Runtime Verification - 17 Years Later

Klaus Havelund*[1] and Grigore Roşu[2]

[1] Jet Propulsion Laboratory, California Institute of Technology
[2] University of Illinois at Urbana-Champaign

Abstract. Runtime verification is the discipline of analyzing program executions using rigorous methods. The discipline covers such topics as specification-based monitoring, where single executions are checked against formal specifications; predictive runtime analysis, where properties about a system are predicted/inferred from single (good) executions; specification mining from execution traces; visualization of execution traces; and to be fully general: computation of any interesting information from execution traces. Finally, runtime verification also includes fault protection, where monitors actively protect a running system against errors. The paper is written as a response to the 'Test of Time Award' attributed to the authors for their 2001 paper [45]. The present paper provides a brief overview of what lead to that paper, what has happened since, and some perspectives on the future of the field.

1 Introduction

Runtime verification (RV) [39, 55, 10, 26] has emerged as a field of computer science within the last couple of decades. RV is concerned with the rigorous monitoring and analysis of software and hardware system executions. The field, or parts of it, can be encountered under several other names, including, e.g., runtime checking, monitoring, dynamic analysis, and runtime analysis. Since only single executions are analyzed, RV scales well compared to more comprehensive formal methods, but of course at the cost of coverage. Nonetheless, RV can be useful due to the rigorous methods involved. Conferences and workshops are now focusing specifically on this subject, including the Runtime Verification conference, which was initiated by the authors in 2001 as a workshop and became a conference in 2010, and runtime verification is now also often listed as a subject of interest in other conference calls for papers.

The paper is written as a response to the 'Test of Time Award' attributed to the authors for the 2001 paper [45] (*Monitoring Java Programs with Java PathExplorer*), presented 17 years ago (at the time of writing) at the first Runtime Verification workshop (RV'01) in Paris, July 23, 2001.

This paper reports on our own RV work, with some references to related work that specifically inspired us, and discusses the lessons learned and our

* The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

perspective on the future of this field. Note that we do not try to identify all literature that inspired us. That task would be impossible. Previous publications of ours [26, 42, 44] have provided more technical tutorial-like presentations of the field. This paper rather offers information about the motivations for our work and philosophical considerations. As such this paper is closer in spirit to the longer paper [43]. It should be mentioned that most of the works over time have been done in collaboration with other people and inspired/initiated/driven by other people. We have just been lucky to be in the midst of all this work.

The paper is organized according to the time line of events, first leading up to [45], then the work described in that paper, the work that followed, and finally some thoughts on the future of this field.

2 In the Beginning

The initial interest of the first author in formal methods stems from his involvement in the design of the RAISE specification language RSL [30], during the period 1984-1991, and even with earlier work in the early 1980'ies on developing a parser and type checker for its predecessor VDM [14, 15, 28]. These are so-called wide-spectrum specification languages permitting formal specification at a high level, and "programming" at a low level, all within the same language, supported by a formal refinement relation between the different levels. These languages were impressively ahead of their time if one looks at these from a programming language perspective. For example, VDM^{++} has many similarities with today's SCALA programming language.

However, these languages were fundamentally still specification languages, and not programming languages, in spite of the fact that these languages have a lot in common with modern high-level programming languages, such as e.g. ML. The thought therefore was: why not benefit from the evolution of modern high-level programming languages and focus on verification of such? This was the first step: the focus on programs rather than models. This lead to the work [34] of the first author on attempting to develop a specification language for an actual programming language, namely CONCURRENT ML (CML), an extension of Milner's ML with concurrency.

Later work with the very impressive PVS theorem prover [35] helped realize that theorem proving is hard after all, and that some form of more automated reasoning on programs would be useful as a less perfect alternative. Hence, thus far the realization was that *automated* verification of *programs* was a desirable objective. Note that at the time the main focus in the formal methods community was on models, not programs.

