Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Introduction to the Special Section on Runtime Verification

Oleg Sokolsky[1], **Klaus Havelund**[2⋆], **Insup Lee**[1]

[1] Department of Computer and Info. Science, University of Pennsylvania, Philadelphia, PA 19041, USA
[2] Jet Propulsion Laboratory, California Institute of Technology, California, CA 91109, USA

1 Overview

Runtime verification (RV) is a relatively new area of study that is concerned with dynamic monitoring and analysis of system executions with respect to precisely specified properties. A rigorous approach to proving the correctness of programs at run time was first presented in a paradigm called program checking by Blum and Kannan [10]. The papers on program checking also demonstrated that runtime monitoring was feasible in many instances where static, design-time verification does not appear to be. This was the primary initial motivation for this field at its inception about a decade ago. That is, despite the best verification efforts, problems can occur at run time, and dynamic analysis of the running system can improve confidence in its evolving behavior. Since then, researchers came to realize that precisely specified monitoring and checking can be used for many purposes, such as program understanding, systems usage understanding, security or safety policy monitoring, debugging, testing, verification and validation, fault protection, behavior modification (e.g., recovery), etc. At the most abstract level, a running system can be regarded as a generator of execution traces, i.e., sequences of relevant states or events. Traces can be processed in various ways, e.g., checked against formalized specifications [28], used to drive the simulation of behavioral models, analyzed with special algorithms, visualized [36], etc. Statistical analysis [25] and machine learning of system properties can be performed over the traces.

The scope of RV research covers two conceptually separate aspects. One aspect concerns checking of traces, such as property specification, and algorithms for the checking of such property specifications over a given trace. The other aspect has to do with the generation of traces; that is, how observations are made and recorded. Most RV systems rely on some form of instrumentation to extract observations. Of course, these two aspects are not independent. A trace generation method has to ensure that all observations necessary for checking a property are recorded. A missed observation is likely to result in an incorrect checking outcome. On the other hand, generating irrelevant observations increases checking overhead and should be avoided. Recording exactly the right set of observations for a given property is the subject of trace generation research. This dependency between the two aspects may also be turned around: if the trace generation method is fixed, one can pose the question, which property specification language would be the most appropriate. For example, if the trace records state changes — that is, differences between two successive states — rather than values of state variables, checking a state-based property would require an additional step of reconstructing states.

We distinguish three broad categories of RV approaches to property specification. In the first category, properties depend on execution history and their evaluation depends on a sequence of states in a trace. Such properties are often expressed in a variant of temporal logic; however, other specification languages are also used, such as regular expressions and state machines. The second category relies on the use of contracts or assume-guarantee interfaces between modules in the system. Properties in this category typically describe a single state or changes between two consecutive states in the trace. It may be considered a special case of the previous category. However, logics use for property specification as well as checking algorithms tend to be different here. Finally, the third category, sometimes referred to as specification-less monitoring, develops checking algorithms that detect violations of specific common properties. These properties typically relate to concurrent executions, such as freedom from race conditions [22], atomicity [43], serializability [16], etc.

An important research direction in runtime verification, which cuts across both RV aspects mentioned above, is the management of overhead. An inefficient implementation of the monitoring algorithm can easily be disruptive to system performance and limit the use of RV techniques. A significant

⋆ Part of the research described in this publication was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

source of overhead is instrumentation that is necessary to extract observations from a running system. Reducing the number of instrumentation points reduces instrumentation overhead but may lead to missing observations and incorrect checking results.

Last but not least, an RV framework has to provide the right feedback to the users. In many situations, it is not enough to signal that a violation has occurred. If checking is applied to a running system, as opposed to a recorded trace, feedback from the checker can help the system recover from a problem. Understanding the right feedback to produce, and reasoning about its effects on the system overall behavior is yet another important direction in RV research.

The Special Section on Runtime Verification contains five research papers that consider all of the research questions mentioned above. The rest of this introduction puts these papers in context. Throughout the text, references to papers included in this section are highlighted in bold, to distinguish them from papers published elsewhere.

Our introduction is organized as follows. In Section 2, we discuss checkers based on temporal specifications and contract-based checkers. The third category, specification-less monitoring is extensively covered in a survey **[38]** included in this section, and we do not consider it in this overview paper. Section 3 discusses different approaches to instrumenting software for monitoring. Section 4 considers the problem of overhead reduction. Section 5 discusses feedback. We conclude with the outlook on the future of RV research.

