7^{th} International Workshop on Run-time Verification

Preliminary Proceedings

March 13, 2007

Vancouver, British Columbia, Canada

Program Committee:

Mehmet Aksit	University of Twente
Howard Barringer	University of Manchester
Saddek Bensalem	VERIMAG Laboratory
Eric Bodden	McGill Univeristy
Bernd Finkbeiner	Saarland University
Cormac Flanagan	University of California, Santa Cruz
Vijay Garg	University of Texas, Austin
Klaus Havelund	NASA Jet Propulsion Laboratory
Gerard Holzmann	NASA Jet Propulsion Laboratory
Moonzoo Kim	KAIST
Martin Leucker	Technical University of Munich
Oege de Moor	Oxford University
Klaus Ostermann	Darmstadt University of Technology
Shaz Qadeer	Microsoft Research
Grigore Rosu	University of Illinois, Urbana-Champaign
Henny Sipma	Stanford University
Oleg Sokolsky (co-Chair)	University of Pennsylvania
Scott Stoller	State University of New York, Stony Brook
Mario Sudholt	Ecole des Mines de Nantes-INRIA
Serdar Tasiran (co-Chair)	Koc University

Steering Committee:

Klaus Havelund	NASA Jet Propulsion Laboratory
Gerard Holzmann	NASA Jet Propulsion Laboratory
Insup Lee	University of Pennsylvania
Grigore Rosu	University of Illinois, Urbana-Champaign

Workshop Program

09.00 – 10.30 Session 1

Introduction and welcome

Invited talk: *Using PSL for Runtime Verification: Theory and Practice*
Cindy Eisner (IBM Haifa Research Lab)

Temporal Assertions with Parametrised Propositions
V. Stolz (United Nations U.)

11:00 - 12:30 Session 2

From Interaction Patterns to Aspects: a Mechanism for Systematic Runtime Monitoring
I. Krueger, M. Menarini (UC San Diego) and M. Meisinger (TU Munich)

ARVE: Aspect-oriented Runtime Verification Environment
H. Shin, Y. Endoh and Y. Kataoka (Toshiba)

Collaborative Runtime Verification with Tracematches
E. Bodden, L. Hendren (McGill U.), O. Lhotak and N. A. Naeem (U. of Waterloo)

Static and Dynamic Detection of Behavioral Conflicts between Aspects
P. Durr, L. Bergmans and M. Aksit (U. of Twente)

A Semantic-based Runtime Weaver for Dynamic Management of the Join Point History
C. Herzeel, K. Gybels and P. Costanza (Free U. Brussels)

14:00 - 15:30 Session 3

Translation Validation of System Abstractions
J. O. Blech, I. Schaefer and A. Poetzsch-Heffter (U. of Kaiserslautern)

Runtime Checking for Program Verification Systems
K. Zee, V. Kuncak and M. Rinard (MIT)

Rule Systems for Run-Time Monitoring: from Eagle to RuleR
H. Barringer, D. Rydeheard (U. of Manchester) and K. Havelund (NASA JPL)

The Good, the Bad, and the Ugly, but how Ugly is Ugly?
A. Bauer, M. Leucker and C. Schallhart (TU Munich)

Towards a Tool for Generating Aspects from MEDL and PEDL Specifications for Runtime Verification
O. Ochoa, I. Gallegos, S. Roach and A. Gates (UT El Paso)

16:00 - 17:30 Session 4

From Runtime Verification to Evolvable Software
H. Barringer, D. Rydeheard (U. of Manchester) and D. Gabbay (King's College)

Instrumentation of Open-Source Software For Intrusion Detection
W. Mahoney and W. L. Sousan (U. of Nebraska, Omaha)

A Causality-Based Runtime Check for Atomicity
S. Tasiran and T. Elmas (Koc U.)

On the Semantics of Matching Trace Monitoring Patterns
P. Avgustinov, O. de Moor and J. Tibble (Oxford U.)

Statistical Runtime Checking of Probabilistic Properties
U. Sammapun, I. Lee and O. Sokolsky (U. of Pennsylvania)

Temporal Assertions with Parametrised Propositions

Volker Stolz <vs@iist.unu.edu>
United Nations University
Institute for Software Technology (UNU-IIST)

January 26, 2007

In this work, we present an extension to our previous approach to runtime verification of a single finite path against a formula in Next-free Linear-Time Logic (LTL) *with free variables and quantification*.

We introduce *parametrised propositions* that consist of a proposition name $(p, q, \ldots)$ with arity. The payload of such a proposition occurring on a trace contains values from some *object domain* according to its arity. In a formula, a proposition contains the appropriate amount of variables, e.g. $p(X, Y)$ or $q(Z)$.

Variables get instantiated if a proposition matches during evaluation of a trace. Multiple occurrences of the same variable are permitted and work similar to Prolog: if a variable is already bound when a proposition is evaluated, both the proposition occurring in the current state and the bound variables must match.

From our experience with J-LO, the Java Logical Observer [2, 1], we found it necessary to distinguish between read and write accesses to variables, based on a static analysis of the formula. Furthermore, evaluation of uninstantiated propositions had to be considered. As interpretation (through a human) of those formulae resulted difficult and error prone due to the binding semantics, in this article we introduce a special binary *binding operator* $\dot{\rightarrow}$ that simplifies our design in the following aspects:

- simpler binding semantics
- no static analysis necessary
- more general through quantification.

The left-hand side contains a single parameterised proposition, the right-hand side a temporal parametrised formula that may refer to the variables bound in the proposition, e.g.

$$\psi := p(X) \dot{\rightarrow} \varphi(X).$$

Negation is only permitted in propositional subformulae, We call the entire construct a *binding expression*.

Furthermore, we require that every variable occurring in a parametrised formula has previously been bound through the left-hand side of a binding operator. We can thus ensure by construction that evaluation will only encounter completely instantiated propositions, i.e. propositions, where a value for every

variable is known. If the left-hand side does not match the current state during evaluation, the overall expression is evaluated to *false*.

Quantification plays a role when more than one matching proposition holds in the current state. Matching the proposition $p(X)$ against the state $\{p(1), p(3)\}$ yields two distinct bindings for variable X: $X/1$ and $X/3$. Quantifiers may only occur together with a parametrised proposition on the left-hand side of the binding operator. In a binding expression, all newly introduced variables through a proposition must also be quantified.

Additionally to the usual notion of LTL formulae augmented by quantified variables and bindings, we also permit *predicates* and *functions* over bound variables that can be used, for example, to compare values for inequality.

As an example, we consider the Lock-Order Reversal pattern [3], which captures a common error pattern where two processes repeatedly compete for two resources (locks), albeit in different order. This behaviour has the potential for a dead lock which can be detected by monitoring the order in which each process locks/unlocks the resources.

$$\begin{aligned}\Psi = \mathbf{G}\ [\forall t_i \forall l_x : \texttt{lock}(t_i, l_x) \dot{\rightarrow} ([\neg \texttt{unlock}(t_i, l_x)\ \mathbf{U}\ \exists l_{z'} : \texttt{lock}(t_i, l_{z'}) \dot{\rightarrow} l_{z'} \neq l_x] \\ \rightarrow \neg \texttt{unlock}(t_i, l_x)\ \mathbf{U}\ \exists l_z : \texttt{lock}(t_i, l_z) \dot{\rightarrow} [l_z \neq l_x \\ \wedge \forall l_y : \texttt{lock}(t_i, l_y) \dot{\rightarrow} (l_y \neq l_x \wedge G\ \neg(\exists t_j : \texttt{lock}(t_j, l_y) \dot{\rightarrow} [t_i \neq t_j \\ \wedge (\neg \texttt{unlock}(t_j, l_y)\ \mathbf{U}\ \texttt{lock}(t_j, l_x))])])]\end{aligned}$$

`lock` and `unlock` are binary propositions, binding a thread- and a lock-id, $\neq$ is a predicate.

A declarative semantics is given by expanding quantified variables through values from the finite object domain and combining them through conjunction or disjunction according to the quantifier. Operationally, evaluation of such a Temporal Assertion proceeds by means of a variant of Alternating Finite Automata, augemented with a dictionary to maintain the current bindings for each subformula. For runtime verification, we give an algorithm based on sets in disjunctive normal form that traverses the automaton in a breadth-first fashion which requires processing each state in a path exactly once and in order. It is thus suitable for online checking where an error should be detected immediately.

References

[1] E. Bodden and V. Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In W. Löwe and M. Südholt, editors, *5th International Symposium on Software Composition (SC'06)*. To be published in Lecture Notes in Computer Science, Springer, 2006.

[2] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma, editors, *Fifth Workshop on Runtime Verification (RV'05)*, volume 144 of *Electr. Notes in Theor. Comput. Sci.* Elsevier, 2005.

[3] V. Stolz and F. Huch. Runtime Verification of Concurrent Haskell Programms. In K. Havelund and G. Roşu, editors, *Proceedings of the Fourth Workshop on Runtime Verification (RV'04)*, volume 113 of *Electr. Notes in Theor. Comput. Sci.*, pages 201–216. Elsevier, 2005.

From Interaction Patterns to Aspects: a Mechanism for Systematic Runtime Monitoring

Ingolf Krüger, Massimiliano Menarini
University of California, San Diego
9500 Gilman Drive, Mail Code 0404
La Jolla, CA 92093-0404, USA
{ikrueger, mmenarini}@ucsd.edu

Michael Meisinger
Technische Universität München
Institut für Informatik
Boltzmannstr. 3, 85748 Garching, Germany
meisinge@in.tum.de

Extended Abstract

Runtime monitoring of software systems requires the insertion into runnable software of monitors that gather information on system states and their evolution. A simple approach to run time monitoring consists of modifying the software source code to implement monitoring functionalities. The very nature of such monitors, however, makes such modifications repetitive and scattered across the whole code base. Aspect oriented languages have been introduced exactly to address repetitive code changes crosscutting the code. They enable a compact representation of such code modifications. Thus, using aspects to specify such monitors seams a promising avenue.

We focus our research on distributed, loosely coupled systems. This system class is based on a well defined communication infrastructure with systems' functionalities emerging from interactions between components over this communication infrastructure. The runtime verification of such systems requires, therefore, monitoring the communications between components and verifying that the expected communication patterns are observed.

We have developed a specification technique for distributed systems based on the use of Message Sequence Charts (MSC) to capture the interaction patterns between entities. Our models are based on a thorough formal foundation and allow for consistent refinement and refactoring. The MSC graphs can, therefore, be used to capture the temporal properties of the *interaction interfaces* of the distributed system. In particular, it is possible to use the models for verification purposes by generating state machine representing the communication behavior of each node of the system. We have applied this strategy, for instance, for conformance testing of components by runtime monitoring [1], generation of executable prototypes for efficient evaluation of multiple architecture candidates ([4], [3]), and to support product-line engineering [5].

When analyzing the relationships between our interaction specification technique and aspect-oriented programming languages ([3], [4], [5]), we were able to identify many similarities between our interaction specifications and aspects. Aspects promote the compact representation of functionalities that spread across different parts of a system's source code. Similarly, our MSC descriptions compactly capture the interaction of logically or physically distributed entities in the system collaborating to provide some functionality. Then, aspects map crosscutting concerns to elements in the program code (pointcuts), whereas MSCs [3] map crosscutting *interaction* concerns to nodes in the distributed system.

These observations motivate the use of aspect-oriented languages as implementation technique for our interaction based models. Thus, we have developed M2Aspects, a code generation tool leveraging the AspectJ language and producing executable system simulations. M2Aspects translates interactions into aspects and uses weaving techniques [2] to establish the mapping between one abstract interaction specifications and low level deployment models. Aspect-orientation propagates the separation of cross-cutting concerns into aspects, maintaining a one to one mapping between models and code.

We propose to combine our interaction specification approach with aspect-oriented technologies to enable an easy modification of distributed systems implementations. We can then embed system monitors into the executable based on models, thus increasing software dependability. We

are investigating enhancements to our M2Aspects tool, resulting in the creation of run-time monitors out of MSC based interaction descriptions. The monitors leverage AspectJ to directly modify an existing Java implementation. We can compactly specify a monitor observing the protocols implemented by the system and insert it into the existing code using the AspectJ weaver without the need of complex code refactorings.

One difficulty is to match aspect language pointcuts, based on code patterns, with the phases of the protocol implemented by the code; this is, in particular, true for systems developed without run time monitoring in mind. We are exploring extensions of M2Aspects that leverage pointcuts that are generally easy to identify at the code level: message send and receive. We leverage our capability to convert interaction patterns to state machines and weave them into the system to keep explicit track of the protocol state. Those state machines can then be used to establish the right pointcuts where the code for monitoring, verifying or even modifying the interactions can be inserted. The generation of the automata, weaved into the system, is based on our algorithm to transform MSCs into state machines, presented in [7].