The next big move was the development of the JAVA PATHFINDER (JPF), a JAVA model checker, first as a translator from JAVA to the PROMELA modeling language of the SPIN model checker [41] (often referred to as JPF1), and later as a byte code model checker [50] (occasionally referred to as JPF2). The goal of this work was to explore how far model checking could be taken wrt. real code verification, either using JAVA as just a better modeling language, or, in the

extreme case, for model checking real programs. A sub-objective was to explore the space between testing and full model checking.

JPF1 suffered from the problem of translating a complex language such as JAVA to the much simpler language PROMELA, resulting in a sensation that this approach worked for some programs but not for all programs. It was hard to go the last 20%. JPF2 solved part of this problem, but suffered from the obvious problem of state state space explosion. In addition, the model checker itself was a homemade JVM on top of the real JVM, and hence slow.

At this time we came across two inspiring invited talks at the SPIN 2000 workshop, which we organized. The first was a presentation by Jerry Harrow from Compaq on the VISUALTHREADS tool [33]. The purpose of this tool was to support Compaq's customers in avoiding multithreading errors. Specifically two algorithms appeared interesting: predictive data race and deadlock detection. These algorithms can detect the *potential* for a data race or deadlock by analyzing a run that does not necessarily encounter the error. The second invited talk was presented by Doron Drusinsky, on the TEMPORAL ROVER [25] for monitoring temporal logic properties. We implemented the data race algorithm, also known as the Eraser algorithm [61], and a modification of the deadlock detection algorithm in JPF2. The idea was to first execute the program to check for data races and deadlocks using the two very scalable algorithms, and then only if error potentials were found between identified threads, to launch the model checker focusing specifically on those threads.

The two authors of [45] met at NASA Ames in 2000, when the second author started his first job right out of school, and this way, without knowing it, a fruitful, life-time collaboration and friendship with the first author. Inspired by recent joint work with his PhD adviser, Joseph Goguen, the second author was readily convinced that otherwise heavy-weight specification-based analysis techniques can very well apply to execution traces instead of whole systems, and thus achieve scalability by analyzing only what happens at runtime, as it happens. This, paired with provably correct recovery, gives the same level of assurance as formal verification of the whole system, but in a manner that allows us to divide-and-conquer the task. So the second author was "all in", ready to use his fresh algebraic specification and formal verification knowledge to rigorously analyze execution traces.

At this point, the previously mentioned observations about scalability of the traditional verification approaches, the experiments with data race and deadlock detection mentioned above, and some other less technical issues, led to our research focusing just on observing program executions. A constraint was that it should not be based on test case generation, since so many people were studying this already. We wanted to follow the path less explored. This is where the JAVA PATHEXPLORER project began, inspired by other work, but not too much other work.

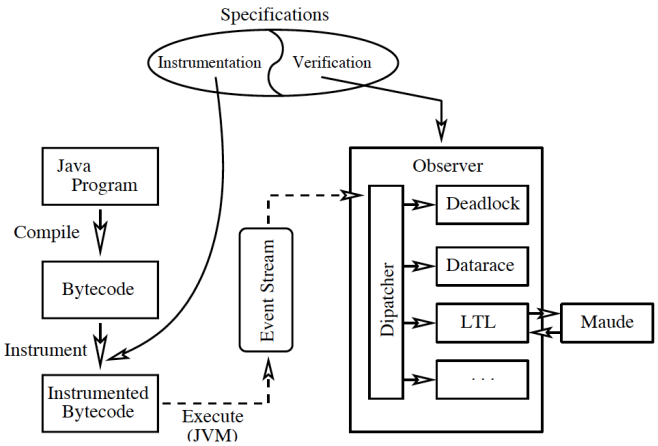

Fig. 1. The JPaX architecture.

3 Java PathExplorer

Our first pure runtime verification system was Java PathExplorer (JPaX), described in the award winning paper [45], as well as in other papers [46–49, 60]. The system is briefly described below.

