AFRL-OSR-VA-TR-2013-0164

Survivable Software

Scott Smolka, Radu Grosu, Klaus Havelund, Scott Stoller, Erez Zadok
State University of New York

April 2013
Final Report

DISTRIBUTION A: Approved for public release.

AIR FORCE RESEARCH LABORATORY
AF OFFICE OF SCIENTIFIC RESEARCH (AFOSR)
ARLINGTON, VIRGINIA 22203
AIR FORCE MATERIEL COMMAND

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to the Department of Defense, Executive Services and Communications Directorate (0704-0188). Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(Include area code)*

Final Report: Survivable Software

Scott A. Smolka, Radu Grosu, Scott D. Stoller, and Erez Zadok
Stony Brook University

Klaus Havelund
NASA JPL

February 28, 2013

Abstract

The *Survivable Software* framework (SSW), developed under award AFOSR FA9550-09-1-0481, uses a synergistic combination of (1) compiler-assisted, aspect-oriented program instrumentation, (2) software monitoring with overhead control, (3) runtime verification with state estimation, and (4) adaptive runtime verification to closely monitor high-criticality monitor instances, thereby increasing the probability of violation detection and concomitantly allowing for appropriate repair and recovery actions to be initiated. Applications include online and offline analysis of operating system kernel-level concurrency, and the analysis of NASA space-mission software. This final report discusses each of these key components of the Survivable Software framework, and highlights project accomplishments on a year-by-year basis, including the production of three PhD dissertations.

1 Introduction

This is the final report for award AFOSR FA9550-09-1-0481, *Survivable Software*, 6/1/09 to 11/30/12. The PI is Scott Smolka and the co-PIs are Radu Grosu, Klaus Havelund (NASA JPL), Scott Stoller, and Erez Zadok. The project web site is `http://www.fsl.cs.sunysb.edu/ssw/`

This grant was awarded within the Software and Systems program. The original Program Manager was David Luginbuhl. Bob Bonneau took over as PM in March 2011.

The award period has been a highly productive one for us. We have published more than 28 project-related papers, released software and documentation for several project-related systems, and, as discussed below, made significant advances on a number of research areas that collectively constitute the Survivable Software (SSW) framework.

The SSW framework comprises the following six key components:

1. *InterAspect* (IA) [12–14], a highly flexible, aspect-oriented program instrumentation framework, based on the GCC plug-in architecture.

2. *Software Monitoring with Controllable Overhead* (SMCO) [6, 8, 9], which uses event sampling and feedback control to provide cost-effective runtime monitoring with bounded-overhead guarantees.

3. *Runtime Verification with State Estimation* (RVSE) [12, 16], a technique that allows a runtime monitor to estimate the probability that a temporal property is satisfied

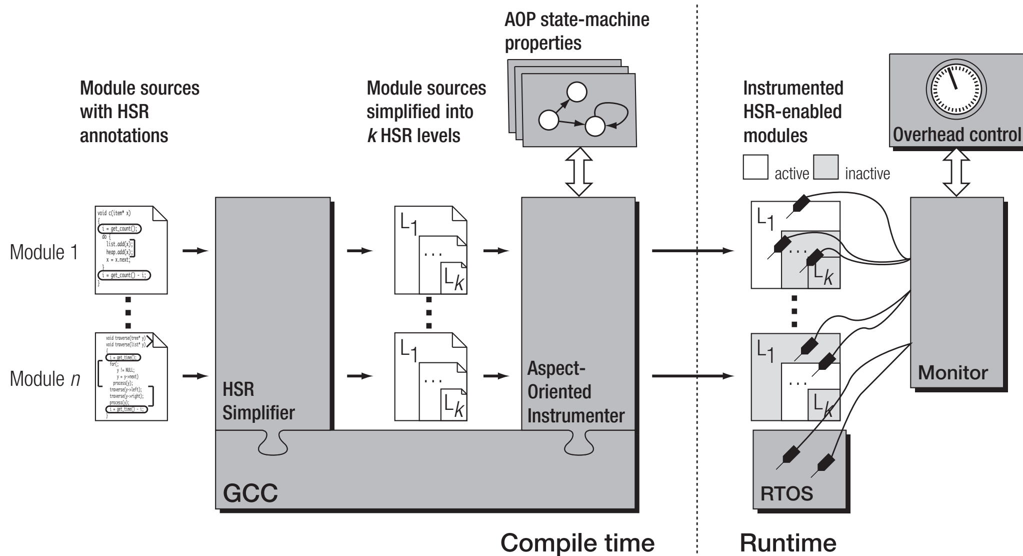

Figure 1: Architectural overview of the Survivable Software framework.

by a run of a program, when there are gaps in the observed execution due to event sampling.