The use of interaction based specifications has emerged as a powerful abstraction to describe a vast set of real systems. In particular it has been identified as a distinguishing element of service oriented specifications. The notion of *service* has attracted increasing attention both in industry and academia as a mechanism to achieve coupling to address integration of large distributed systems. Service-oriented techniques have been successfully applied in ultra-large scale (ULS) systems. Examples of ULS systems include avionics, automotive, command and control, as well as telematics and public safety systems, to name just a few. In all these domains, the primary challenge to software and systems engineering is the integration of a wide variety of subsystems, their associated applications, data models and sources, as well as the corresponding processes, into a high quality system of systems under tight time-to-market, budget, security, policy, governance and other cross-cutting constraints. These requirements characteristics have led to a high demand for loosely-coupled integration architectures [6]. Therefore, the use of interaction based specifications as a starting point for runtime verification of systems has tremendous potential for increasing quality of real industrial applications.

More work is needed to have a complete and general translation to Aspects implemented in the M2Aspects tool. Moreover, experiments are currently in progress to establish the practicality and usability of this approach in practical applications. Finally, we are investigating the integration of runtime verification, based on the outlined technique, with a full service oriented development process for fail safe systems. We expect this integration to allow us to specify and integrate into existing systems, failure management code to increase the reliability of distributed systems without incurring in the risks introduced by substantial refactoring.

References

[1] J. Ahluwalia, I. Krüger, M. Meisinger, W. Phillips. Model-Based Run-Time Monitoring of End-to-End Deadlines. In Proc. of the Conference on Embedded Systems Software (EMSOFT 2005), 2005.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect Oriented Programming. Technical report, Xerox Corporation, 1997

[3] I. Krüger, G. Lee, M. Meisinger. Automating Software Architecture Exploration with M2Aspects. In Proc. of the ICSE 2006 Workshop on Scenarios and State Machines (SCESM'06) ACM Press, 2006.

[4] I. Krüger, R. Mathew, M. Meisinger. Efficient Exploration of Service-Oriented Architectures Using Aspects. In Proc. of the 28th Intl Conference on Software Engineering (ICSE 2006), ACM Press, 2006.

[5] I. Krüger, R. Mathew, M. Meisinger. From Scenarios to Aspects: Exploring Product Lines. In Proc. of the ICSE 2005 Workshop on Scenarios and State Machines (SCESM'05), ACM Press, 2005.

[6] I. Krueger, M. Meisinger, M. Menarini, S. Pasco. Rapid Systems of Systems Integration – Combining an Architecture-Centric Approach with Enterprise Service Bus Infrastructure. In Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI 2006), 2006.

[7] I. Krüger, R. Grosu, P. Scholz, M. Broy: From MSCs to Statecharts, in: Franz J. Rammig (ed.): Distributed and Parallel Embedded Systems, Kluwer Academic Publishers, 1999

ARVE: Aspect-oriented Runtime Verification Environment

Hiromasa Shin, Yusuke Endoh, and Yoshio Kataoka
System Engineering Laboratory, Corporate R&D Center, Toshiba Corporation
{hiromasa.shin, yusuke.endoh, yoshio.kataoka}@toshiba.co.jp

Categories and Subject Descriptors D.2 [*Software Engineering*]: Testing and Debugging

General Terms Aspect-oriented programming, Debugger

Keywords Runtime verification, Debugger, Optimization

1. Introduction

Software testing, including runtime verification, is essential for developing reliable software products. As is often the case with the development of industrial software products, the scale of a test program tends to be larger than that of a product program. As the test program is inherent in the product program, it is often structured in an ad hoc manner. We consider that improving modularity, reusability and portability of a test program constitutes an important issue. Since a test program is concerned with its target program in cross-cutting manner, the paradigm of aspect-oriented programming [1] is useful for improving the composition of a test program.

Even today many industrial software products, such as embedded software, are written in C/C++, and popular software development tools, which are commonly available in various platforms, are classic tools, such as symbolic debuggers and performance profilers. Meanwhile, most of aspect-oriented language are designed for Java language, or are not available in embedded platforms. Considering these circumstances in industrial software development, we have designed a practical tool named aspect-oriented runtime verification environment (ARVE). ARVE enhances the capability of a symbolic debugger by employing aspect-oriented technology. In ARVE, a test program is written in aspect-oriented script language and can be dynamically woven into a target program.

We briefly present the features, structure and evaluation results of ARVE and explain simple applications. We also explain work-in-progress and speculative future work concerning ARVE.

2. Recent work

For illustration of ARVE's functionality, we firstly present an application of ARVE to an event sequence checker. The following script *RegexpChecker* gathers events of file handling operations in target program execution, and checks whether or not the operation sequence satisfies the pattern specified in regular language. Having found the operation deviating from the specified sequence, this checker breaks the execution of the target program and dumps the execution stack.


```
import "RegexpChecker.pl";
aspect FileRegexpChecker extends RegexpChecker {
  pointcut mark() : call(^fopen$) || call(^fread$)
                 || call(^fwrite$) || call(^fclose$);
  sub new () {
    my $class = shift;
    my $self = RegexpChecker->new(
       "A-fopen[-1] (B-fread[3]|B-fwrite[3])* B-fclose[0]");
    return bless $self, $class;
  }
}
```

This ARVE script is written in the Perl-based language equipped with aspect-related syntax similar to AspectJ [2]. The concrete aspect *FileRegexpChecker* inherits the abstract aspect *RegexpChecker*, and describes the event and the pattern by overriding pointcut *mark()* and constructor *new(..)* argument. The parent aspect *RegexpChecker* is a reusable aspect, which contains the algorithm to generate DFA (Deterministic Finite Automaton) from the regular expression and to drive the DFA by invocation of advice related with pointcut *mark()*. The meaning of the terminal symbol in the regular expression is *"(after or before)-(name of joinpoint)[argument index of file handler]"*. We applied this aspect to monitor the API usage of socket handling in the server process, such as Apache and Squid, and conformed that it worked properly. In this example, the basic mechanism of aspect-oriented programming improves the modularity of the checker, and especially inheritance mechanism improves the reusability of the checker.

Figure 1 is an architecture diagram of the ARVE system consisting of ARVE script, ARVE kernel, symbolic debugger, script interpreter and target program. The symbolic debugger works as a peripheral system for the ARVE kernel, and provides the functionality of breakpoint management, the target's symbol table management, and the target's memory access for the ARVE kernel. The ARVE system utilizes the debugger's function via a clear-cut debugger interface. This interface layer ensures the independence of the ARVE kernel from a specific debugger. Any debugger satisfying this interface can work in the ARVE system. In this implementation, we adopted the debugger GDB [3], which is a popular debugger and supports many embedded platforms.

We evaluated runtime performance of the prototyped ARVE system. In a laptop PC (Dynabook TECRA 9000, Pentium-III 1.2GHz, Linux 2.4.20 and gdb 6.4), the elapsed time for a subroutine call is about 10 nanoseconds, and the elapsed time for the same call with an empty advice is about 6 microseconds. Installing an advice at each subroutine call, the program execution will take 600 times longer time. However, this situation will be extreme; many applications will use fewer advice calls than in this case. Typical overhead time is 6 microseconds, and this value will be acceptable for monitoring appropriately selected places in network communication or user interactive application.

We briefly summarize the difference from related work. ARVE uses a symbolic debugger to weave aspect into the target, and this

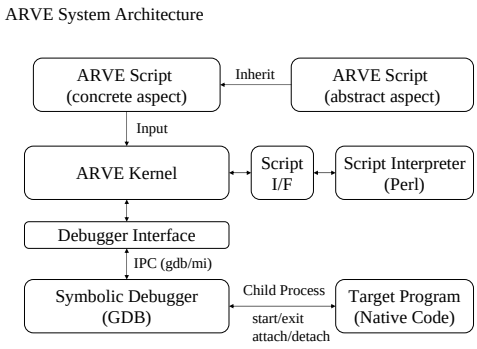

Figure 1. ARVE System Architecture

feature differs from usual dynamic weaving techniques based on JVM (Java Virtual Machine) reported in articles [4, 5]. ARVE uses script language to describe aspect, and this feature differs from the usual dynamic instrumentation techniques reported in articles [6, 7].

Compared to the usual dynamic instrumentation techniques, the ARVE approach has a disadvantage in terms of runtime efficiency; however, the ARVE approach has an advantage in terms of platform portability. The aspect of ARVE is written in script language. ARVE has a definite interface with the symbolic debugger, and can use a different symbolic debugger for each platform. Thus the script of ARVE remains independent of the platform. This feature will improve portability of a test program.

3. Work in progress

We are working on two plans. The first plan is to extend the ARVE kernel to support multi-process environments. In our experiment on an Apache server, we had difficulty in tracing many processes forked by the server. If ARVE supports an aspect among multi-processes and automatically attaches to multiple processes, the runtime verification aspect concerning IPC (Inter Process Communication) can be naturally described in a single aspect.

The second plan is to design ARVE script to check an event sequence specification written in LTL (Linear Temporal Logic). We have already prototyped the event sequence checker based on regular expression. Analogous to the case of the regular expression, the reusable abstract aspect implements the automaton, which is constructed by existing tableau construction techniques and is driven by execution event.

4. Highly speculative work

We have two plans concerning speculative future work. The first plan is to apply ARVE to an execution environment for model-based testing [8]. In order to utilize the capability of ARVE, we feel a strong need to connect ARVE usage and upstream design. In model-based testing, we can extract a test suite from the formal specification of a target. Converting the test suite to ARVE script, ARVE can execute the conformance test. Since ARVE has a debugger's capability, it can not only monitor the relation between input and output in testing, but also monitor the internal state of IUT (Implementation Under Test).

The second plan is to make ARVE and static analysis complement each other. The static analysis, such as ESP [9], has an advantage in terms of full path coverage without execution, but has a disadvantage in lack of information due to abstract interpretation. Since the advantage and disadvantage of dynamic analysis are opposite of those, the complementary combination of each analysis is expected to constitute a powerful approach. The specification language, such as ESP's OPAL, describing the finite state machine, can be viewed as an aspect-oriented automaton description language. Unifying the specification language between dynamic analysis and static analysis, will be a good starting point to make the two analysis methods complement each other.

5. Summary

We have presented a practical tool, ARVE, which enables development of a test program in script language and in aspect-oriented paradigm, and achieves independence from an underling symbolic debugger. As work in progress, an extension for multi-process support and a development for LTL-based verifying aspect are presented. As a highly speculative work, combination with model-based testing or static analysis is presented. All these approaches are aimed at improving reusability, portability and modularity of a test program in industrial software development.

References

[1] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[2] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[3] Richard M. Stallman. *Debugging With GDB: The Gnu Source-Level Debugger (9th Edition)*. Free Software Foundation, 2002.

[4] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[5] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM Press.

[6] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[7] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[8] Michael Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Towards a tool environment for model-based testing with asml. In *FATES*, pages 252–266, 2003.

[9] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

Collaborative runtime verification with tracematches

Eric Bodden[1], Laurie Hendren[1], Ondřej Lhoták[2], Nomair A. Naeem[2]

[1] McGill University, Montréal, Québec, Canada
[2] University of Waterloo, Waterloo, Ontario, Canada

In the verification community it is now widely accepted that, in particular for large programs, verification is often incomplete and hence bugs still arise in deployed code on the machines of end users. Yet, in most cases, verification code is taken out prior to deployment due to large performance penalties induced by current runtime verification approaches. Consequently, if errors do arise in a production environment, bugs are hard to find, since the available debugging information is often very limited.

In previous work on *tracematches* [1], we have shown that in many cases runtime monitoring can be made much more efficient using static analysis of the specification [2] and program under test [3]. Most often, the imposed runtime overhead can be reduced to under 10%. However, the evaluation we performed also showed that some classes of specifications and programs exist for which those optimizations do not perform as well and hence large overheads remain. According to researchers in industry [5], larger industrial companies would likely be willing to accept runtime verification in deployed code if the overhead is below 5%. Hence, additional work is required in order to make runtime verification scale even better.

In this work, we tackle this problem by applying methods of remote sampling [4] to runtime verification. Remote sampling makes use of the fact that companies which produce large pieces of software (which are usually hard to analyze) often have access to a large user base. Hence, instead of generating a program that is instrumented with runtime verification checks at all necessary places, one can generate different kinds of partial instrumentation ("probes") for each such user. A centralized server then combines results of all runs of those users. This method is generally very flexible. In particular, we see the following advantages over a complete runtime verification.

Less runtime overhead per user. The program each user runs is only partially instrumented and hence the instrumentation overhead can be kept to a moderate level.