3.1 Architecture

JPaX was a general framework for analyzing execution traces. It supported three kinds of algorithms: propositional temporal logic conformance checking, data race detection, and deadlock detection, as discussed earlier. Figure 1 shows JPaX's architecture. A Java program is instrumented (at byte code level) to issue events to the monitoring side, which is customizable, allowing the addition of new monitors. The temporal logic monitoring module was originally based on a propositional future time linear temporal logic, but was later extended to also cover past time.

An interesting aspect of the system was the use of the Maude [21] rewriting system for implementing monitoring logics as deep DSLs. One could in very few lines implement, e.g., linear temporal logic (LTL), with syntax and its monitoring algorithm, and have Maude function as the monitoring engine as well. There was a grander vision present at the time: to use a powerful Turing complete language, such as Maude, for monitoring, and not be restricted to just, e.g., LTL. However, that vision did not evolve beyond the thought stage, and had to wait some additional years, as discussed in Section 4. Below we briefly discuss some of the algorithms developed during the JPaX project.

Future Time LTL The future time LTL monitoring used Maude to rewrite formulas. Consider, e.g., the LTL formula $p \,\mathcal{U}\, q$, meaning q eventually becomes

true and until then p is true. The implementation of JPaX was based on classical equational laws for temporal operators, such as:

$$p \,\mathcal{U}\, q = q \wedge \bigcirc(p \,\mathcal{U}\, q) \qquad \text{and} \qquad \Box p = p \wedge \bigcirc(\Box p) \tag{1}$$

Consider the sample formula $\Box(green \rightarrow \bigcirc(\neg red \,\mathcal{U}\, yellow))$. Upon encountering a *green* in a trace, the formula will be rewritten into the following formula, which must be true in the next state: $(\neg red \,\mathcal{U}\, yellow) \wedge \Box(green \rightarrow (\neg red \,\mathcal{U}\, yellow))$. In Maude this was realized by a few simple rewrite rules, including the following two for the until operator (E is an event and T is a trace, the first rule handles the case of a trace consisting of only one event):

```
eq E |= X U Y = E |= Y .
eq E,T |= X U Y = E,T |= Y or E,T |= X and T |= X U Y .
```

3.2 Past Time LTL

Later, an efficient dynamic programming algorithm for monitoring *past time* linear temporal logic was developed [48], inspired by an initial encoding in Maude described in [45]. Consider the following past time formula: $red \rightarrow \blacklozenge green$ (whenever *red* is observed, in the past there has been a *green*). The algorithm for checking past time formulas like this uses two arrays, now and pre, recording the status of each sub-formula now and previously. Index 0 refers to the formula itself with positions ordered by the sub-formula relation. Then for this property, for each observed event the arrays are updated as follows.

```
bool pre[0..3], now[0..3];

fun processEvent(e) {              // Sub-formula:
  now[3] := (event = red)          // red
  now[2] := (event = green)        // green
  now[1] := now[2] || pre[1]       // PREV green
  now[0] := !now[3] || now[1]      // red -> PREV green
  if !now[0] then output("property violated");
  pre := now;
}
```

This dynamic programming algorithm was generalized and optimized in [49, 59] and later found way into three other systems for monitoring parametric temporal formulas, namely MOP [57], MonPoly [11], and DejaVu [40].

3.3 Data Races and Deadlocks

When used for bug finding, the effectiveness of runtime verification depends on the choice of test suite. For concurrent systems this is critical, due to the many possible non-deterministic execution paths. *Predictive runtime verification* approaches this problem by replacing a target property P with a stronger property

Q such that there is a high probability that the program satisfies P iff a random trace of the program will satisfy Q. As already mentioned, one such algorithm was the Eraser algorithm [61], for detecting *potentials* for data races (where two threads can access a shared variable simultaneously). It is often referred to as the *lock set* algorithm as each variable is associated with a set of locks protecting it. The *lock graph* algorithm [33], would detect "dining philosopher"-like deadlock potentials by building a simple lock graph where a cycle indicates a deadlock potential. In [13] we augmented the original lock graph algorithm to reduce false positives in the presence of so-called guard locks (locks that prevent cyclic deadlocks). That paper was later followed by [12], which suggested a code instrumentation method (inserting wait statements) for confirming found deadlock potentials. Other forms of data races than those detected by Eraser are possible. In [3] a dynamic algorithm for detecting so-called high-level data races (races involving collections of variables) is described.