4. *Adaptive Runtime Verification* (ARV) [4, 12], a new approach to runtime verification in which SMCO, RVSE, and predictive analysis are synergistically combined so that high-criticality monitor instances are allocated a higher portion of the available monitoring overhead, thereby increasing the probability of violation detection.

5. *Redflag* [12,15], a framework that uses compiler-assisted instrumentation for targeted offline and online analysis of operating system kernel-level concurrency.

6. *Hierarchical Simplified Redundancy* (HSR), an approach to making software survivable, in which a number of successively simpler versions of each module are developed. When a runtime error is detected, the system can failover to a simpler but potentially more robust version of the module in question.

The overall architecture of the SSW framework is given in Figure 1.

The remainder of this report develops along the following lines. Sections 2-6 describe each of the components of the SSW framework in more detail. Section 7 concludes by outlining project highlights on a year-by-year basis.

2 InterAspect

In [12–14], we presented the InterAspect instrumentation framework for GCC, a widely used compiler infrastructure. The addition of plug-in support in the latest release of GCC makes it an attractive platform for runtime instrumentation, as GCC plug-ins can directly add instrumentation by transforming the compiler's intermediate representation. Such transformations, however, require expert knowledge of GCC internals. InterAspect addresses this situation by allowing instrumentation plug-ins to be developed using the familiar vocabulary of Aspect-Oriented Programming: pointcuts, join points, and advice functions. Moreover, InterAspect uses specific information about each join point in

a pointcut, possibly including results of static analysis, to support powerful customized instrumentation.

We have also developed the InterAspect *Tracecut extension* to generate program monitors directly from formally specified tracecuts. A tracecut [17] matches *sequences of pointcuts* specified as a regular expression. Given a tracecut specification T, InterAspect Tracecut instruments a target program so that it communicates program events and event parameters directly to a monitoring engine for T. The tracecut extension adds the necessary monitoring instrumentation exclusively with the InterAspect API.

To illustrate InterAspect's practical utility, we have developed a number of program-instrumentation plug-ins that use InterAspect for custom instrumentation. These include a *heap visualization* plug-in designed for the analysis of JPL Mars Science Laboratory software; an *integer range analysis* plug-in that helps find bugs by tracking the range of values for each integer variable; and a *code coverage* plug-in that, given a pointcut and test suite, measures the percentage of join points in the pointcut that are executed by the test suite.

The full source of the InterAspect framework is available from the InterAspect website under the GPLv3 license [10].

3 Software Monitoring with Controllable Overhead

In [6,8,9], we introduced the technique of *Software Monitoring with Controllable Overhead* (SMCO), which is based on a novel combination of supervisory control theory of discrete event systems and linear PID-control theory of continuous systems. SMCO controls monitoring overhead by temporarily disabling monitoring of selected events for as short a time as possible under the constraint of a user-supplied target overhead o_t. This strategy is optimal in the sense that it allows SMCO to monitor as many events as possible, within the confines of o_t. SMCO is a general monitoring technique that can be applied to any system interface or API.

We have applied SMCO to a variety of monitoring problems, including: *integer range analysis*, which determines upper and lower bounds on integer variable values; and *Non-Accessed Period (NAP) detection*, which detects stale or underutilized memory regions. We benchmarked SMCO extensively, using both CPU- and I/O-intensive workloads, which often exhibited highly bursty behavior. We demonstrated that SMCO successfully controls overhead across a wide range of target-overhead levels; its accuracy monotonically increases with the target overhead; and it can be configured to distribute monitoring overhead fairly across multiple instrumentation points.

4 Runtime Verification with State Estimation

In [12,16], we introduced the concept of *runtime verification with state estimation* (RVSE), and showed how this concept can be applied to estimate the probability that a temporal property is satisfied by a run of a program when monitoring overhead is reduced by sampling. In such situations, there may be *gaps* in observed program executions, making accurate estimation challenging.

