

Dynamic Program Analysis

Background, Existing Systems, Challenges

Klaus Havelund

Kestrel Technology, Palo Alto, California, USA

<http://www.havelund.com>

1 Background and Existing Systems

Program verification is difficult and it is likely that parts of the verification task will remain as unproved proof obligations. It is reasonable to not throw such proof obligations away, but to monitor them during program testing, or in the operations phase, allowing property violations to trigger proper actions to be executed. Dynamic program analysis, sometimes referred to as runtime verification, in the broadest sense means using information from program runs to conclude or check properties about a program. Runtime verification is fundamentally an aspect of testing, but tries to go beyond more ad hoc approaches by using logic and automata for expressing correctness properties and by using advanced algorithms for extracting optimal monitors from specifications. Much work has been done within the runtime verification area within the last 5 years plus, for example as documented by the proceedings the two workshops: RV (Runtime Verification) [1] and WODA (Workshop on Dynamic Analysis) [50].

Logics Specification-based runtime verification consists of monitoring a program's execution against a *user-provided specification* of intended program behavior. The property specification approaches include simple predicate assertions as in Java, pre-post conditions and invariants as in the Eiffel programming language [19], JML [33], or Larch [36] that supports abstract (axiomatic) data specification in combination with pre-post conditions.

Of more temporal nature are process algebras and automata-based approaches, where the specifications are process descriptions or state machines. The Jass system [32] uses a CSP-like process algebra in a combination with pre-post conditions. Specifications are here abstract programs. Also state machines have been used, as exemplified in the TLChart system [18], that monitors a combination of temporal logic and state machines. Systems for monitoring timed automata are T-UPPAAL [44] and another similar system is described in [9]. In [14] is described a technique for monitoring 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 [42].

A majority of existing runtime monitoring systems, however, attempt to support more declarative property descriptions, such as temporal logic, regular ex-

pression languages, or grammar languages inspired by parser generators. Temporal logics can be past-time logics [35, 30, 31], future time logics [24, 29, 27, 43], regular expressions [40, 3], grammars [48], or real time logics [16, 45]. Some logics attempt to mix these features into one logic, also allowing data to be referred to across time points [16, 22, 21, 7, 17, 21, 15]. The logic PSL [37] has been adopted by the hardware industry and is used for checking simulations, essentially an application of runtime verification. Temporal logics are often mapped to automata, although other interpretations are possible, such as for example described in [38, 28], where rewriting is used to interpret temporal logic for monitoring.

A monitoring language may be a complete formal specification language. This is the approach taken at Microsoft where ASML (Abstract State Machine Language) [6] is used for runtime verification as part of a general test case generation framework.

Aspect Oriented Programming and Instrumentation An interesting trend is that runtime verification techniques are starting to appear within the aspect oriented programming community. In a traditional AOP language such as AspectJ [34], an aspect consists of advices that specify augmentations/modifications to an existing program. An aspect can specify the addition of functionality or it can specify a correctness property and appropriate actions to be executed when the property is violated. Traditionally actions are weaved into the program when certain code patterns occur, specified by what is 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 [3], future time linear temporal logic [43, 13], state machines [47], and grammars [48]. Alternative ways of instrumenting code include BCEL [8], Valgrind [46], Java-MOP [11].

Algorithm-Based Runtime Verification Some systems use pre-programmed algorithms to detect errors, usually concency errors. Examples are data race detection algorithms [39, 4, 5, 23, 49, 26] and and deadlock detection algorithms [25, 10, 2, 20, 26]. These algorithms check whether there are any *potentials* for errors rather than checking whether any errors occur, which makes these algorithms unsound and incomplete, but very effective in practice. A generalized predictive analysis framework is presented in [41]. A characteristic of these methods is that they require no user input in the form of a specification. An interesting subject is generating specifications from runs [12], thereby freeing humans from writing them. This is the dual of checking runs against specifications.

2 Challenges

The challenges in this field consist of:

- Determining appropriate specification languages for stating properties about programs that facilitate efficient monitoring.