3.4 Code Instrumentation

JPaX code instrumentation was performed with Compaq's JTrek [22], a Java byte code instrumentation tool. Operating at the byte code level offers expressive power, but makes writing code instrumentation instructions inconvenient. An attempt was later made to develop an easier to use code instrumentation tool named JSpy [31] on top of JTrek. In this tool code instrumentation could be expressed as a set of high-level rules, formulated in Java (an internal Java DSL), each consisting of a predicate and an action.

3.5 Trace Visualization

Execution trace visualization is a subject that in our opinion has promising potential, although our own involvement in this direction is limited to [4]. The advantage of visualization is that it can provide a free-of-charge abstract view of the trace, from which a user potentially may be able to conclude properties about the program, or at least the execution, without having to explicitly formulate these properties. We can distinguish between two forms of trace visualization as outlined in [4]: *still visualization*, where all events are visualized in one view, and *animated visualization*. In [4], an extension of Uml sequence diagrams with symbols is described for representing still visualizations of the execution of concurrent programs.

4 The Aftermath

The period after JPaX followed two tracks, which can be summarized as: experiments with aspect-oriented programming for program instrumentation, and so-called parametric monitoring of events carrying data.

4.1 Aspect-oriented Programming

Whilst initial runtime verification frameworks targeted Java, the RMOR (Requirement Monitoring and Recovery) framework [36] targeted the monitoring of C programs against state machines using a homegrown aspect-oriented framework to perform program instrumentation. RMOR was implemented in OCAML using CIL (C Intermediate Language), a C program analysis and transformation system, itself written in OCAML. Later it was attempted to "go all aspect-oriented", meaning that aspects no longer were thought of as just the plumbing for performing code instrumentation, but instead that monitors *are* aspects. Some of our experiments went in the direction of what today is called *state-full aspects* [65, 1]. Here one takes a starting point in an aspect-oriented language framework (such as e.g. ASPECTJ) and extends it with so-called *tracecuts*, denoting predicates on the execution trace. An advice can be associated with a tracecut, and executes when the tracecut is matched by the execution. We proposed this line of work already in [27]. Other later work included [51, 63, 16, 62]. The main observation in these works was that aspect-oriented programming can be extended vertically (allowing more pointcuts) and horizontally (allowing temporal advice, essentially monitoring temporal constraints).

4.2 Runtime Verification with Data

JPAX had a number of limitations. The perhaps most important was the propositional nature of the temporal logics. One could not, for example, monitor parametric events carrying data, such as *openFile("data.txt")*, where *openFile* is an event name and *"data.txt"* is data. It is perhaps of interest to note, that at the time we were not (and are still not) aware of any system that at the time was able to monitor such parametric events in a temporal logic.

4.3 The Beginning of Data

These considerations lead to two different systems: EAGLE [6] and MOP [19]. EAGLE was a small and general logic having similarities with a linear time μ-calculus, supporting monitoring events with data, and allowing user-defined temporal operators. The later HAWK system [23] was an attempt to tie EAGLE to the monitoring of JAVA programs with automated code instrumentation using aspect-oriented programming, specifically ASPECTJ [53].