The main idea behind our approach is to use a statistical model of the monitored system to "fill in" sampling-induced gaps in event sequences, and then calculate the probability that the property is satisfied. In particular, we appeal to the theory of Hidden Markov Models [11]. An HMM is a Markov model in which the system being modeled is assumed

to be a Markov process with unobserved (hidden) states. In a regular Markov model, states are directly visible to the observer, and therefore state transition probabilities are the only required parameters. In an HMM, states cannot be observed; rather, each state has a probability distribution for the possible observations (formally called *observation symbols*). The classic *state estimation* problem for HMMs is to compute the most likely sequence of states that generated a given observation sequence.

The main contributions of this work are:

- We use HMMs to formalize the RVSE problem as follows. Given an HMM system model H, temporal property ϕ, and observation sequence O (an execution trace that may have gaps due to sampling), compute $\Pr(\phi \mid O, H)$, i.e., the probability that the system's behavior satisfies ϕ, given O and H. Note that we use *Hidden* Markov Models, meaning that the states of the system are hidden from the observer. This is because we intend to use machine learning to learn the HMM from traces that contain only observable actions of the system, not detailed internal states of the system.

- The *forward algorithm* [11] is a classic recursive algorithm for computing the probability that, given an observation sequence O, an HMM ended in a particular state. This problem is the so-called *filtering* version of the state estimation problem for HMMs. We present an extension of the forward algorithm for the RVSE problem that computes a similar probability, but in this case for the paired execution of an HMM system model and a monitor automaton for the temporal property ϕ. We first present a version of the algorithm that does not consider gaps; in this case, the states of the monitor are completely determined by O, because the monitor is deterministic.

- We then present an algorithm that handles gaps. We use a special symbol to mark gaps, i.e., points in the observation sequence where unobserved events might have occurred. Gap symbols may be inserted in the trace by the instrumentation when it temporarily disables monitoring; or, if gaps may occur everywhere, a gap symbol can be inserted at every point in the trace. When the algorithm processes a gap, no observation is available, so the state of the monitor automaton is updated probabilistically based on the current state estimation for the HMM and the observation probability distribution for the HMM. Since the length of a gap (i.e., the number of consecutive unobserved events) might be unknown, we allow the gap length to be characterized by a probability distribution.

- We evaluate our RVSE methodology using a case study based on human operators in a ground station issuing commands to a Mars rover [1]. Sampling of execution traces is simulated using SMCO-style overhead control [9]. Our evaluation demonstrates high prediction accuracy for the probabilities computed by our algorithm. It also shows that our technique is much more accurate than simply evaluating the temporal property on the given observation sequences, ignoring the gaps.

5 Adaptive Runtime Verification

Adaptive Runtime Verification (ARV) [4] is a new approach to runtime verification in which overhead control, runtime verification with state estimation, and predictive analysis are synergistically combined. Overhead control maintains the overhead of runtime verification

at a specified target level, by enabling and disabling monitoring of events for each monitor instance as needed.

In ARV, predictive analysis based on a probabilistic model of the monitored system is used to estimate how likely each monitor instance is to violate a given temporal property in the near future, and these *criticality levels* are fed to the overhead controllers, which allocate a larger fraction of the target overhead to monitor instances with higher criticality, thereby increasing the probability of detecting a violation if a violation occurs.

Since overhead control causes the monitor to miss events, we use Runtime Verification with State Estimation (RVSE) to estimate the probability that a property is satisfied by an incompletely monitored run. A key aspect of the ARV framework is a new algorithm for RVSE that performs the calculations in advance, dramatically reducing the runtime overhead of RVSE, at the cost of introducing some approximation error. We have demonstrated the utility of ARV on a significant case study involving runtime monitoring of concurrency errors in the Linux kernel.

6 Redflag

In [12,15], we presented Redflag, a framework that uses compiler-assisted instrumentation for targeted offline and online analysis of operating system kernel-level concurrency.

Although there are previous tools for runtime detection of several kinds of concurrency errors, they cannot easily be used at the kernel level. Redflag brings these essential techniques to the Linux kernel by addressing issues faced by other tools. First, other tools typically examine every potentially concurrent memory access, which is infeasible in the kernel because of the overhead it would introduce. Redflag supports user-configurable targeted monitoring of specified kernel components and data structures, to reduce overhead and avoid presenting developers with error reports for components they are not responsible for. Second, other tools do not take into account some of the synchronization patterns found in the kernel, resulting in false alarms; Redflag is designed to take those patterns into account.