2 Categories of runtime verification

2.1 Monitoring and checking of temporal specifications

This category of RV tools have been developed as a direct extension of the static verification concept of model checking. Model checking algorithms [17] verify a model of the system against properties specified in a temporal logic or similar formalism. The original RV research question was whether an execution of the system can be verified at run time against the same set of properties. From the theoretical perspective, the problem is to deal with the fact that the trace is evolving as checking is being performed. Repeating the process from the beginning of the trace every time a new observation arrives is wasteful. On-line or incremental variants of the algorithms are necessary to make RV efficient.

Efficient on-line algorithms for checking a trace with respect to formulas in past- and future-time temporal logics, as well as regular expressions, have been available for about a decade [9,20,27,28,32] and are incorporated in a variety of tools [11,26,31]. One of the most extensive toolsets for runtime verification of formal specifications is the MOP framework **[34]**, which is presented in this section. It supports a variety of specification formalisms, such as temporal logics and finite state machines, and several monitoring targets, such as Java programs and bus snooping.

Comparison of RV algorithms for different specification formalisms reveals that they have much in common. All of the algorithms maintain a checker state that is updated when new observation from a target system arrives. For example, in the case of past-time linear temporal logic (ptLTL), the checker state contains the valuations of all subformulas of a given formula in the current state of the trace, which is updated using a dynamic programming approach [27].

This similarity between algorithms for seemingly different formalisms has led to the research on special monitoring logics that can be used as the common underlying formalism for specifying monitors. Eagle [3] is a logic with explicit fixpoint operators that is capable of implementing future- and past-time temporal logics, interval logics, extended regular expressions, and other formalisms that may be useful in the monitoring context. An execution engine for Eagle specifications makes it a flexible platform for implementing checkers. RuleR [5] takes this approach one step further by replacing explicit fixed point operators with rules that activate each other in response to observations. RuleR is lower-level logic that requires more effort to encode high-level semantics of a commonly used logic. However, it allows much finer management of the checker state, leading to more efficient checkers, and yields a system that is easier to implement and maintain.

2.2 Design-by-contract monitoring

Design by contract, pioneered in the Eiffel programming language [35], relies on interface specifications between components in the system. These specifications take the form of pre-conditions, post-conditions, and invariants. Contracts are typically state predicates (that is, expressions over the values of system variables in the current state). Contracts also typically allow us to relate "old" and "new" values of variables. Like any other correctness properties, contracts can be verified statically (within the inherent limitations of verification tools), but they often are also checked at run time for unexpected violations.

Unlike temporal specifications discussed earlier, contract checking is typically integrated more tightly with the system execution, and can conceptually be viewed as part of the system. Semantics of contracts determine, at which points the contracts should be checked. For example, a pre-condition for a method in an object-oriented program is checked when the method is called, and a post-condition is checked just before the method returns. Because of this tighter integration, instrumentation is typically not an issue.

Some languages, such as Eiffel and Spec# [2], provide special syntax for contracts. Other language frameworks support embedding of contract as code, usually providing a library of contract primitives. For example, the *Code Contracts* project [23] provides contracts for .NET languages as calls to methods of a Contract class. The TraceContract project [4] supports writing temporal specifications in Scala as calls to methods of a `Monitor` class. There are several comment-based contract frameworks for Java, such as JML [15] and Jass [6]. Both are based on writing specifications as special recognizable comments, which can then be extracted by tools. Some contract checking systems, Jass among them, extend standard contracts with *trace assertions* [7], which can be used to specify restrictions on sequences of invocations of methods of the class. With this extension, the distinction between contracts and temporal properties begins to disappear.

3 Instrumentation

In order to extract observations necessary for checking properties, RV tools rely on instrumentation. Most tools use active instrumentation, that is, insertion of code probes into the running system. One of the few exceptions is BusMOP [37], an instantiation of the MOP framework **[34]** for the monitoring of PCI buses. BusMOP uses passive instrumentation, directly observing traffic on the bus. In order to avoid missing relevant observations and allow RV to scale to large systems, instrumentation should be automatic. Automatic instrumentation requires us to perform analysis of the target system in order to identify where events of interest occur.

An increasingly popular way of implementing instrumentation of source code is through the use of aspect-oriented programming [30]. Indeed, multiple probes are added to the code that collectively supply observations to the monitor. Thus, instrumentation satisfies the definition of a cross-cutting concern, which underlies aspect-oriented programming. The use of aspect-oriented techniques in RV research was pioneered in the tools J-Lo [11] and Hawk [19]. An even closer connection between RV and aspect-oriented instrumentation was made in the AspectBench Compiler using the notion of *tracematches* [1]. A tracematch is an extension of the AspectJ language, which captures a regular pattern of events. In other words, it is similar to other temporal specifications discussed in Section 2.1. However, tracematches are given aspect semantics so that they can be automatically weaved by the compiler. Since then, several RV tools, including MOP **[34]** and Larva [18], rely on AspectJ for instrumentation. Aspect-oriented instrumentation for other programming languages is less common. Here we mention the InterAspect system [41], which performs instrumentation based on the intermediate representation of the GCC compiler, thus providing aspect-oriented instrumentation for languages with GCC front-ends, in particular, C and C++.