The same JPAX limitations that motivated the development of EAGLE also stimulated the apparition of monitoring-oriented programming (MOP) [19, 18, 20]. MOP proposed that runtime monitoring be supported and encouraged as a fundamental principle of software development, where monitors are automatically synthesized from formal specifications and integrated at appropriate places in the program. Violations and/or validations of specifications can trigger user-defined code at any points in the program, in particular recovery code, outputting/sending messages, or raising exceptions. MOP has made three important

early contributions. First, it proposed specification formalism independence, allowing users to insert their favorite or domain-specific requirements specification formalisms via *logic plugin* modules. Second, it proposed automated code instrumentation as a means to weave the monitoring checking code within the application; the first version in 2003 used Perl for instrumentation [19], while the subsequent versions starting with 2004 [18] used AspectJ [53]. Finally, it proposed a formalism-independent semantics and implementation for parametric specifications. Conceptually, execution traces are sliced according to each observed instance of the parameters, and each slice is checked by its own monitor instance in a manner that is independent of the employed specification formalism. The practical challenge is how to deal with the potentially huge number of monitor instances. JavaMop proposed several optimizations, presented in [58] together with the mathematical foundations of parametric monitoring.

The Eagle system mentioned earlier was considered quite an elegant system, but its implementation was complicated. The subsequent rule-based lower level Ruler system [9] was meant as an "assembler" into which other temporal specification languages could be compiled for efficient trace checking. However, it assumed a life of its own as a specification language. Ruler was given a finite-trace semantics with four verdicts. The verdicts STILL_TRUE and STILL_FALSE are given if the rule system would accept/reject the trace if it were to end at the current event, whilst the verdicts TRUE and FALSE were reserved for traces where every extension would be accepted/rejected. Ruler allowed for very complex rule systems that could be *chained* together such that one rule system produced outputs for another rule system to consume as input events. Rule systems could be combined sequentially, in parallel, and conditionally.

A project solidly rooted in an actual space mission was the development of the LogScope temporal logic for log analysis [7]. The purpose of the project was to assist the team testing the flight software for JPL's Mars rover Curiosity, which successfully landed on Mars on August 6, 2012. The software produces rich log information. Traditionally, these logs are analyzed with complex Python scripts. The LogScope logic was developed to support notations comprehensible to test engineers, including a very simple and convenient data parameterized temporal logic, which was translated to a form of data parameterized automata, which themselves could be used for specification of more complex properties that the temporal logic could not express. LogScope was furthermore implemented in Python, allowing Python code fragments to be included in specifications, all in order to integrate with the existing Python scripting culture at JPL.

4.4 Internal DSLs

Earlier we mentioned a grander vision to use a powerful Turing complete language for monitoring. The fundamental problem with a logic is that it likely may be insufficient for practical purposes if not designed extremely optimally. Engineers are, e.g., often observed using Python for monitoring tasks. Of course in lack of a better notation, but also because it provides expressive power to perform arbitrary computations, e.g. on observed data. This observation led to

several experiments with so-called internal DSLs, where one extends a programming language with monitoring features. This allows the user to use the features of the programming language when the features of the monitoring logic do not suffice. TraceContract [8, 37] is such an internal Scala DSL (effectively an API) for monitoring, based on a mixture of temporal logic and state machines. It is developed using Scala's features for defining internal DSLs. TraceContract, although a research tool, was later used for analysis of command sequences sent to NASA's LADEE (Lunar Atmosphere and Dust Environment Explorer) spacecraft throughout its mission.