Better coverage of relevant paths. In order for runtime verification to be complete, perfect path coverage is necessary. In general, this is nearly impossible to achieve. If instrumentation could be dynamically adapted, it could be focused on paths that are actually being executed during users' program runs.

Assigning priorities. Similarly, usage data could be used to assign priorities to bugs that are triggered by many users.

Automatic analyses. The server that receives the event data in the end can apply arbitrarily sophisticated analyses on the received data and automatically attach this information to a bug report. This is in contrast to existing error reporting systems, which are mostly operated manually.

In this work we focus on the first part, reducing the runtime overhead, and present experiments for providing such an infrastructure based on static compilation of tracematches. Since tracematches allow for per-object specifications via free variables, special attention has to be paid to object bindings. Using a flow-insensitive whole-program analysis proposed in [3], we obtain groups of related instrumentation points which need to be triggered at runtime in order to obtain a property violation. Each probe is defined as such a set of instrumentation points. We extended our compiler such that each probe is guarded by a Boolean flag whose status can be dynamically changed.

As we will show, it is safe to freely enable and disable probes while still preserving the correct tracematch semantics. In general, this approach gives up completeness, though. Hence, we explain how techniques from Liblit et al. [4] can be used to express probabilities with which a given piece of software is correct if no errors are detected. In the situation where there exist at least as many users as different probes and if probes are evenly distributed amongst those users, this probability can amount up to 100%. In those cases no precision is lost with respect to a fully instrumented program.

In order to prove the feasibility of our approach, we applied our modified compiler to some of our largest benchmarks from previous evaluations [3]. Our results show that in many cases, the instrumentation overhead can indeed be lowered from as much as 250% to less than 10% (for each user). We also identified two possible sources of problems. As mentioned in [3], under some unfortunate circumstances (imprecise points-to sets, very long-lived objects) probes can become larger than usual. Consequently, in those cases the instrumentation overhead per user might be higher. Secondly, the reconfigurable instrumentation which we statically insert may impede other static optimizations due to the introduction of a more complicated branching structure. We discuss approaches to overcoming those problems as well as the possibility of dynamic reconfiguration of probes on the level of a Java virtual machine.

References

1. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.
2. Pavel Avgustinov, Julian Tibble, Eric Bodden, Ondřej Lhoták, Laurie Hendren, Oege de Moor, Neil Ongkingco, and Ganesh Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, http://www.aspectbench.org/, 03 2006.
3. Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. Technical Report abc-2006-4, http://www.aspectbench.org/, 12 2006.
4. Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.
5. Wolfgang Grieskamp (Microsoft Research), January 2007. Personal communication.

Static and Dynamic Detection of Behavioral Conflicts between Aspects

Pascal Durr, Lodewijk Bergmans, Mehmet Aksit
University of Twente
{durr,bergmans,aksit}@ewi.utwente.nl

ABSTRACT

Aspects have been successfully promoted as a means to improve the modularization of software in the presence of crosscutting concerns. The so-called *aspect interference problem* is considered to be one of the remaining challenges of aspect-oriented software development: aspects may interfere with the behavior of the base code or other aspects. Especially interference between aspects is difficult to prevent, as this may be caused solely by the composition of aspects that behave correctly in isolation. A typical situation where this may occur is when multiple advices are applied at the same, or *shared*, join point.

In [1] we explained the problem of behavioral conflicts between aspects at shared join points. We presented an approach for the detection of behavioral conflicts that is based on a novel abstraction model for representing the behavior of advice. This model allows the expression of both primitive and complex behavior in a simple manner that is suitable for automated conflict detection. The presented approach employs a set of conflict detection rules, which can be used to detect both generic conflicts, as well as, domain- or application specific conflicts. The application of the approach to Compose*, which is an implementation of Composition Filters, demonstrates how the use of a declarative advice language can be exploited for aiding automated conflict detection.

This paper presents the need for and a possible approach to a runtime extension to the described static approach. The approach uses the declarative language of Composition Fillers. This allows us to reason efficiently about the behavior of aspects. It also enables us to detect these conflicts with minimal overhead at runtime.

An example conflict: Security vs. Logging.

We first briefly present an example of a behavioral conflict. Assume that there is a base system which uses a *Protocol* to interact with other systems. Class *Protocol* has two methods: one for transmitting, *sendData(String)* and for receiving, *receiveData(String)*. Now image that we would like to secure this protocol. To achieve this, we encrypt all outgoing messages and decrypt all incoming messages. We implement this as an encryption advice on the execution of method *sendData*. Likewise, we superimpose a decryption advice on method *receiveData*. Now imagine a second aspect which traces all the methods and possible arguments. The implementation of the tracing aspect uses a condition to dynamically determine if the current method should be traced, as tracing all the methods is not very efficient. The tracing aspect can, for instance, be used to create a stack trace of the execution within a certain package.

These two advices are superimposed on the same join point, in this case *Protocol.sendData*[1]. As the advices have to be sequentially executed, there are two possible execution orders here. Now assume that we want to ensure that no one accesses the data before it is encrypted. This constraint is violated, if the two advices are ordered in such a way that advice *tracing* is executed before advice *encryption*. We may end up with a log file which contains "sensitive" information. The resulting situation is what we call a behavioral conflict. We can make two observations, the first is that there is an ordering dependency between the aspects. If advice *trace* is executed before advice *encryption*, we might expose sensitive data. The second observation is that, although this order can be statically determined, we are unsure whether the conflicting situation will even occur at runtime, as advice *trace* is conditionally executed.

Approach.

An approach for detecting such behavioral conflicts at shared join points has been detailed in [1]. A shared join point has multiple advices superimposed on it. These are, in most AOP systems, executed sequentially. This implies an ordering between the advices, which can be (partially) specified by the aspect programmer. This ordering may or may not cause the behavioral conflict. The conflict in the example, is the case where the ordering causes the conflict. However there are conflicts, like synchronization and real-time behavior, which are independent of the chosen order.

One key observation we have made, is the fact that modelling the entire system, is not only extremely complex but it also does not model the conflict at the appropriate level of abstraction. With this we mean, that during the transformation, of behavior to read and write operations on a set of variables, we might loose important information. In our example we *encrypt* the arguments of a message to provide some level of security. Modelling this as a write on the arguments can work in some cases, however this makes expressing application specific conflict patterns hard. i.e. we do not want to consider all changes of all arguments of all messages conflicting. Also semantically, the *encrypt* operation does not change the value of the arguments, it only presents the data in a different form.

Our approach revolves around abstracting the behavior of an advice into a resource operation model. Here the resources present common or shared interactions (e.g. a semaphore). These resources are thus potential conflicting "areas". Advices interact with resources using operations. As the advices are sequentially composed at a shared join point, we can also sequentially compose the operations for each (shared) resource. After this composition, we verify whether a set of rules accepts the resulting sequence of operations

[1]Here, we only focus on join point *Protocol.sendData*, but a simular situation presents itself for join point *Protocol.receiveData*.

for that specific resource. These rules can either be conflict rules, i.e. a pattern which is not allowed to occur, or an assertion rule, i.e. a pattern which must always occur. These rules can be expressed as a regular expression or a temporal logic formula.

In [1], an instantiation of the presented model for the Composition Filters approach is shown. We adopted this approach, as the filter language is to a large extent declarative, and the compositional semantics are well-defined. This improves reasoning about the combination of multiple advice at the same join point. In addition, the filters provide encapsulation of the behavior through the use of filter types, which can be reused. However, there are elements which are filter instance specific and must be analyzed for each instance of a filter, such as the condition and matching parts of the filter. The conflict detection model can be enriched with domain or application specific information to capture more application or domain specific conflicts.

There are many steps involved in processing and analyzing a sequence of filters on a specific join point. One such step is to analyze the effects of each of the composed filters. A filter can either execute an accept action or a reject reject, given a set of conditions and a message. Next we have to determine which filter actions can be reached and whether, for example, the *target* has been read in the matching part. These actions perform the specific tasks of the filter type, e.g. the *Encrypt* action of filter type *Encryption* will encrypt the arguments. Likewise, the *Trace* action of the filter type *ParameterTracing* will trace the message. Most filter types execute the *Continue* action if the filter rejects. Imagine the following composed filter sequence on method *Protocol.sendData* in our example. The result is the following composed filter sequence:

```
1 trace : ParameterTracing = { ShouldTrace => [*.*] };
2 encrypt : Encryption = { [*.sendData] }
```

Listing 1: Composed filter sequence example.

This filter sequence can be translated to the filter execution graph in figure 1. The *italic* labels on the transitions are evalutions of the conditions (e.g. *ShouldTrace*), and the message matching, e.g. *message.sel(ector) == sendData*. The **bold** labels on the transitions show the filter actions. The underlined labels are resource-operations tuples corresponding to the evaluation of the conditions, matching parts and the filter actions.

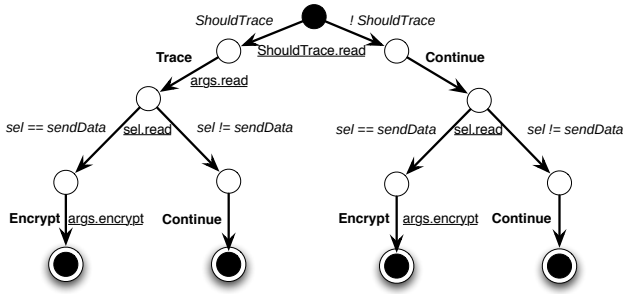

Figure 1: Filter execution graph example

From this graph we can see that in the left most path, the *arguments* are *read* before they are *encrypted*. This path thus violates the encryption requirement described in the example.

In Compose* we analyze the conflicts statically. However, it is not always possible to determine statically whether certain conflicts occur. There are three situations where dynamic verification is relevant:

1. If a program uses dynamic or conditional superimposition, and we detect a conflict in the program, we can only issue a warning at compile time. Only at runtime can we be sure that the conflict occurs.
2. Similarly, if the program uses conditional or dynamic advice execution. Here we also have to monitor the system at runtime to detect the conflict.
3. Concurrency can cause a wide variation of interleavings, including potentially conflicting sequences. This requires full monitoring of the advice at shared join points.

For the dynamic and conditional superimposition or advice execution, we can only issue a warning at compile time but we have to monitor the execution to detect the conflict at runtime. However, we can use the analysis results from the compile time to determine which paths of the composed program at a shared join point may potentially lead to a conflict.

As illustrated in figure 1, we have an internal representation of the sequence of filters at a shared join point. This representation is an execution graph, in which all the possible messages are simulated. Each end state of this graph corresponds to an unique path through the filter sequence. The graph branches if a condition is used within the filters. It also accounts for the various ways a message can flow through the filter sequence.

For each such path we know which conflict rule matches and which assertion rule rule does not match. We also know the transitions required to reach the erroneous end state. This information can be used to inject bookkeeping information at the transitions with are part of the path leading to the erroneous end state. This bookkeeping information performs the operations on the specific resources. If a, possible erroneous, final state is reached, we verify whether the conflict rules match or whether assertion rules do not match. If so, we can throw an exception, which can be handled by the user.

Conclusion.

The presented approach does not only provide feedback in an early stage of software development, i.e. while writing and compiling the aspect, it also provides an optimized way of checking whether certain conditional or dynamic conflicts actually occur at runtime. We only monitor those cases where it is known that a conflict could occur, but can not be completely statically determined. The declarative language of Composition Filters enables us to only verify those combinations which may lead to a conflict. It also enables us to reason about aspects without detailed knowledge of the base code, i.e. we only need to know the join points of the system, thus providing some form of isolated reasoning. Currently, only static verification has been implemented, in Compose*. However, we do plan to implement the proposed runtime extension in the near future.

This work has been partially carried out as part of the Ideals project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program. This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe).

References

[1] P. Durr, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In R.Chitchyan, J. Fabry, L. Bergmans, A. Nedos, and A. Rensink, editors, *Proceedings of ADI'06 Aspect, Dependencies, and Interactions Workshop*, pages 10–18. Lancaster University, Lancaster University, Jul 2006.

A Semantic-based Runtime Weaver for Dynamic Management of the Join Point History

Charlotte Herzeel, Kris Gybels, Pascal Costanza
{charlotte.herzeel, kris.gybels, pascal.costanza}@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel

January 25, 2007

1 Introduction

Although early research in aspect-oriented programming focussed on aspects that are triggered at a single join point, more recent research has evolved towards aspects that are triggered based on the occurrence of a series of join points in the execution of a program. These types of aspects were dubbed *event-based aspects*, *stateful aspects* [2] and *context-aware aspects* [10], and a number of novel pointcut languages with direct support for these kinds of aspects are currently being developed [9, 5, 1].