- Determining the appropriate instrumentation machinery that allows for ease of use and optimal performance leading to minimal runtime impact.
- Determining the appropriate programming language constructs supporting runtime verification. The several extensions of AspectJ are examples of a tight integration of logic with programming language.
- Determining how to state actions to be executed when properties are violated/satisfied.
- Determining how to prove a system together with its monitor correct. Does runtime verification make it possible to prove systems correct (for example in case a property is violated, a simple provable correct replacement program is executed).
- Determining the interaction between static analysis and dynamic analysis.

References

1. *The Runtime Verification (RV) Workshops*, volume 55(2), 70(4), 89(2), 113 and RV'05 to be published of *ENTCS*. Elsevier Science Direct. <http://www.runtime-verification.org>.
2. R. Agarwal, L. Wang, and S. D. Stoller. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD - 3), IBM Verification Conference, Haifa, Israel*, November 2005. To be published in LNCS.
3. 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.
4. C. Artho, K. Havelund, and A. Biere. High-Level Data Races. *Software Testing, Verification and Reliability*, 13(4), 2004.
5. 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.
6. ASML. <http://research.microsoft.com/fse/asml>.
7. 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.
8. BCEL. <http://jakarta.apache.org/bcel>.
9. 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. <http://www.runtime-verification.org>.
10. S. Bensalem and K. Havelund. 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.
11. 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]. <http://www.runtime-verification.org>.
12. Daikon. <http://pag.csail.mit.edu/daikon>.

13. M. D'Amorim and K. Havelund. Runtime Verification for Java. In *Workshop on Dynamic Program Analysis (WODA'05)*, March 2005.
14. M. D'Amorim and G. Rosu. Efficient Monitoring of Omega-Languages. In *CAV'05*, LNCS. Springer-Verlag, 2005.
15. 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.
16. 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.
17. D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *CAV'03*, volume 2725 of LNCS, pages 114–118. Springer-Verlag, 2003.
18. 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. <http://www.runtime-verification.org>.
19. Eiffel. <http://www.eiffel.com>.
20. Y. Eytani, K. Havelund, S. Stoller, and S. Ur. Toward a Benchmark for Multi-Threaded Testing Tools. *Concurrency and Computation: Practice and Experience*, 2005. To appear.
21. 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. <http://www.runtime-verification.org>.
22. 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. <http://www.runtime-verification.org>.
23. C. Flanagan and S. Freund. Atomizer: A Dynamic Atomicity Checker for Multi-threaded Programs. *SIGPLAN Not.*, 39(1):256–267, 2004.
24. 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.
25. 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.
26. 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.
27. 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. <http://www.runtime-verification.org>.
28. 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.
29. K. Havelund and G. Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, 24(2), March 2004.
30. K. Havelund and G. Roşu. Efficient Monitoring of Safety Properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.

31. 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.
32. Jass. <http://csd.informatik.uni-oldenburg.de/~jass>.
33. JML. <http://www.cs.iastate.edu/~leavens/JML>.
34. 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.
35. 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]. <http://www.runtime-verification.org>.
36. Larch. <http://www.cs.iastate.edu/larch-faq-webboy.html>.
37. PSL/Sugar. <http://www.pslsugar.org>.
38. G. Roşu and K. Havelund. A Rewriting-based Approach to Trace Analysis. *Automated Software Engineering*, 12(2):151–197, 2005.
39. 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.
40. 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. <http://www.runtime-verification.org>.
41. 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.
42. SPIN. <http://spinroot.com>.
43. 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.
44. T-UPPAAL. <http://www.cs.aau.dk/~marius/tuppaal>.
45. 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. <http://www.runtime-verification.org>.
46. Valgrind. <http://valgrind.org>.
47. W. Vanderperren, D. Suvé, M. Augustina Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. In *Workshop on Software Composition, ETAPS 2005*, 2005.
48. 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.
49. L. Wang and S. Stoller. Run-Time Analysis for Atomicity. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV'03)* [1]. <http://www.runtime-verification.org>.
50. WODA06. Fourth international workshop on dynamic analysis (woda 2006). <http://www.cs.arizona.edu/~ngupta/WODA06>.