Another example of an internal Scala DSL is LogFire [38]. LogFire is a rule-based system similar to Ruler, but based on a modification of the Rete algorithm [29, 24], used in several rule-based systems. LogFire was part of an investigation of the Rete algorithm's applicability for runtime verification. LogFire has become part of the software that daily processes telemetry data from JPL's Mars Curiosity rover. LogFire's ability to generate facts can be used for Complex Event Processing (CEP) [56], where higher-level events (abstractions) are generated from lower-level events. CEP can be used for further analysis and/or human comprehension, e.g. through visualization. Another CEP system is Nfer [52], which in part was influenced by our work on rule-based systems, and LogFire in particular. The result of applying an Nfer specification to an event stream is a set of time bounded intervals. The specification consists of rules of the form: name :− body (a rule name followed by a rule body). The semantics is similar to that of Prolog (hence the :− symbol): when the body is true an interval is generated with that name. A difference from Prolog is that rule bodies contain temporal constraints based on operators from Allen Temporal Logic [2]. Nfer was created due to a need for comprehending large telemetry streams from Mars rovers. Abstracting these to higher level intervals, compared to the low level raw event stream, should ease human comprehension.

4.5 First-Order Beyond Slicing

Ruler, as a layer of syntactic sugar on top of the rule formalism, offered a sub-formalism resembling a data parameterized automaton language. Likewise, LogScope, inspired by Ruler, offered a data parameterized automaton notation (in addition to the temporal logic). Quantified event automata (Qea) [5] was an attempt to design a pure data parameterized automaton monitoring system logic, using the efficient trace slicing approach previously introduced in the JavaMop tool [57], but dealing with some of the limitations with respect to expressiveness. A Qea specification consists of a list of first-order quantifications (universal and existential) and an automaton. They can be compared to extended state machines (allowing arbitrary guards and actions on transitions operating on local state, but are more succinct due to the fact that automata are "spawned" according to parameters (there is a local state for each combination of parameters).

A different approach to optimizing monitoring of parametric data is implemented in the DejaVu tool [40], which uses BDDs [17] to efficiently represent

data observed in the trace. Logic-wise, the system supports a standard past time temporal logic with quantification. The logic in itself is not the innovation, rather it is the use of BDDs to represent the sets of values observed in the trace for the quantified variables. The representation of sets of assignments as BDDs allows a very simple algorithm that naturally extends the dynamic programming monitoring algorithm for propositional past time temporal logic shown on page 5) and presented in [47], using two vectors *now* and *pre*. However, while in [47] the vectors contain Boolean values, here the values are BDDs.

5 Discussion

Numerous runtime verification logics have been developed over time. They include various forms of temporal logics, state machines, regular expressions, context free grammars, rule systems, variations of the μ-calculus, process algebras, stream processing, timed versions of these, and even statistical versions, where data can be computed as part of monitoring. It is clear that parametric/first-order versions of these logics are needed. Some efforts have been made to combine two or more of these logics, such as, e.g., combining temporal logic and regular expressions. An interesting trend is logics which not just produce a Boolean value, but rather a data value of any type. This leads to systems computing arbitrary data values from traces. It is, however, nearly impossible at this point to estimate which of these approaches would potentially get infused in industrial settings.

Whether to develop a DSL as external or internal is a non-trivial decision. An external DSL is usually cleaner and more directly tuned towards the immediate needs of the user. In addition, they are easier to process and therefore optimize for efficiency. However, the richer the DSL becomes (moving towards Turing-completeness) the harder the implementation effort becomes. An internal DSL can be very fast to implement and augment with new (even user-defined) operators, and can provide an expressiveness that would require a major effort to support in an external DSL. One also gains the advantage of IDEs for the host language. A hypothesis is that monitoring logics used in practice will need to support very expressive expression languages to process data, such as strings and numbers that are part of the observed events. Temporal logic could become part of a programming language assertion language. This could be seen as part of a design-by contract approach also supporting pre/post conditions and class invariants.

An important topic may be inferring specifications from execution traces. Our own limited work in this area includes [64, 54]. Related to specification mining is execution trace visualization (the visualization can be considered a learned model). The advantage of visualization is that it can provide a free-of-charge abstract view of the trace, from which a user potentially may be able to conclude properties about the program, or at least the execution, without having to explicitly formulate these properties.