One of the most challenging aspects of developing an aspect language that supports these *history-based aspects*, is managing the join point history. In an ideal situation, we would keep all data about join points in memory forever, so that we could write arbitrary pointcuts over this history. However, in reality, this is not feasible. Currently a series of static analysis techniques (e.g. AspectJ [8], Alpha [7]) have been proposed where one analyzes the base code and aspect code to derive *join point shadows*, which are places in the base code that generate a join point that could possibly trigger a pointcut. Henceforth an optimal weaver can be build that omits generating unnecessary join points and hence the recorded join point history is reduced. In this paper we take a look at how the join point history can be managed at runtime so that data about join points can be deleted once it is no longer relevant for resolving pointcuts. More concretely, we discuss the weaver of the HALO language.

The logic-based language HALO provides support for writing history-based aspects that are defined in terms of temporal relations between join points. For this purpose, HALO offers a predefined set of higher-order temporal predicates – derived from temporal logic programming – for connecting pointcuts. The latter is exploited by the HALO weaver: because of the predefined set of temporal relations the weaver can – in some cases – with certainty decide whether data about a specific join point will ever (again) be used in matching a pointcut. Hence we can effectively reduce the join point history as the program runs.

2 The HALO language

HALO is an extension of Common Lisp, allowing one to express history-based aspects over a CLOS program. In addition HALO (similarly to CARMA [4] and Alpha [9]) is based on logic programming and as such pointcuts are expressed as logic queries over the join point history. The built-in pointcut predicates in HALO capture the key events in the execution of a CLOS program. For this discussion, explaining the pointcut predicate for capturing generic function calls suffices. `(gf-call ?gfName ?arguments)` captures generic function call join points and exposes the generic function's name and argument list through the logic variables `?gfName` and `?arguments`. In addition, pointcuts can be composed from other pointcuts by means of the higher-order temporal predicates. In this discussion we consider the temporal predicates: `most-recent`, `all-past` and `since`. For example, the pointcut below matches at a generic function call named `checkout` and also captures the most

recent generic function call named `buy` along with the most recent call to `checkout` before the latter.

```
(at ((gf-call 'checkout ?user1)
       (most-recent (gf-call 'checkout ?user2)
            (most-recent (gf-call 'buy ?user2 ?article2))))
       (format t "~s just bought ~s" ?user2 ?article2))
```

3 HALO weaver

The bulk of the HALO weaver consists of a query engine that matches logic facts generated for each join point against pointcuts; The latter query engine is based on the Rete forward chaining algorithm [3]. Put briefly, logic queries (or pointcuts in HALO) are represented as a network of nodes in Rete. Each such node has a *memory table* that is used to cache partial matches of the query, which are computed by propagating facts through the network. In standard Rete the two main types of nodes are filter nodes and join nodes. Filter nodes store logic facts, whereas join nodes cache conjunctions of the latter. We have extended the Rete forward chaining algorithm with novel types of join nodes to implement the different temporal predicates in HALO. In addition we extended the Rete algorithm to incorporate removing old conclusions when propagating inserts through these nodes.

As an example the Rete network for the pointcut discussed above is depicted in figure 1. A sample program run is depicted in the same figure. In addition, the figure displays tables labelled `LT` (life time): the intervals stored by these tables indicate the begin and end point for the interval during which entries in the memory tables are kept. Note that though the entries in the third filter node are removed as new entries are made, the derived conclusions are not also removed at the same time: at time 7 for example, when the entry made for `(gf-call 'buy <lotte> <dvd>)` is removed, the derived conclusion for time 5 in the first `most-recent` join node is kept. This ensures that at time 8 it can be used to match the pointcut. But this does not mean the derived conclusion is kept forever. The first `most-recent` join node is itself the input of another `most-recent` join node. The input nodes of this second join node share no variables. So the entry for time 5 in the output memory table of the first join node is removed when any other entry is made, which in this example will happen the next time a user checks out if he bought something (e.g. if the user `lotte` does another checkout).

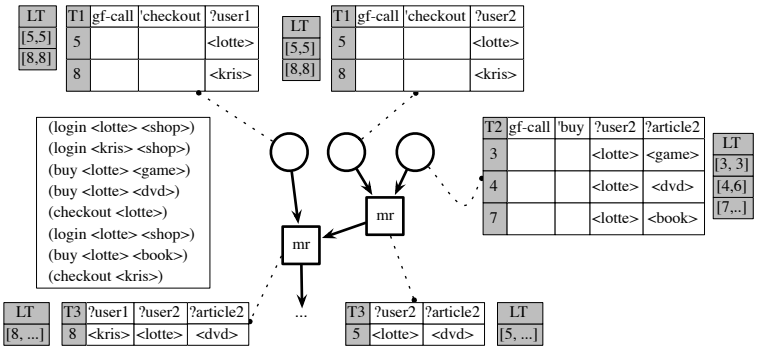

Figure 1: Garbage collection of the join point history.

4 Presentation Outline

The goal of this presentation is to discuss the possible benefits of enabling a dynamic management of the join point history, and to contrast our approach with the static analysis techniques used to avoid generating join points. For this purpose, we are currently benchmarking a web application we extended with two new features using HALO [6]. The results from this experiment will then be presented and used to validate our approach.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotk, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM Press.

[2] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Con-*

cerns, pages 170–186, London, UK, 2001. Springer-Verlag.

[3] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, September 1982.

[4] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference on Aspect-Oriented Software Development*, 2003.

[5] C. Herzeel, K. Gybels, and P. Costanza. A temporal logic language for context awareness in pointcuts. In "Workshop on Revival of Dynamic Languages", 2006.

[6] C. Herzeel, K. Gybels, and P. Costanza. Modularizing crosscuts in an e-commerce application in lisp using halo. In *submitted for review at International Lisp Conference (ILC) 2007*, 2007.

[7] K. Klose, K. Ostermann, and M. Leuschel. Partial evaluation of pointcuts. In *Proceedings of the Ninth International Symposium on Practical Aspects of Declarative Languages (PADL)*. Springer, 2007.

[8] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction (CC2003)*, 2003.

[9] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming*, 2005.

[10] É. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-aware aspects. *Lecture Notes in Computer Science, Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, 4089:227–242, 2006.

Translation Validation of System Abstractions

Jan Olaf Blech, Ina Schaefer, Arnd Poetzsch-Heffter

Software Technology Group
University of Kaiserslautern
Germany

Abstraction is a technique intensively used in the verification of large, complex or infinite-state systems. Due to inherent limitations model checkers are unable to deal with such systems directly, instead abstraction can reduce or discretize the state space. During the past decade a significant amount of research has focussed on finding abstraction methods reducing the state space sufficiently while preserving necessary precision. With abstraction algorithms getting more and more complex it is often difficult to see whether valid abstractions are generated. However, for using abstraction in model checking it has to be ensured that properties verified for the abstract system also hold in the concrete. In principle, there are two ways to guarantee correctness of abstractions: Abstraction algorithms (and their implementations!) can be verified once and for all or a tool can be build that verifies the abstraction results for each distinct run of the algorithm's implementation.

In this work, we will show how to use the second variant. For verifying a system abstraction, we set up a tool that is given a concrete system and a property to be checked. As output it produces an abstract system, a corresponding abstract property and furthermore a proof script that the abstraction is property preserving. The abstraction is correct if the proof script succesfully passes a theorem prover. Thus, the abstraction algorithm's implementation is runtime verified.

Our work towards runtime verification of system description abstractions is inspired by a translation validation [8] based approach for compilers [5]. In the area of compiler verification it has turned out that runtime verification of compilers is often the method of choice for achieving guaranteed correct compilation results.

While previously correctness of abstractions was established by showing soundness for all possible systems [2, 3], in our approach the abstraction is proved correct for a specific system and properties to be verified. Runtime verification of abstractions allows to view the tool generating abstractions as black box, although this black box may still provide basic hints on the performed abstraction. If the abstraction algorithm is replaced it is not always necessary to change the generation mechanism of the correctness proofs. Correctness proofs for distinct abstractions are usually less complex and easier to establish than proofs for a general abstraction algorithm. Also note, that in this approach the correctness of abstractions is proved formally using a theorem prover instead of a paper-and-pencil-proof.

We formalise concrete and abstract system semantics in the theorem prover Isabelle/HOL[7]. Additionally, we formulate a correctness criterion based on sim-

ulation between original and abstract system. The correctness criteria are based on property preservation of temporal logic fragments under simulation, for instance the universal fragment of CTL* is preserved under simulation [1] which can further be extended to fragments of the mu-calculus [6, 4]. The actual proof of simulation consists of two main tasks. First, a concrete simulation relation satisfying the correctness criterion has to be found. Each class of abstractions has its own requirements on the selection of this relation. Second, proof scripts to be run in Isabelle/HOL have to be generated. These are highly dependant on the original system and performed abstractions. Hence, for both steps hints provided by the abstraction algorithm are desirable. It is furthermore crucial for the verification process that the produced proof scripts do not only allow the derivation of a correct proof but can also be checked in adequate time.

The proposed technique is applied for the verification of embedded adaptive systems in the automotive sector [9]. Beside potentially unbounded data domains the size of the considered systems is huge. For model checking, these systems can in a first step be abstracted by mapping unbounded data domains to finite abstract domains to reduce and discretise the state space. We have successfully applied runtime verification of this kind of data abstraction.

For future work, we aim at extending our approach to a broader class of abstraction techniques and plan to combine abstraction with modular reasoning applying runtime verification for correctness of resulting system transformations.

References

1. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5):1512–1542, September 1994.
2. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of POPL*, pages 238–252. ACM Press, January 1977.
3. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. pages 269–282. ACM Press, January 1979.
4. Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.
5. Marek Jezry Gawkowski, Jan Olaf Blech, and Arnd Poetzsch-Heffter. Certifying Compilers based on Formal Translation Contracts. Technical Report 355-06, Technische Universität Kaiserslautern, November 2006.
6. Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
7. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
8. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384, 1998.
9. I. Schaefer and A. Poetzsch-Heffter. Using Abstraction in Modular Verification of Synchronous Adaptive Systems. In *Proc. of "Workshop on Trustworthy Software", Saarbrücken, Germany, May 18-19*, 2006.

Runtime Checking for Program Verification Systems

Karen Zee Viktor Kuncak Martin Rinard
MIT, CSAIL
Cambridge, USA
{kkz,vkuncak,rinard}@csail.mit.edu

Abstract

One of the goals of program verification is to show that a program conforms to a specification written in a formal logic. Oftentimes, this process is hampered by errors in both the program and the specification. The time spent in identifying and eliminating these errors can even dominate the final verification effort. A runtime checker that can evaluate formal specifications can be extremely useful for quickly identifying such errors. Such a checker also enables verification approaches that combine static and dynamic program analyses. Finally, the underlying techniques are also useful for executing expressive high-level declarative languages.

This paper describes the run-time checker we are developing in the context of the Jahob verification system. One of the challenges in building a runtime checker for a program verification system is that the language of invariants and assertions is designed for simplicity of semantics and tractability of proofs, and not for run-time checking. Some of the more challenging constructs include existential and universal quantification, set comprehension, specification variables, and formulas that refer to past program states. In this paper, we describe how we handle these constructs in our runtime checker, and describe several directions for future work.

1. Introduction

This paper describes a run-time checker we are developing in the context of the Jahob verification system. The primary goals of this run-time checker are debugging specifications and the program and using run-time information in loop invariant inference.

Jahob [5, 18] is a program verification system for an imperative, sequential, memory-safe language that is a subset of Java. [1] Specifications are written as special comments within the source code, so developers can compile and run programs using standard Java interpreters and runtimes. Jahob specifications are written as formulas in higher-order logic (HOL), using the syntax of the input language to the Isabelle proof assistant [20].

Jahob specifications include declarations and definitions of specification variables (similar to model fields in JML), data structure invariants, procedure pre- and postconditions, as well as assertions. Specification variables are abstract fields defined by the programmer that can be referenced in the invariant, pre- and postconditions, and assertions. In the standard program verification usage of Jahob, the data structure invariants, pre- and postconditions, and assertions are guaranteed to hold using a combination of static analysis and theorem proving. Our runtime checker ensures that these properties hold by evaluating them dynamically.

Contributions. This abstract presents the current state of our runtime checker for Jahob. The checker evaluates a subset of higher-order logic formulas containing quantifiers, set comprehensions, integer and object expressions, sets, and relations. Among the interesting features of our checker is the evaluation of certain expressions that denote infinite sets, and the evaluation of formulas that refer to old values of fields of an unbounded number of objects.

[1] Jahob's implementation language does not support reflection, dynamic class loading, multi-threading, exceptions, packages, subclassing, or any Java 1.5 features.