In offline analysis mode, Redflag uses an efficient logging system to obtain execution traces that can be analyzed for data races, potential atomicity violations, and incorrect use of a processor's weak memory model. Our analysis uses known algorithms that we enhanced to take into account some specifics of synchronization in the kernel. The most significant enhancement is the introduction of *Lexical Object Availability (LOA)* analysis to deal with multi-stage escape and other complex order-enforcing synchronization. Multi-stage escape is a sophisticated synchronization pattern in which initialization of a data structure is divided into phases, and in each phase, specific parts of the data structure become accessible to specific code segments that may be executed concurrently in other threads.

In online analysis mode, Redflag can detect atomicity violations as they occur. Redflag can also perturb the thread schedule, with the goal of creating schedules that lead to actual atomicity violations. Online mode allows analysis of long executions that generate too many events to practically log.

Our experience applying Redflag to two Linux file systems and a video driver demonstrated its effectiveness for analysis of kernel-level concurrency.

7 Project Highlights

Year 1 (6/1/09 to 5/31/10)

1. Software Monitoring with Controllable Overhead (SMCO) uses control theory to monitor as many events as possible within the confines of a user-supplied target overhead. For both CPU- and I/O-intensive, often bursty workloads, we succeeded in reducing SMCO's base overhead from 18-20% to a much more acceptable 3-4%.

2. We published a paper [9] on SMCO in the *International Journal on Software Tools for Technology Transfer* (STTT).

3. We developed the core functionality of InterAspect, a new Aspect-Oriented Programming API that operates at the level of a compiler's intermediate representation. The advantages of this approach include the possibility of joinpoint-specific and static-analysis-aware code weaving for more efficient and effective source-code instrumentation. We used InterAspect to develop a memory-visualization plug-in for use by JPL's Mars Science Laboratory Flight Software team.

4. We also developed a rule-based framework for monitoring sequences of events against formal requirements expressed in temporal logic. A requirement in our framework consists of a set of rewrite rules that may contain event patterns involving data variables, over which concrete events emitted by InterAspect source-code instrumentation are matched. Such data parameterization significantly increases the expressive power of requirement monitoring.

5. Sean Callanan completed his PhD dissertation [6] on *Flexible Debugging with Controllable Overhead*.

Year 2 (6/1/10 to 5/31/11)

1. We extended InterAspect, the GCC-based aspect-oriented instrumentation framework developed during the first year of the grant, with a *tracecut mechanism* that allows matching of sequences of runtime events against a property specification given as a regular expression with parameters. Our approach interprets a tracecut specification as a finite state machine, and generates the code needed to perform the state machine transitions.

2. We developed the *TraceContract API* for trace analysis in the Scala programming language. TraceContract combines a high-level language with data-parameterized state machines and temporal logic, and has been adopted by the NASA LADEE mission for validating spacecraft command sequences.

3. In our efforts to extend the Simplex architecture for dynamic control-system upgrade to hybrid systems, we designed a high-performance controller for a nonlinear hybrid model of excitable cells (e.g. cardiac cells, neurons). Our controller uses a lookup table to achieve the maximum action-potential duration for a given stimulus magnitude and period, a desired biological response.

4. In conjunction with a Sabbatical-year visit by PI Smolka to co-PI Havelund at NASA JPL, we started the development of a framework for performing runtime verification of temporal and quantitative properties in the presence of incomplete traces due to event sampling. This framework eventually became RVSE [12, 16].

5. Xiaowan Huang completed his PhD dissertation [8] on *Compiler-Assisted Software Model Checking and Monitoring*.

Year 3 (6/1/11 to 5/31/12)

1. Our paper on RVSE [16] in the *Second International Conference on Runtime Verification* (RV 2011) received the *Best Paper Award*.

2. We published a paper [15] on Redflag in ICA3PP 2011 demonstrating how targeted and configurable monitoring of specific kernel data structures, together with enhanced runtime analysis algorithms, can achieve highly effective kernel-level concurrency analysis.

3. In a TACAS 2011 paper [5], we introduced and addressed the problem of *Model Repair for Probabilistic Systems*. Given a probabilistic system M and a temporal property φ such that M fails to satisfy φ, Model Repair seeks to find an M' that satisfies φ and differs from M only in the transition flows. Moreover, the cost associated with modifying M's transition flows to obtain M' is minimized.