Full verification is of course preferred over partial verification performed by a monitor. The combination of static and dynamic verification can provide the best of both worlds: prove as much as is feasible statically and verify the remaining proof obligations during runtime. To properly achieve this goal, we need formal specifications not only for the properties to verify, but also for the programming language itself. Moreover, we need provably correct monitor generation techniques, so we can put all the specification and proof artifacts together and assemble a proof of correctness for the entire system. Interestingly, once a specification of the programming language itself is available, then one can go even one step further and monitor the execution of the program even against the language specification. This may seem redundant at first, but it actually makes full sense for some languages with complex semantics, like C. For example, tools like Valgrind or UBSan detect undefined behaviors in C/C++ programs, which are essentially deviations from the intended language semantics. The RV-Match tool [32] is an attempt to push runtime verification in this direction.

In fault-protection strategies, the goal is to recover the system once it has failed. The general problem of how to recover from a bad program state is interesting and quite challenging. The ultimate solution to this problem can be found in planning and scheduling systems, where a planner creates a plan (straight-line program) to execute for a limited time period, an executive executes the plan, and a monitor monitors the execution. Upon failure detected by the monitor, a new plan (program) is generated online.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, October 2005.
2. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
3. C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4), 2004.
4. C. Artho, K. Havelund, and S. Honiden. Visualization of concurrent program executions. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 2, pages 541–546, July 2007.
5. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard. Quantified Event Automata - towards expressive and efficient runtime monitors. In D. Giannakopoulou and D. Méry, editors, *18th International Symposium on Formal Methods (FM'12), Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.
6. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
7. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.
8. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *Proc. of the 17th International Symposium on Formal Methods (FM'11)*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.

9. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. *Journal of Logic and Computation*, 20(3):675–706, 2010.
10. E. Bartocci, Y. Falcone, A. Francalanza, M. Leucker, and G. Reger. An introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–23. Springer, 2018.
11. D. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *Proceedings of the 28th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–60. Schloss Dagstuhl - Leibniz Center for Informatics, 2008.
12. 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.
13. S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Haifa Verification Conference, Haifa, Israel, November 13-16, 2005*, volume 3875 of *LNCS*, pages 208–223. Springer, 2006.
14. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
15. D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982. ISBN 0-13-880733-7.
16. E. Bodden and K. Havelund. Aspect-oriented race detection in Java. *IEEE Trans. Softw. Eng.*, 36(4):509–527, July 2010.
17. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv. ()*, 24(3):293–318, 1992.
18. F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 357 – 373. Springer-Verlag, 2004.
19. F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Proc. of the 3rd Int. Workshop on Runtime Verification (RV'03)*, volume 89(2) of *Elec. Notes Theo. Comput. Sci.*, pages 108 – 127. Elsevier Science Inc., 2003.
20. F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, pages 569–588. ACM, ACM SIGPLAN Notices, 2007.
21. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
22. S. Cohen. JTrek.
23. M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.
24. R. B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1995. Ph.D. Thesis.
25. D. Drusinsky. The temporal rover and the ATG rover. In *Proc. of the 7th International SPIN Workshop on Model Checking and Software Verification (SPIN'00)*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
26. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In M. Broy, D. Peled, and G. Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