2. Quantifiers and set comprehensions

While Jahob's specifications are written in HOL, for practicality purposes we restrict the runtime checker to support only first-order quantification. Even with first-order quantification, however, it is possible to write formulas that cannot be executed, as is the case when the domain of the quantifier is unbounded. Consider, for example, the formula $\forall x : T.P(x)$. Note that x refers not only to all objects of type T in the heap, but to all possible objects of type T, which would be highly impractical to compute. Therefore, in most cases, the runtime checker checks only those quantified formulas where the domain of the quantification is bounded. For integers, this means that quantification must be restricted to a range of integers. For objects, Jahob has a built-in notion of the set of allocated objects, so that quantification over all allocated objects of type T is written $\forall x : T.x : AllocatedObjects \longrightarrow P(x)$. The same applies to set comprehensions, which also need to be confined to a bounded domain in order to be evaluated. (There is an interesting case in which the runtime checker can handle even unbounded quantification, which we explain in the following section.)

Even bounded domains, however, may be large, and we would like to avoid considering all objects in the heap if at all possible. For example, in the formula $\forall x : T_x \forall y : T_y.x : AllocatedObjects \wedge y : AllocatedObjects \wedge x.next = y \longrightarrow P(x, y)$, the quantified variable y is introduced for the purposes of naming and can be easily evaluated without enumerating all elements of the heap. The runtime checker handles these cases by searching into the body of quantified formulas, through conjunctions and implications, to determine if the bound variable is defined by an equality. If so, we can evaluate the body of the formula without having to enumerate a large number of objects. While it may be possible to write the same formula without introducing a quantified variable, being able to do so may not only make a specification easier to understand, it can also make its evaluation more efficient. If the bound variable appears more than once in the body of the formula, the introduction of the quantified variable identifies a common subexpression that is essentially being lifted, so that the runtime checker need only evaluate it once.

3. Specification variables

Jahob supports two types of specification variables: standard specification variables and ghost variables. These are sometimes referred to as model fields and ghost fields, respectively, as in JML [19].

A standard specification variable is given by a formula that defines it in terms of the concrete state of the program. When the runtime checker evaluates a formula that refers to a standard specification variable, it evaluates the formula that defines the specification variable in the context of the current program state.

A ghost variable, on the other hand, is updated by the programmer using special comments in the code. They behave very much like normal variables in the program. In general, they are treated similarly by the runtime checker, though in addition to standard program types such as booleans, integers, and objects, the runtime checker also supports ghost variables of types tuple and set.

When ghost variables are updated, the right-hand side of the assignment statement consists of a formula that the runtime checker evaluates to produce the new value of the ghost variable. It then stores the resulting value in the same way as it would for the assignment of a normal program variable. This formula is a standard Jahob formula and may contain quantifiers, set comprehensions, set operations, and other constructs not typically available in Java assignment statements.

Since ghost variables of type set are allowed, it is also possible to write the following code:

```
//: private ghost specvar X :: int set;
int y = 0;

//: X := {z. z > 0};
//: assert y ~: X;
y = y + 1;
//: assert y : X;
```

The above code is an interesting case because the ghost variable x is assigned to the value of an unbounded set. The runtime checker handles this case by deferring the evaluation of x until it reaches the assert statements. It then applies formula simplifications that eliminate the set comprehension. Of course, the runtime checker uses the same simplifications when presented with a formula $y \in \{z.z > 0\}$, but the above case is an interesting example of being able to check formulas that one might not expect to be able to check.

4. The old construct

The old construct is common to most program specification languages for referring to the value of an expression in an earlier state of the program. In Jahob, an old expression refers to the value of the enclosed expression as evaluated on entry to the current procedure. Unlike the old construct in JML [19], which is syntactically restricted so that an old expression can be fully evaluated in the procedure pre-state, old expressions in Jahob are not restricted in this way. While this makes the Jahob specification language more expressive, it also makes it necessary for a runtime checker to access past program state in order to evaluate such expressions.

One simple but inefficient method of providing the checker access to past program state would be to snapshot the heap before each procedure invocation. Unfortunately, this approach is unlikely to be practical in terms of memory consumption; the memory overhead would be a product of the size of the heap and the depth of the call stack.

Instead, the runtime checker obtains access to the pre-state by means of a recovery cache (also known as a recursive cache) [14] that keeps track of the original values of modified heap locations. It is implemented as a stack that behaves as follows. On entry to a procedure, the runtime checker pushes a new, empty frame onto the stack. When a write occurs, the checker notes the memory address of the write, as well as the original value of the location before the write. Subsequent writes to the same address do not require an update to the frame. When the checker needs to evaluate an old expression, it simply looks up the necessary values in the topmost frame. If it does not find a value there, that means that the heap location was not changed, and that the current value is also the old value.

There are several features of this solution worth noting. First, it takes advantage of the fact that we need only know the state of the heap on procedure entry, and not the state of any intermediate heaps between procedure entry and the assertion or invariant to be evaluated. Also, where the state of a variable is unchanged, the old value resides in the heap, so that reads do not incur a performance penalty excepting reads of old values. Finally, one of the ideas underlying this solution is that we expect the amount of memory required to keep track of the initial writes to be small relative to the size of the heap. While there is a trade-off between memory and performance—there is now a performance penalty for each write—the overhead is greatest for initial writes, and less for subsequent writes to the same location.

5. Related Work

Run-time assertion checking has a long history [9]. Among the closest systems for run-time checking in the context of static verification system are tools based on the Java Modeling Language (JML) and the Spec# system [2].

JML [19] is a language for writing specifications of Java programs. Tools are available both for checking JML specifications at runtime and for verifying statically that a program conforms to its JML specification [6]. As such, the work on JML shares at least one of the goals of the work in this paper—that of being able to use a runtime checker to aid in the process of verifying programs with respect to their specifications. The JML compiler, jmlc [8], is the primary runtime assertion checking tool for JML. It compiles JML-annotated Java programs into bytecode that also includes instructions for checking JML invariants, pre- and post-conditions, and assertions. Other assertion tools for JML include Jass [3] and jmle [17].

One of the goals in the design of JML was to produce a specification language that was Java-like, to make it easier for software engineers to write JML specifications. It also makes JML specifications easier to execute. Jahob, on the other hand, is first and foremost a program verification system, and, as such, uses an expressive logic as its specification language. The advantage of this design is that the semantics of the specifications is clear, and the verification conditions generated by the system can easily be traced back to the relevant portions of the specification, which is very helpful in the proof process.

Spec# is another system [2] for which both runtime checking and program verification tools are available. Spec# is a superset of C# and includes a specification language. The Spec# system compiles its specifications into inline checks, which may also be verified using the Boogie verifier [1]. The Spec# specifications that we are aware of do not contain set comprehensions and transitive closure expressions.

We are not aware of any techniques used to execute such specifications in the context of programming language run-time checking systems. Techniques for checking constraints on databases [4, 12, 13,15,21,22] contain relevant techniques, but use simpler specification specification languages and are optimized for particular classes of checks.

To evaluate old expressions in our specifications, we use a recovery cache, or recursive cache, a technique from fault-tolerant computing [14]. Fault-tolerant systems use recovery caches to restore the program state to a previous state in the presence of a failure.

6. Conclusions and Future Work

The Jahob run-time checker is currently built as an interpreter and is meant for debugging and analysis purposes as opposed to the instrumentation of large programs. Among the main directions for future work are compilation of run-time checks [7, 10] to enable checking of the assertions that were not proved statically [11], and combination with a constraint solver to enable modular run-time checking [16].

References

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. 4th Intl. Sym. on Formal Methods for Components and Objects*, 2005.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[3] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass–Java with assertions. In *Proc. 1st Workshop on Runtime Verification*, volume 55 of *ENTCS*, pages 103–117, 2001.

[4] P. A. Bernstein and B. T. Blaustein. Fast methods for testing quantified relational calculus assertions. In *Proc. 1982 ACM SIGMOD intl. conf. on Management of data*, pages 39–50. ACM Press, 1982.

[5] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard. Using first-order theorem provers in a data structure verification system. In *Proc. 8th Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, 2007.

[6] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, March 2003.

[7] F. Chen, M. d'Amorim, and G. Rosu. Checking and correcting behaviors of java programs at runtime with java-mop. *ENTCS*, 144(4):3–20, 2006.

[8] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, April 2003.

[9] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Soft. Eng. Notes*, 31(3):25–37, 2006.

[10] B. Demsky, C. Cadar, D. Roy, and M. C. Rinard. Efficient specification-assisted error localization. 2004.

[11] C. Flanagan. Hybrid type checking. In *Proc. 33rd Annual ACM Sym. on the Principles of Programming Languages*, pages 245–256, 2006.

[12] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Eng.*, 9(3):508–511, 1997.

[13] L. J. Henschen, W. McCune, and S. A. Naqvi. Compiling constraint-checking programs from first-order formulas. In H. Gallaire, J.-M. Nicolas, and J. Minker, editors, *Advances in Data Base Theory, Proc. of the Workshop on Logical Data Bases*, volume 2, pages 145–169, 1984.

[14] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Proc. Intl. Sym. on Operating Systems*, volume 16 of *LNCS*, pages 171–187, 1974.

[15] H. V. Jagadish and X. Qian. Integrity maintenance in object-oriented databases. In *Proc. 18th Conf. on Very Large Data Bases*, 1992.

[16] S. Khurshid and D. Marinov. TestEra: Specification-based testing of java programs using SAT. *Autom. Soft. Eng.*, 11(4):403–434, 2004.

[17] B. Krause and T. Wahls. jmle: A tool for executing JML specifications via constraint programming. In *Proc. 11th Intl. Workshop on Formal Methods for Industrial Critical Systems*, volume 4346 of *LNCS*, pages 293–296, 2007.

[18] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[19] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, December 2006.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[21] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J.-M. Nicolas, and J. Minker, editors, *Advances in Data Base Theory, Proceedings of the Workshop on Logical Data Bases*, volume 2, pages 171–209, 1984.

[22] X. Qian and G. Wiederhold. Knowledge-based integrity constraint validation. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *Proc. 12th Intl. Conf. on Very Large Data Bases*, pages 3–12, August 1986.

Rule systems for run-time monitoring: from EAGLE to RULER

Howard Barringer, David Rydeheard*
EMAIL: {Howard.Barringer, David.Rydeheard}@manchester.ac.uk

Klaus Havelund†
EMAIL: Klaus.Havelund@jpl.nasa.gov

January 26, 2007

Summary

EAGLE was introduced in [2] as a general purpose rule-based temporal logic for specifying run-time monitors. A novel and relatively efficient interpretative trace-checking scheme via stepwise transformation of an EAGLE monitoring formula was defined and implemented. However, application in real-world examples has shown efficiency weaknesses, especially those associated with large-scale symbolic formula manipulation. For this presentation, first we reflect briefly on the strengths and weaknesses of EAGLE and then we introduce RULER, a primitive conditional rule-based system, which can be more efficiently implemented for run-time checking, and into which one can compile various temporal logics used for run-time verification.

Background and motivation

A plethora of logics have been used for the specification of behavioural system properties that can be dynamically checked either on-line throughout an execution of the system or off-line over an execution trace of the system. Some form of linear-time temporal logic usually forms the basis for the specification logic. This large variety of logics prompted the search for a small and general framework for defining monitoring logics, which would be powerful enough to capture most of the existing logics, thus supporting future and past time logics, interval logics, extended regular expressions, state machines, real-time and data constraints, and stochastic behaviour. The framework should support the definition of new logics easily and should support the monitoring of programs with their complex program states. EAGLE was the result.

The EAGLE logic is a restricted first order, fixed-point, linear-time temporal logic with chop (concatenation) over finite traces. As such, the logic is highly expressive and, not surprisingly, EAGLE's satisfiability (validity) problem is undecidable; checking satisfiability in a given model, however, is decidable and that is what's required for run-time verification. The syntax and semantics of EAGLE is succinct. There are four primitive temporal operators: $\bigcirc$ — next, $\odot$ — previously, $\cdot$ — concatenation, and ; — chop (overlapping concatenation, or sequential composition). Temporal equations can be used to define schema for temporal formulae, where the temporal predicates may be parameterized by data as well as by EAGLE formulas. The usual boolean logical connectives exist. For example, the linear-time $\Box$, $\mathcal{U}$ and $\mathcal{S}$ temporal operators can be introduced through the following equational definitions.

$$\textbf{max } \texttt{Always}(\textbf{Form } F) = F \wedge \bigcirc \texttt{Always}(F)$$

$$\textbf{min } \texttt{Until}(\textbf{Form } F_1, \textbf{Form } F_2) = F_2 \vee (F_1 \wedge \bigcirc \texttt{Until}(F_1, F_2))$$

$$\textbf{min } \texttt{Since}(\textbf{Form } F_1, \textbf{Form } F_2) = (F_2 \vee (F_1 \wedge \odot \texttt{Since}(F_1, F_2)))$$

The qualifiers **max** and **min** indicate the positive and, respectively, negative interpretation that is to be given to the associated temporal predicate at trace boundaries — corresponding to maximal and minimal solutions to the equations. Thus $\bigcirc\texttt{Always}(p)$ is defined to be true in the last state of a given trace, whereas $\bigcirc\texttt{Until}(p, q)$ is false in the last state.

