Patterns. In R.N. Taylor and M.B. Dwyer, editors, *12th International Symposium on the Foundations of Software Engineering. ACM*, 2004.
63. L. Wang and S. Stoller. Run-Time Analysis for Atomicity. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV'03)* [1].