27. R. Filman and K. Havelund. Source-code instrumentation and quantification of events. In *Foundations of Aspect-Oriented Languages (FOAL'02), Enschede, The Netherlands*, April 2002.
28. J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.
29. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
30. C. George, P. Haff, K. Havelund, A. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England, 1992.
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. Guth, C. Hathhorn, M. Saxena, and G. Rosu. Rv-match: Practical semantics-based program analysis. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *LNCS*, pages 447–453. Springer, July 2016.
33. J. Harrow. Runtime checking of multithreaded applications with visual threads. In *SPIN 2000, Model Checking and Software Verification*, volume 1885 of *LNCS*. Springer, 2000.
34. K. Havelund. *The Fork Calculus - Towards a Logic for Concurrent ML*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, Denmark, 1994.
35. K. Havelund. Mechanical verification of a garbage collector. In *Fourth International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, 1999.
36. K. Havelund. Runtime verification of C programs. In *Proc. of the 1st TestCom/-FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, 2008. Springer.
37. K. Havelund. Data automata in Scala. In *Proc. of the 8th International Symposium on Theoretical Aspects of Software Engineering (TASE'14)*. IEEE Computer Society, 2014.
38. K. Havelund. Rule-based runtime verification revisited. *Int. J. Softw. Tools Technol. Trans.*, 17(2):143–170, 2015.
39. K. Havelund and A. Goldberg. Verify your runs. In *Verified Software: Theories, Tools, Experiments, VSTTE 2005*, pages 374–383, 2008.
40. K. Havelund, D. A. Peled, and D. Ulus. First order temporal logic monitoring with BDDs. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 116–123. IEEE, 2017.
41. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000.
42. K. Havelund and G. Reger. Runtime verification logics - a language design perspective. In *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday, Aalborg University, 19-20 August 2017*, volume 10460 of *LNCS*, pages 310–338. Springer, 2017.
43. K. Havelund, G. Reger, and G. Roşu. Runtime verification - past experiences and future projections. volume 10000 of *LNCS*. Springer, 2018.
44. K. Havelund, G. Reger, D. Thoma, and E. Zălinescu. Monitoring events that carry data. In E. Bartocci and Y. Falcone, editors, *Lectures on Runtime Verification,*

Introductory and Advanced Topics, volume 10457 of *LNCS*, pages 61–102. Springer, 2018.

45. K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Proc. of the 1st Int. Workshop on Runtime Verification (RV'01)*, volume 55(2) of *Elec. Notes Theo. Comput. Sci.*. Elsevier, 2001. Paris, France, 23 July. Won the RV 2018 Test of Time Award.
46. K. Havelund and G. Roşu. Monitoring programs using rewriting. In *Proc. of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 135–143, 2001.
47. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Grenoble, France, April 8-12, 2002*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.
48. K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2), March 2004.
49. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2):158–173, 2004.
50. K. Havelund and W. Visser. Program model checking as a new trend. *STTT*, 4(1):8–20, 2002.
51. K. Havelund and E. V. Wyk. Aspect-oriented monitoring of C programs. In *The Sixth IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments, Pasadena, CA, May 17-18*, 2008.
52. S. Kauffman, K. Havelund, and R. Joshi. nfer – a notation and system for inferring event stream abstractions. In *Proc. of RV 2016, the 16th International Conference on Runtime Verification, Madrid, Spain, September 23–30, 2016, Proceedings*, volume 10012 of *LNCS*, pages 235–250. Springer, 2016.
53. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.
54. C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 591–600, 2011.
55. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2008.
56. D. Luckham, editor. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.
57. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *J Software Tools for Technology Transfer*, pages 249–289, 2011.
58. G. Roşu and F. Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8(1), 2012.
59. G. Roşu, F. Chen, and T. Ball. Synthesizing monitors for safety properties – this time with calls and returns –. In *Workshop on Runtime Verification (RV'08)*, volume 5289 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2008.
60. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, Apr 2005.
61. 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, Nov. 1997.

62. J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok. InterAspect: aspect-oriented instrumentation with GCC. *Formal Methods in System Design*, 41(3):295–320, 2012.
63. D. R. Smith and K. Havelund. Toward automated enforcement of error-handling policies. Technical Report number: TR-KT-0508, Kestrel Technology LLC, August 2005.
64. S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Proc. of RV 2011, the 11th International Conference on Runtime Verification, San Francisco, CA, USA*, volume 7186 of *LNCS*, pages 193–207. Springer-Verlag, 2011.
65. R. Walker and K. Viggers. Implementing protocols via declarative event patterns. In R. Taylor and M. Dwyer, editors, *ACM Sigsoft 12th International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159–169. ACM Press, 2004.