Even without data parametrization, the primitive concatenation temporal operators in conjunction with the recursively defined temporal predicates takes the logic into the world of context-free expressivity. Parametrization of temporal predicates by data values allows us to define real-time and stochastic logical operators.

We will reflect on two related questions: Is EAGLE too expressive for run-time monitoring? If not, is EAGLE expressive enough? For example, there are arguments to use deterministic versions of temporal concatenation and chop for run-time monitoring — and there are several different forms of deterministic cut, e.g. left minimal, left maximal, right minimal, right maximal, etc..

What can be said about the computational effectiveness of algorithms for EAGLE trace-checking? Firstly, trace-checking of full EAGLE can be under-

*School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK

†Columbus Technologies, Laboratory for Reliable Software, NASA's Jet Propulsion Laboratory, Pasadena, CA 91109, USA

taken on a state-by-state basis, even though the logic has the same temporal expressiveness over the past as over the future; basically, our published trace-check algorithm maintains sufficient knowledge about the past in the evolving monitor formulas. Unfortunately, given the presence of data arguments in temporal predicates, an explosion in the size of the evolving monitor formula may occur.

What was clear to us at the time was that there were some practically useful and efficiently executable subsets of EAGLE. One such fragment for which we computed complexity results was the LTL (past and future) fragment of EAGLE [3]. Despite the nice features of EAGLE, we still believed we should continue to search for a powerful and simpler "core" logic, one that is easy and efficient to evaluate for monitoring purposes.

Introducing RULER

The EAGLE trace-checking algorithm is essentially interpretative. Given a monitor formula and an input state, the trace-checker computes a new monitor formula that will need to hold in the next state for the original monitor formula to hold on the current input; recursively defined temporal predicates are replaced by their definitions and separated into what has to hold now and in the future. Considerable formula rewriting, i.e. data structure manipulation, is required. The question thus arises: what compilation strategy might be possible in order to optimize the interpretation process? Perhaps some form of predicated automata can be compiled. Similar issues arose when interpretation improvements were being sought for the executable logic METATEM [1]. Fisher's separated normal form was developed [4], leading to improved temporal resolution-based theorem-proving techniques.

As an experiment, we have constructed a simple rule system into which one can compile various forms of linear-time temporal logic. The rules bear a strong resemblance to the step rules used in graph-based temporal resolution. Let us give a flavour for the propositional case of RULER. Let the letters a, b, c, etc., denote propositional atoms that can be evaluated in a given input state, and the letters r_1, r_2, r_3, etc., denote rule names, which in turn are associated with conditional monitoring step rules. A rule name is also treated as a propositional atom. Rule definitions are of the form:

$$\text{ruleName} : \text{antecedent} \multimap \text{consequent}$$

where the antecedent is a conjunctive list of atoms, and the consequent is a disjunctive list of conjunctive lists of atoms. Here's an example of a set of rules representing the temporal monitoring formula $\Box((\bigodot(a \,\mathcal{S}\, b)) \Rightarrow \bigcirc(a \vee \bigcirc c))$, assuming we have r_0, r_1 and r_3 initially active.

$$\begin{array}{lll} r_0: & \multimap & r_0, r_1, r_3 \\ r_1: & b \multimap & r_2 \\ r_2: & a \multimap & r_2 \end{array} \quad \Big\| \quad \begin{array}{lll} r_3: & r_2 \multimap & a \,|\, r_4 \\ r_4: & \multimap & c \end{array}$$

The evaluation of a rule name in a state determines the associated rule's activity status. Only active rules are applied, and the consequent of a rule is applied only if the rule's antecedent holds (an empty antecedent is always true). Thus when the rule r_0 is applied, the next rule activation state must contain rules r_0, r_1 and r_3. The rule r_1 requires, however, that the atom b is true in order for the consequent to apply in the next activation state. The rule r_3 has an antecedent of r_2, which means that r_2 must be a currently active rule in order for the consequent of r_3 to be applied. The latter gives a choice: the next state must have atom a true or rule r_4 must be active. Monitoring a sequence of states with such rule sets proceeds as follows.

```
create an initial set of initial rule
                          activation states
WHILE observations exist DO
  obtain next observation state
  merge observation state with the set of
                     rule activation states
  raise monitoring exception if there's
                               total conflict
  for each of the current merged states,
    apply activated rules to generate a
              successor set of activation states
  union successor sets to form the new frontier
                       of rule activation states
OD
```

The merge of observation state with the set of rule activation states results in a set of consistent rule activation sets. If no consistent sets results, we say a total conflict with given rule set has occurred, i.e. the observation trace has failed to satisfy the given rule set. For rule set satisfaction, we need to state which rule names are allowed to be active once the whole observation trace has been monitored — similar to **max** and **min** in EAGLE. Whilst we can't show any details of this working, we assert that this primitive rule system can be more efficiently executed than the direct interpretation of EAGLE.

Naturally, our full paper will provide semantic details for RULER and translations from various temporal logic subsets. In addition, we will discuss other variations on these primitive rules, such as universality (in the above example, rule r_0 acted as a generator for r_1 and r_3), interpretations for negation of rules (forced non activation) and giving rules priorities for use in conflict resolution.

A rather fuller bibliography will be provided in the full paper!

References

[1] H. Barringer, M. Fisher, D. Gabbay, R. Owens and M. Reynolds. *The Imperative Future: Principles of Executable Temporal Logic*. Research Studies Press. 1996.

[2] H. Barringer, A. Goldberg, K. Havelund and K. Sen. Rule-Based Runtime Verification. Proceedings of the *VMCAI'04, 5th International Conference on Verification, Model Checking and Abstract interpretation*, Venice. Volume 2937, Lecture Notes in Computer Science, Springer-Verlag, 2004. 2004.

[3] H. Barringer, A. Goldberg, K. Havelund and K. Sen. Run-time Monitoring in Eagle. Proceedings of *PADTAD '04, Santa Fe, New Mexico*, IEEE Computer Society, IDPDS'04, Volume 17, Number 17, pp 264b, 2004.

[4] M.D. Fisher. A Normal Form for Temporal Logics and its Applications in Theorem-Proving and Execution. Journal of Logic and Computation, Volume 7, Number 4, pp 429-456, 1997.

The good, the bad, and the ugly, but how ugly is ugly?

Andreas Bauer, Martin Leucker, and Christian Schallhart

Institut für Informatik, Technische Universität München

Abstract. When monitoring a system w. r. t. a property defined in some temporal logic, e. g., LTL, a major concern is to settle with an adequate interpretation of observable system events; that is, models of temporal logic formulae are usually infinite streams of events, whereas at runtime only prefixes are available.
This work defines a four-valued semantics for LTL over finite traces, which extends the classical semantics, and allows to infere whether a system behaves (1) according to the monitored property, (2) violates the property, (3) will possibly violate the property in the future, or (4) will possibly conform to the property in the future, if the system has stabilised. Notably, (1) and (2) correspond to the classical semantics of LTL, whereas (3) and (4) are chosen whenever an observed system behaviour has not yet lead to a violation or acceptance of the monitored property.
This logic called FLTL seems to correspond with the semantics realised by the Temporal Rover and has, to the best of our knowledge, not been formally captured elsewhere. We further present a monitor construction for FLTL properties.

1 Introduction

While the syntax and semantics of LTL on infinite traces is well accepted in the literature, there is no consensus on defining LTL over finite strings. Several versions of a *two-valued* semantics for LTL on finite strings have been proposed. For instance, Eisner et al. give a good overview on the topic [EFH$^+$03]. Further, in [BLS06], a three-valued semantics is proposed which extends the classical semantics over finite traces in a natural manner: a property is $true$, respectively $false$, w. r. t. a finite observation, iff the observation is either a satisfying prefix, respectively violating prefix, of all possible infinite extensions; otherwise, the observation is said to be inconclusive, and the property assigned a ?. This scheme coincides well with the notion of *safety* (e. g., Gp—always p) and *co-safety* (e. g., Fp—eventually p) properties, since these are either finitely refutable or satisfiable.

However, monitoring a system w. r. t. a safety property that does, in fact, never exhibit violating behaviour, results in infinitely many inconclusive results from the monitor, likewise with co-safety properties. Further, when monitoring a *liveness* property [AS84] that is not co-safety, i. e., finitely satisfiable, then neither the violation nor the satisfaction of the property can be determined using a finite stream of observations, and not much is said about the possible future.

Contribution. In this work, we submit the idea that an inconclusive result of a monitor should be more detailed, allowing to draw conclusions what the future may hold for a system w. r. t. a trace seen so far and the type of property being monitored; that is, we define a four-valued semantics for LTL that not only results in either $true$, $false$, or ?, but yields *possibly true* and *possibly false* whenever the system's behaviour so far is not conclusive in the strictly Boolean sense. We call the resulting logic *Finite Linear Temporal Logic* (FLTL).

Further, we have defined a translation from formulae in FLTL to Mealy machines, which then form a suitable foundation for runtime verification, in that the output alphabet of the automata corresponds to the four truth values sketched above.

2 FLTL at a glance

As discussed in [MP95], the difficulty for an LTL semantics over finite strings lies in the next-state operator X. Given a finite string $u = a_0 \dots a_{n-1}$ of length n, it is unclear whether $u, n-1 \models X\varphi$ holds. Hence, a first axiom of a sound truncated path semantics would be to require that

– *$X\varphi$ means there exists a next state and this state satisfies φ* **(∃X)**

which we term the *existential-next view*, abbreviated by (∃**X**). Consequently, the above example is $false$, as there is no next state. A second axiom we consider essential is that negation, indeed, expresses that a formula's truth value is complemented, formulated as

– *a formula and its negation yield complementary truth values.* **(¬=C)**

Then, however, a negated next-state formula should be $true$. This, however, conflicts the classical equivalence $\neg X\varphi \equiv X\neg\varphi$, which can no longer hold on finite strings (unless $true$ equals $false$). It is, therefore, helpful to distinguish a *strong* or *existential* (denoted by X) and a *weak* or *universal* version (denoted by $\bar{X}$) of the next-state operator.

This view is meaningful in a setting, in which we are faced with only *maximal* traces. In runtime verification, however, we are given a *prefix* of an infinite trace. Therefore, it is clear that there will be a next state, but not known *what* it will be.

It can also be argued (see, e. g., [HR02]) that the finally operator F is of existential nature, as some property should finally hold, while the globally operator G is of a universal character, in a sense that something should hold in every position of a path. Accordingly, one can argue that $F\varphi$ should evaluate to $false$ if φ does not hold in the currently known prefix, while $G\varphi$ should be $true$, if φ is not violated in the currently known prefix, and in both cases nothing is known about the successor states.

In LTL, it holds that $F\varphi \equiv \varphi \vee XF\varphi$, as well as, $G\varphi \equiv \varphi \wedge XG\varphi$. Consequently, $XF\varphi$ should be $false$, if no subsequent state exists, while $XG\varphi$ should be $true$ in the same situation. This contradiction is elegantly solved by having the existential as well as the universal version of the next-state operator, opening the possibility of having the equivalence $F\varphi \equiv \varphi \vee XF\varphi$ and $G\varphi \equiv \varphi \wedge \bar{X}G\varphi$.

Therefore, for runtime verification, we postulate two further axioms. The first can be stated as

– *say true or false only, if the future does not matter.* **(Sound)**

The string a (of length 1) clearly satisfies the proposition p iff $p \in a$. While, understanding a as a prefix of an infinite string, the value of $X\varphi$ is of less certainty, as the successor state of a is not known. Choosing either true or false (depending on whether to understand X strongly or weakly) would diminish the qualitative difference of the knowledge on p and $X\varphi$ based on the string a. Therefore, we require a semantics to yield four values: *true*, *possibly true*, *possibly false*, and *false*. Roughly, $true$ and $false$ are used for Boolean combinations of propositions and *possibly true* and *possibly false* for statements for which the future is important.

Actually, this view seems to be already adopted in the runtime verification tool Temporal Rover [Dru00]. However, no formal semantics of Rover's employed LTL is available. Note that identifying *possible true* and $true$ as $true$, respectively *possible false* and $false$ as $false$, yields the finite trace semantics as proposed in [MP95].