Based on the mode of interaction between the instrumented system and the checker, we distinguish between synchronous and asynchronous monitoring. In synchronous monitoring, whenever a relevant observation is produced by the system, further execution is stopped until the checker confirms that no violation has occurred. Synchronous monitoring may deliver a higher degree of assurance than the asynchronous one, because it can block a dangerous action. However, this comes typically with a higher instrumentation overhead. In some applications, where the system can tolerate a slower response, but effects of a property violation are dramatic [18], this additional overhead is justified.

4 Overhead reduction

Runtime checking of complicated properties that involve many system variables imposes high overhead on the system performance. There has been much work on reducing the checking overhead by combining static analysis techniques with subsequent runtime checking.

The concept of combining static and dynamic analysis originates in the programming language community in the context of typestate analysis. Typestate properties are properties of paths in a program and are similar to behavioral specifications studied in the RV literature. A typical typestate property may be expressed as a state machine constraining valid sequences of API calls in a program. Residual typestate analysis has been proposed in [21]. It is based on the observation that, while static typestate analysis often leads to inconclusive results, it produces valuable information that can reduce the state machine that needs to be analyzed at run time.

Complementary to the residual analysis is the work of Eric Bodden and his colleagues [12, 14]. Here, static analysis of the code to be monitored is performed with the goal of identifying instrumentation points that can be safely removed. Results of this work have been implemented in the tool Clara **[13]**, which is included in this section.

In some cases, however, it may be acceptable to lower the accuracy of monitoring by missing some of the execution events. In these cases, there is a trade-off between accuracy and lower overhead. The paper by X. Huang, *et al.*, included in this section **[29]**, presents a control-theoretic approach to implementing this trade-off.

5 Feedback and runtime enforcement

An important question when designing an RV system is what to do when a violation is discovered. If the system is undergoing testing, it may be sufficient to alert the operator, who will stop the system and diagnose the problem. However, in order to realize the vision of RV for a post-deployment alternative to the design-phase verification, a more programmatic approach is needed.

Several RV systems, including MaC and MOP, have the ability to invoke user-specified recovery routines. Monitoring specifications allow the user to specify, what actions are to be invoked when a particular violation occurs, and what information is to be passed to the recovery routine. The system can be partially reset or switched to a failsafe mode. In some situations, for example, when monitoring performance and quality-of-service properties [39], the system can be steered to a more suitable state.

This approach relies on an implicit assumption that the recovery routine will be effective, regardless of the reason the property was violated. For a property that depends on the interaction of several parts of a system, this assumption may not hold. To give a simple example, suppose the property is that the size of a queue, buffering traffic from a sender to a receiver, should be always below a threshold. If the property is violated, it may mean that the sender is sending the messages too fast or that the receiver is processing them too slow. Different recovery actions may be warranted in each of these cases.

A possible approach to solving this deficiency is to incorporate diagnostic facilities into an RV system. The first work to explore this line of research was [8]. A more efficient approach, that combines on-line and off-line techniques has been studied in [42].

Instead of having the monitor react to a property violation, it is sometimes possible to prevent the violation by delaying or reordering events, or by preventing certain events from happening. This extension of the RV research has come to be known as *runtime enforcement*. In general, some events may not be under the control of a monitor. However, runtime enforcement has proved very effective in many important areas such as security. The power of an enforcement monitor determines the class of properties that can be enforced. Blocking of events is sufficient to enforce safety properties [40], while the capability to delay events allows one to handle some liveness properties as well [33]. Falcone, *et al.*, **[24]** generalize existing enforcement approaches in a single framework and further extends the class of enforceable properties.

6 Future of runtime verification

The diversity of research topics centered around the general theme of runtime verification, demonstrated by the articles in this section, is the evidence of the vibrancy of the field and the variety of practical problems that it is addressing. Connections to a wide variety of established research areas such as model-based testing, formal verification, debugging, etc. bode well for RV research as it will keep flourishing as a necessary, complementary technique, without being subsumed by any of these areas.