4. In a Scala Days 2011 paper [3] and an FM'11 paper [2], we showed how the TraceContract DSL (Domain-Specific Language) can be used to write executable specifications of NASA space-mission flight rules and for monitoring such rules. The DSL offers the combination of high-level programming with temporal logic, and is currently used by two NASA missions (SMAP@JPL and LADEE@NASA Ames).

5. We released the source code and documentation for InterAspect (see Section 2), an open-source framework for writing aspect-oriented GCC plug-ins. Please see [10] for the InterAspect web site.

Year 4 (6/1/12 to 11/30/12)

1. Extended our technique of Runtime Verification With State Estimation (RVSE) [12, 16] with *peek events*: observation symbols of the form $peek(S_p)$, where S_p is a subset of the states of the monitor (DFSM) representing the temporal property under investigation. A peek event represents knowledge from an oracle that the DFSM can only be in one of the states in the set S_p. The incorporation of peek events into RVSE can significantly improve state estimation [12].

2. Developed a rule-based runtime verification engine in Scala, based on the RETE algorithm [7]. This work explores the ideas known from well-studied production systems developed in the field of AI, but in the context of runtime verification. The algorithm has been modified to suit the RV domain. The implementation is being used at JPL to analyze telemetry from the Curiosity Rover currently operating in Mars.

3. Justin Seyster completed his PhD dissertation [12] on *Runtime Verification of Kernel-Level Concurrency Using Compiler-Based Instrumentation*.

References

[1] 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.

[2] H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *17th International Symposium on Formal Methods (FM'11), Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.

[3] H. Barringer, K. Havelund, E. Kurklu, and R. Morris. Checking flight rules with TraceContract: application of a Scala DSL for trace analysis. In *Scala Days 2011, Stanford University, California*, 2011.

[4] E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok, and J. Seyster. Adaptive runtime verification. In *Proc. 3rd International Conference on Runtime Verification (RV'12)*, pages 168–182, Istanbul, Turkey, September 2012. LNCS Vol. 7687, Springer.

[5] E. Bartocci, R. Grosu, P. Katsaros, C. R. Ramakrishnan, and S. A. Smolka. Model repair for probabilistic systems. In *Proceedings of TACAS 2011, 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 326–340, Saarbrucken, Germany, March-April 2011. Springer, LNCS Vol. 6605.

[6] S. Callanan. *Flexible Debugging with Controllable Overhead*. PhD thesis, Computer Science Department, Stony Brook University, August 2009. Technical Report FSL-09-01, `www.fsl.cs.sunysb.edu/docs/callanan09phdthesis/callanan09phdthesis.pdf`.

[7] K. Havelund. What does AI have to do with RV? In *Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA'12), Heraklion, Crete, France, October 15-18, 2012. Proceedings*, volume 7610 of *LNCS*. Springer, 2012.

[8] X. Huang. *Compiler-Assisted Software Model Checking and Monitoring*. PhD thesis, Computer Science Department, Stony Brook University, December 2010. `www.fsl.cs.sunysb.edu/ssw/files/Download/xiaowan-disssertation.pdf`.

[9] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(3):327–347, 2012.

[10] InterAspect. `www.fsl.cs.stonybrook.edu/interaspect`.

[11] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[12] J. Seyster. *Runtime Verification of Kernel-Level Concurrency Using Compiler-Based Instrumentation*. PhD thesis, Computer Science Department, Stony Brook University, December 2012. `www.fsl.cs.sunysb.edu/docs/jseyster-dissertation/redflag.pdf`.

[13] 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 *Proc. of the 1st International Conference on Runtime Verification (RV 2010)*, Lecture Notes in Computer Science. Springer, November 2010.

[14] 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*, August 2012.

[15] J. Seyster, P. Radhakrishnan, S. Katoch, A. Duggal, S. D. Stoller, and E. Zadok. Redflag: A framework for analysis of kernel-level concurrency. In *Proc. of the 11th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'11)*, Melbourne, Australia, October 2011.

[16] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Proc. 2nd International Conference on Runtime Verification (RV'11)*, pages 193–207, San Fransisco, CA, September 2011. LNCS Vol. 7186, Springer. (**Won best paper award**).

[17] 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.