So far, we have disregarded a further issue that we consider important. When considering $X\varphi$ in the last state of a finite string u, there is no reason to go for false (or possibly false), if every possible continuation of u satisfies φ. A trivial example would be $X\,true$. While every single letter extension of u would make $X\,true$ true in u's last position, the semantics discussed so far would come up with false or possibly false. Therefore, we require that

– *if the future does not matter, say true or false.* **(Precise)**

FLTL captures the ideas formulated as **(∃X)**, **(¬=C)**, **(Sound)**, and **(Precise)**, and can be efficiently translated into Mealy machines, whose output alphabet corresponds to the four truth values.

References

[AS84] Bowen Alpern and Fred B. Schneider. Defining liveness. Technical report, Ithaca, NY, USA, 1984.

[BLS06] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 4337 of *LNCS*. Springer, December 2006.

[Dru00] Doron Drusinsky. The temporal rover and the atg rover. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.

[EFH+03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *CAV03*, volume 2725 of *LNCS*, pages 27–39, Boulder, CO, USA, July 2003. Springer.

[HR02] Klaus Havelund and Grigore Rosu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.

Towards a Tool for Generating Aspects from MEDL and PEDL Specifications for Runtime Verification

Omar Ochoa, Irbis Gallegos, Steve Roach, Ann Gates
Department of Computer Science
The University of Texas at El Paso

In this paper, we describe an Aspect Oriented extension to the verification tool Java Monitoring and Checking (Java-MaC) [1]. This approach generates AspectJ aspects from Java-MaC specifications. We then use these aspects to monitor the program during execution [2]. To demonstrate the described approach, we apply it to a "benchmark" from formal methods research [3], a safety-critical railroad crossing system composed of a train, a gate and a controller. In this system, the gate must be down while the train is crossing and up when no train is crossing.

The Java-MaC framework allows users to specify system states to be monitored, define high-level events based on run-time system states, and describe correctness properties in terms of high-level events. The framework uses a runtime component called a *filter* to track the collection of probes inserted into the target program and a separate runtime component called an *event recognizer* to detect events from the state information received from the filter. The Meta-Event Definition Language (MEDL) is based on an extension of linear-time temporal logic and is used to express a large subset of safety properties of systems, including real-time properties such as "when a train is crossing, the gate is down". The Primitive Event Definition Language (PEDL) is used to describe events and conditions in terms of system objects such as methods and variables. PEDL specifications define the events recognized by the event recognizer, and these event definitions are used to automatically instrument the original program. The event recognizer emits event streams to the run-time checker, which verifies the sequence of events with respect to the specified properties [4].

Aspect-Oriented Programming (AOP) aids programmers in the encapsulation of cross-cutting concerns, i.e., specific requirements that span different modules in a system and that cannot be modularized into one component. Aspects can include fields and methods, which are merged with classes by a program called a *weaver*. Aspect weaving can occur at the source code level, at post compilation, or at class-load time [5, 6]. Aspects provide the benefit of good modularity: code simplicity, ease of development and maintenance, and potential for reuse [7]. AspectJ [8] is an AOP implementation for the Java programming language. A *join point* is a place in the code where additional behavior is required. A *pointcut* is a specification of a set of join points. There are two types of pointcuts: primitive and user defined. User-defined pointcuts are Boolean combinations of primitive pointcuts. Pointcuts may match a method invocation at either the call site or the method site, at an assignment or read from a field, or at a point where some condition holds. For example, one could verify if variable *x* is updated by using the construct: *pointcut checkx() : set(int Class.x)*. Where *checkx()* identifies the aspect, *set()* recognizes when the specified non-private field is updated, and *int Class.x* specifies field *x* in class *Class* as the field of interest. The behavior of the program can be changed at each join point by specifying a construct called *advice*, which is code to be executed at a join point.

Since primitive events in PEDL correspond to transfer of control between methods or

assignments to variables, PEDL events represent pointcuts in a program. MEDL properties correspond to safety requirements, or the advice for each pointcut. While Java-MaC provides for runtime verification of stand-alone applications, it requires full access to the source code of the application. Aspect orientation provides a way to weave aspects without having to access the source code, thus providing a black box approach to instrumentation and monitoring.

Our goal is to automatically generate AspectJ aspects from MEDL and PEDL specifications. In order to generate aspects, we define a one-way mapping from MEDL and PEDL grammars to the AspectJ grammar. MEDL and PEDL specifications are used to identify properties to be monitored and instrumentation locations. An aspect is created and is woven into either the source code or the byte code. Depending on the needs of the user, the aspect-enhanced code may monitor and detect violations, it may emit data to the event recognizer, or it may emit an event stream to the runtime checker. This allows the Java-MaC architecture to be used in emerging technologies such as web and grid services.

The MEDL and PEDL files for the railroad crossing example were defined as described in [4]. The PEDL file contained the following events: *startIC* occurs when a train reaches the crossing; *endIC* occurs when the last train passes the crossing; *startGD* occurs when the gate is closed; and *endGD* occurs when the gate starts to rise. The MEDL file contained the properties *IC*, which means a train is crossing, and *GD*, which means a gate is down. These conditions are represented as *Cond IC= [startIC, endIC]* and *Cond GD= [startGD, endGD]*, with the safety condition *safeRRC= !IC || GD*. The aspect generated from this specification consists of the safety condition *safeRRC= !IC || GD*. Two pointcuts are generated to monitor each part of the condition. Pointcut *IC* is triggered when *train_x + train_length > cross_x && train_x <= cross_x + cross_length*, which represents the train crossing. Pointcut *GD* is triggered after *Gate.gd()* is executed, but before *Gate.gu()* is called, which represent the gate going down or up, respectively. The aspect monitors the safety condition and, if it is violated, an alarm is raised. Once the aspect was generated, it is woven into the railroad application. The simple aspect-instrumented version detected the same violations as the Java-MaC-monitored version.

The aspect generation is not yet fully automated. In the near future, we anticipate being able to support the MEDL and PEDL languages in their entirety.

References:

[1] Java MaC, “Run-time Monitoring and Checking (MaC)”. [Online] Available http://www.cis.upenn.edu/~rtg/mac/index.php3, November 23, 2006.

[2] N. Delgado, A. Gates, and S. Roach, “A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools”. *in Proc. IEEE Transactions on Software Engineering*, 30(12), pp.859-872, December 2004.

[3] C. Heitmeyer and D. Mandrioli, “Eds. Formal Methods for Real-Time Systems”. *Number 5 in Trends in Software*. John Wiley & Sons, 1996.

[4] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. “Java-MaC: A Run-time Assurance Tool for Java”. *in Proc. 1st International Workshop on Run-time Verification*. 2001.

[5] E. Hilsdale, and J. Hugunin, “Advice Weaving in AspectJ”, *in Proc. Aspect-oriented Software Development 2004*, 2004, pp. 26-35.

[6] Palo Alto Research Center. “The AspectJ Programming Guide”. [Online] Available http://www.eclipse.org/aspectj/doc/released/progguide/index.html, October 1, 2006.

[7] G. Kiczales, J. Lamping, and A. Mendhekar, “Aspect-Oriented Programming”. *in Proc. European Conference on Object-Oriented Programming 1997*,1997, volume 1241 of LNCS, pp. 220-242.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *in Proc. of the European Conference on Object-Oriented Programming 2001*, 2001.

From Runtime Verification to Evolvable Software

Howard Barringer, David Rydeheard *
EMAIL: {howard.barringer, david.rydeheard}@manchester.ac.uk

Dov Gabbay †
EMAIL: dov.gabbay@kcl.ac.uk

January 26, 2007

Summary

In [4] we developed a revision-based logical modelling approach for evolvable systems, built as hierarchical assemblies of components. A component may encapsulate *horizontal* compositions of interacting sub-components as well as specially paired *vertical* compositions of supervisor and supervisee subcomponents. Our work here extends this logical framework to incorporate programs within each component. We first consider a setting where the programs associated with both supervisor and supervisee components are written in the same guarded command style imperative language. However, as supervisor programs typically follow a monitor, diagnose and revise pattern, we then illustrate how temporal logic rule based supervisor programs, mixing declarative and imperative styles, can be semantically incorporated. Indeed, our modelling framework can fibre as many different programming languages as are necessary for the natural expression of the desired evolvable system behaviour. We use a model of a reactively planned remote roving vehicle as a motivating example.

Background and Motivation

We are interested in developing theories and tools to support the construction and running of safe, robust and controllable systems that have the capability to evolve or adapt their structure and behaviour dynamically according to both internal and external stimuli. We distinguish such evolutionary changes from the normal computational flow steps of a program; in particular, such changes may involve the revision of fixed structural elements, replacement of components and/or programs, or larger scale reconfigurations of systems. Evolution steps may be determined by internal monitoring of a system's behaviour identifying a need for change in structure or computation, or may be triggered by some external influence, e.g. a human user or some other computational agent. We refer to such systems in general terms as *evolvable* systems.

Many computational systems are naturally structured and modelled as evolvable systems. Examples include: business process modelling, which adapt their processes according to internal and external imperatives [1]; supervisory control systems for, say, reactive planning [9, 13]; systems for adaptive querying over changing databases [7]; autonomous software repair [12], data structure repair [6]; hybrid systems [11] that change their computational behavior in response to environmental factors that they may themselves influence. Features of evolvable systems, such as the monitoring of aspects across components, are also found in Aspect-oriented Programming [10] and Monitor-oriented Programming [5]. Clearly, the work in the field of runtime verification, addressing the monitoring of system behaviour against desired properties, or specification, is highly relevant to the design and structuring of such evolvable systems.

Our logical account of evolvable computational systems given in [4] aimed at a more refined understanding of these complex system behaviours. We introduced evolution at a level of abstraction that allows us to describe systems that are constructed as a hierarchical assembly of software and hardware components. Software (and hardware) components are modelled as logical theories built from predicates and axioms. The state of a component is a set of formulae of the theory; the formulae record observations that are valid at that stage of the computation. As components compute, their states change. For normal computational steps, these changes are described as revisions to the set of formulae, in a style familiar in revision-based logic [8]. This is just one particular approach to describing computational behaviour, however, it is an approach with built-in persistence — an important feature for describing evolutionary behaviour.

Components may also be constructed as a pairing of a supervisor and supervisee component in which the supervisor component embodies a process of monitoring and possibly evolving its supervisee

*School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK

†Department of Computer Science, Kings College London, The Strand, London, WC2R 2LS, UK

component. Although the supervisor is a component, it stands in a special relationship to its supervisee. In our logical account, this relationship is that of the supervisor theory being *meta* to the object-level supervisee theory. In other words, the supervisor theory has access to the (entirety of the) logical structure of the theory of the supervisee, thus including its predicates, formulae, state, axioms, revision actions, and its subcomponent theories. This equips the supervisor with sufficient capability to describe evolutionary object-level supervisee changes. Thus, not only can meta-level (supervisor) states record observations of its own state of computation, but they can also record observations about the object-level (supervisee) system. Revision actions at the meta-level update the state of the supervisor and, as a consequence of being meta to the supervisee, may also induce a transformation of the object-level, or supervisee, system. It is in this way that we capture evolutionary change. By introducing tree-structured logical descriptions and associated revision operations, we showed how the framework could be extended to evolvable systems built from hierarchies of evolvable components.

Programs for evolution

In this presentation, we outline how our modelling approach can be extended with the introduction of programs over the actions of the component theories. Without digressing into the debate on whether components should be viewed as active, or passive service providers, we enhance our component model and theory so that each component (and hence all of its subcomponents) comes equipped with its own "main" program, which is executed upon component instance creation. This choice of active componentry is not restrictive as it can easily be used to model passive service-provider component models. Furthermore, component instances of different component schema may use different programming languages. Of course, this is common in practice; for example, shell scripts supervising the execution of particular programs (in different languages), or temporal logic (or history/trace) based languages used for monitoring imperative C or Java programs. Seldom, however, do such combinations come equipped with a logical account of the combined systems.

A structural operational semantics, as well as a trace-based denotational semantics, has been provided for the various ways that component programs may be combined, including, in particular, the supervisor-supervisee combination of evolvable components. This provides not only a foundation for static proof analysis of an evolvable component hierarchy but also a natural setting for dynamic, reasoned and programmed, control of a system's evolution as a generalization of standard runtime verification techniques. In addition, we illustrate a temporal rule-based language, blending concepts from EAGLE [3] and METATEM [2], for supervisory programming.