References

1. C. Allan, P. Avgustinov, S. Kuzins, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, A. S. Christensen, L. Hendren, and O. Lhoták. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 345–364, October 2005.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, March 2004.
3. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57, January 2004.
4. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *Proceedings of 17^{th} International Symposium on Formal Methods(FM'11)*, volume 6664 of *LNCS*, June 2011.
5. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In *Proceedings of the 7^{th} Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125, March 2007.
6. D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with assertions. In *Proceedings of the 1^{st} Workshop on Runtime Verification (RV'01)*, July 2001.
7. W. Bartussek and D.L. Parnas. Using assertions about traces to write abstract specifications for software modules. In *Proceedings of the 2^{nd} Conference on European Cooperation in Informatics*, volume 65 of *LNCS*, pages 211–236, 1978.
8. A. Bauer, M. Leucker, and C. Schallhart. Model-based runtime analysis of distributed reactive systems. In *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)*, pages 243–252, April 2006.
9. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.
10. M. Blum and S. Kannan. Designing programs that check their work. In *Proceedings of the 21^{st} Annual ACM Symposium on Theory of Computing*, pages 86–97, May 1989.
11. E. Bodden. A lightweight LTL runtime verification tool for Java. In *9th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 306–307, October 2004.
12. E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *International Conference on Software Engineering (ICSE'10)*, pages 5–14, May 2010.
13. E. Bodden and L. Hendren. The Clara framework for hybrid typestate analysis. *Software Tools for Technology Transfer, Special Section on Runtime Verification*, in this volume, 2011.
14. E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the 21^{st} European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 525–549, July 2007.
15. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
16. K. Chen, S. Malik, and P. Patra. Runtime validation of transactional memory systems. In *International Symposium on Quality Electronic Design*, pages 750–756, 2008.
17. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2000.
18. C. Colombo, G. Pace, and P. Abela. Compensation-aware runtime monitoring. In *Proceedings of the 1^{st} International Conference on Runtime Verificaiton (RV'10)*, volume 6418 of *LNCS*, pages 214–228, November 2010.
19. M. D'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *Workshop on Dynamic Program Analysis (WODA'05)*, volume 30 of *ACM Sigsoft Software Engineering Notes*, pages 1–7, 2005.
20. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330, 2000.
21. M. B. Dwyer and R. Purandare. Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the 22^{nd} International Conference on Automated Software Engineering (ASE'07)*, pages 124–133, November 2007.

22. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In *Proceedings of the Workshop on Formal Approaches to Testing and Runtime Verification (FATES/RV'06)*, August 2006.
23. M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*, pages 2103–2110, 2010.
24. Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *Software Tools for Technology Transfer, Special Section on Runtime Verification*, in this volume, 2011.
25. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting statistics about runtime executions. *Formal Methods in System Design*, 27(3):253–274, 2005.
26. K. Havelund and G. Rosu. Monitoring Java programs with JavaPathExplorer. In *Proceedings of the 1^{st} Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2001.
27. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 342–356, April 2002.
28. K. Havelund and G. Rosu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2):158–173, August 2004.
29. X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. *Software Tools for Technology Transfer, Special Section on Runtime Verification*, in this volume, 2011.
30. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242, June 1997.
31. M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance approach for Java programs. *Formal Methods in Systems Design*, 24(2):129–155, March 2004.
32. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M.Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.
33. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, 2009.
34. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rosu. An overview of the MOP runtime verification framework. *Software Tools for Technology Transfer, Special Section on Runtime Verification*, in this volume, 2011.
35. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
36. W. De Pauw and S. Heisig. Visual and algorithmic tooling for system trace analysis: a case study. In *Proceedings of the 22^{nd} ACM Symposium on Operating Systems Principles (SOSP'09)*, 2009.
37. R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In *Proceedings of the 29^{th} Real-Time Systems Symposium (RTSS'09)*, pages 481–491, December 2008.
38. S. Qadeer and S. Tasiran. Runtime verification of concurrency-specific correctness criteria. *Software Tools for Technology Transfer, Special Section on Runtime Verification*, in this volume, 2011.
39. U. Sammapun, I. Lee, O. Sokolsky, and J. Regehr. Statistical runtime checking of probabilistic properties. In *Proceedings of the 7^{th} Workshop on Runtime Verification*, volume 4839 of *LNCS*, pages 164–175, March 2007.
40. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
41. J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S.A. Smolka, S.D. Stoller, and E. Zadok. Aspect-oriented instrumentation with GCC. In *Proceedings of the 1^{st} International Conference on Runtime Verification*, volume 6418 of *LNCS*, pages 405–420, November 2010.
42. S. Tripakis. A combined on-line/off-line framework for blackbox fault diagnosis. In *Proceedings of the 9^{th} Workshop on Runtime Verification (RV'09)*, pages 152–167, July 2009.
43. Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, February 2006.