A roving example

We will motivate our approach to the inclusion of programs in component models using an abstraction of a reactive planning-based remote roving vehicle. Let us here give just an informal idea. At the base level, we model a rover as a simple linear-plan execution engine. The rover's plans, i.e. programs, are sequences of actions such as taking a picture, setting a destination heading, and driving towards the destination. A reactive planner is then modelled as a supervisor for this base engine. The supervisor sets an initial plan, then monitors the execution engine's behaviour. If a planned action fails, e.g. a drive action fails because of some unexpected obstruction, the supervisor must diagnose the problem, replan and reinstall a more appropriate plan for the rover. In this basic application, the supervisory control is modelled using a straightforward guarded command programming language; monitoring is reduced to looking for action failures. Obviously, more complex monitoring, both temporal and spatial, would be required for a more sophisticated, potentially predictive, supervisor. Furthermore, a hierarchy of supervisors may also be necessary. For example, suppose the replan action of a first-level supervisor fails because there is no unobstructed route to the given destination. A higher-level supervisor (re-planner) may well revise the goal to drive to another location or may be able to employ some other technology to remove part of the obstruction.

Whilst supervisory control of planning-based systems is hardly new, this example neatly illustrates how the architecture of such systems and their programs are modelled in a logical framework that provides foundation for static and dynamic reasoning.

References

[1] D. Balasubramaniam, R. Morrison, G.N.C. Kirby, K. Mickan, B.C. Warboys, I. Robertson, B. Snowdon, R.M. Greenwood and W. Seet. A software architecture approach for structuring autonomic systems. In *ICSE 2005 Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005)*, St Louis, MO, USA. ACM Digital Library. 2005.

[2] H. Barringer, M. Fisher, D. Gabbay, G. Gough and R. Owens. METATEM: An introduction. *Formal Aspects of Computing*, 7(5): 533–549, 1995.

[3] H. Barringer, A. Goldberg, K. Havelund and K. Sen. Rule-Based Runtime Verification. Proceedings of the *VMCAI'04, 5th International Conference on Verification, Model Checking and Abstract interpretation*, Venice. Volume 2937, Lecture Notes in Computer Science, Springer-Verlag. 2004.

[4] H. Barringer, D. Gabbay and D. Rydeheard. Logical Modelling of Evolvable Systems. Submitted for publication, 2006. See also http://www.cs.manchester.ac.uk/evolve

[5] F. Chen and G. Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003. http://www.elsevier.nl/locate/entcs/volume89.html

[6] B. Demsky and M. Rinard. Data Structure Repair Using Goal-Directed Reasoning. *Proc. 2005 International Conference on Software Engineering.* St. Louis, Missouri. 2005.

[7] K. Eurviriyanukul, A.A.A. Fernandes and N.W. Paton. A Foundation for the Replacement of Pipelined Physical Join Operators in Adaptive Query Processing. *Current Trends in Database Technology (EDBT Workshops)*, Springer, 589-600. 2006.

[8] R.E. Fikes and N.J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3–4):189–208. 1971.

[9] M.P. Georgeoff and A.L. Lansky. Reactive Reasoning and Planning. *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, WA. 677–682, July 1987.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Proc. of the European Conference on Object-Oriented Programming*, 1241, pp 220–242. 1997.

[11] X.D. Koutsoukos, P.J. Antsaklis, M.D. Lemmon and J.A. Stiver. Supervisory Control of Hybrid Systems. *Proc. of the IEEE, Special Issue on Hybrid Systems*, 88(7),1026-1049. 2000.

[12] R. Levinson. Unified Planning and Execution for Autonomous Software Repair. *Workshop of Plan Execution: A Reality Check, ICAPS'05*, 2005.

[13] N. Muscettola and G. Dorais and C. Fry and R. Levinson and C. Plaunt. Idea: Planning at the core of autonomous reactive agents. Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space, October 2002.

Instrumentation of Open-Source Software For Intrusion Detection

William Mahoney
University of Nebraska at Omaha
282F PKI, 6001 Dodge Street
Omaha, Nebraska 68182-0500
1-402-554-3975
wmahoney@unomaha.edu

William L. Sousan
Computer Science Department
University of Nebraska at Omaha
Omaha, Nebraska
1-402-554-2423
wsousan@unomaha.edu

ABSTRACT

A significant number of cyber assaults are attempted against open source internet support software written in C, C++, or Java. Examples of these software packages include the Apache web server, open source DHCP servers, and network share software such as Samba. These attacks attempt to take advantage of inadvertent flaws left in software systems due to a lack of complete testing, inexperienced developers, intentional backdoors into the system, and other reasons. Detecting all of the flaws in a large system is still a daunting, unrealistic task. If it is not possible to completely secure a system, there is a desire to at least detect intrusion attempts in order to stop them while in progress, or repair the damage at a later date.

The information assurance area of expertise known as "intrusion detection" attempts to sense unauthorized attempts to obtain access to or manipulate information, or to deny the information to other legitimate users. There are several traditional methods used for intrusion detection, which can be categorized into two broad classes: Anomaly Detection, and Misuse Detection.

Anomaly Detection uses statistical approaches and pattern prediction techniques to generate profiles of "typical" user interaction with a system. For example, a certain percentage of the page accesses on a web site may be to a log-in page, and a certain percentage may refer to a page showing the users "shopping cart". Occasionally the user will mistype their password and the log in will fail; for this and other reasons it is likely that more references would be made to the login page than the shopping cart page. If, though, certain pages are suddenly referenced far more frequently, this is an unusual activity and may indicate an intrusion attempt. The advantages of this technique include the capability to detect intrusions which other methods miss, and the fact that the systems are generally adaptable to change over time. But anomaly detection via statistical approaches suffers from a few drawbacks. For example, a nefarious user who knows that the system is adaptable can gradually change the probability for future events until the behavior is considered to be normal. At that point the attacker can penetrate the system without triggering any of the detection alarms. As a counter to these approaches, many anomaly intrusion detection systems also incorporate some form of neural network, which learns to predict a user's next activity and signals an alarm when this prediction is not met.

Misuse Detection systems are typified by expert system software which has knowledge of many known attack scenarios and can monitor user behavior searching for these patterns. A misuse detection system can be thought of as more similar to anti-virus software, which continually searches files and memory for known attack patterns, and alerts the user if any are matched. Misuse systems include a state-based component called an "anticipator", which tries to predict the next activity that will occur on a system. A knowledge base contains the scenarios which the expert system uses to make this prediction, and the audit trail in the system is examined by the expert system to locate partial matches to these patterns. A wildly differing "next event" in a pattern could be an indicator that an intrusion attempt is in progress.

Both types of intrusion detection systems can rely on a variety of data sources in order to build an accurate picture of the normal versus abnormal system activity. However these data sources are almost exclusively comprised of two types: network traffic, and audit logs [1].

This research presents a new approach to generating records for intrusion detection by means of enhancements to the GCC compiler suite. These modifications automatically insert instrumentation calls into the compiled code; the intent of the instrumentation is to generate trace data for intrusion detection systems. Open source code such as a web server can be compiled in this manner, and the execution path of the server can be observed externally in near real-time. (We claim only "near" real-time since the instrumentation is typically queued for a short period between the producing instrumented program and the intrusion detection software.) This method thus creates a completely new source of intrusion detection data which can be incorporated into a detection system.

This "instrumentation compiler" is used for software which is run in a controlled environment in order to gather typical usage patterns. These patterns are ideal for an "anticipator" module in a misuse detection system, as they are made up of the actual execution path of the software under typical usage scenarios. The data included in the instrumentation tracks each procedure entry

and exit point in the software as well as the entry to each basic block in the compiled code.

In a sense the tool appears similar to the Linux utility "gcov" and similar software engineering programs which are used for verifying that each line of code has been executed and tested. However "gcov" and similar tools operate in a batch mode where they first collect statistics, and then later display the program coverage. Our modifications create trace information as each block of the original code is executed. The data generated includes the currently executing function name, the line number in the original software, and the basic block number (for debugging our system) within the function itself. The data could obviously be saved to a file for later analysis, similar to "gcov". But the data is readily available as the program executes and thus can serve as an immediate data feed to our misuse detection system. In addition, our system can change the coverage dynamically during runtime by indicating which functions are to be monitored without restarting the system.

This research paper outlines our intrusion detection scheme and includes two main foci.

First, we discuss the techniques used to modify the internal representations used by the GCC compilers to allow this instrumentation. The compiler uses an internal representation called RTX. Additional calls to the instrumentation functions are automatically generated in RTX just prior to emitting assembly language output. The research paper addresses the techniques for locating the instrumentation points and avoiding problems when software is compiled with optimization. We also present figures addressing the slowdown due to the instrumentation overhead and the additional memory requirements that result by including our instrumentation. The slowdown in compute-bound programs is significant, but our focus is typified by heavily I/O bound processes such as web servers.

Secondly, we have designed and describe a simple a priori domain specific language which we use in order to test for intrusion attempts. Since we are implementing a proof of concept system to determine the feasibility of this method for intrusion detection, our system does not currently encompass any learning modes; instead we manually enter rules based on the past known good observed behavior of the software we are compiling for instrumentation.

Our domain language is a way in which we can specify possible sequences of events which are expected from the instrumentation output, along with the probability of each successor to that event. In this way, potential state transitions create a DFA-like automaton. There is one automaton structure for each possible sequence, and these automatons are traversed in parallel according to the instrumentation output of the program being observed. Final states in the DFA correspond to acceptable sequences of events, while a sufficient number of invalid transitions may be an indicator of an intrusion attempt. Reaching a final state causes all automatons to reset to their initial state. Our language is thus compiled from a human readable format into this set of automatons, which the intrusion detection system then matches against the instrumentation coming from the server program in near real-time.

Our paper lastly outlines the results of this research in general and the issues we have raised but have not yet addressed.

[1] See for example: DARPA Intrusion Detection Evaluation Data Sets, Lincoln Laboratory, Massachusetts Institute of Technology. Available at http://www.ll.mit.edu/IST/ideval/index.html.

A Causality-Based Runtime Check for (Rollback) Atomicity

Serdar Tasiran Tayfun Elmas
Koc University, Istanbul, Turkey
{stasiran,telmas}@ku.edu.tr

Atomicity is used widely for expressing interference-freedom requirements between code blocks in executions of concurrent programs. In this study, we propose a new notion of atomicity called "rollback atomicity".

Commonly used definitions of atomicity have the following form: *A concurrent execution σ^{conc} of a program is* atomic *iff there exists an equivalent, serialized execution σ^{ser} in which every* `atomic` *block by thread* t *is executed with no interruption by actions of other threads.* Rollback atomicity fits into this template as well. In rollback atomicity, we require a particular kind of match between the states of σ^{conc} and the witness execution σ^{ser} at certain points in each execution approximately corresponding to completion points of atomic blocks. A subset F of the shared data variables is designated by the user as the "focus" variables. The rest of the shared variables (the set P) are called "peripheral" variables. The valuation of focus variables in σ^{ser} right after an atomic block $\mathbf{A}$ completes is required to match a valuation obtained from σ^{conc} by (i) considering the program state at the point where $\mathbf{A}$ completes in σ^{conc} and (ii) by rolling back the effects of other atomic blocks $\mathbf{B}$ that commit later, i.e. appear later than $\mathbf{A}$ in σ^{ser}.

Formalizations of atomicity in the literature differ in the notion for the equivalence of executions that they use to interpret the definition above. Reduction and its variants ([2, 5]) are defined based on actions that are left-, right- and both-movers and actions that are non-movers. They require that it be possible to obtain σ^{ser} from σ^{conc} by swapping actions that commute. Conflict-serializability requires that σ^{ser} consist of the same accesses as in σ^{conc} and that the order of accesses to each variable remain the same. View-serializability is a more relaxed notion for atomicity. It requires that σ^{ser} consist of the same accesses as in σ^{conc}, that the final write to each variable in both executions be the same, and that the write seen by each read be the same in both executions. In commit atomicity[3], the final state of a concurrent execution is required to match that of an execution in which atomic blocks are run one at a time, in the order of the occurrence of their commit points in the concurrent execution. Rollback atomicity is a weaker requirement than reduction and conflict-serializability, but is incomparable with view serializability and commit-atomicity. It provides more observability at more points along the execution, but is more permissive in other regards. We highlight some of the differences in Figure 1 where we provide an example in which view-serializability as well as reduction and conflict-serializability are unnecessarily restrictive.

In this example, several concurrent threads can each run the `send` method of a different `Msg` object. The `Msg` objects that are to be sent wait in a queue called `toSendQueue`, thus, `toSendQueue` is shared among different threads. The static field `Msg.KBSentThisSec` and the pool of bytes to be sent, `SendPool` are also shared among threads. Each `Msg` object has a boolean field `sent` that indicates whether it has been sent or not. The `send` method copies the contents of the message (a byte array) to a pool byte by byte where each byte has a message id and a sequence number. The programmer wants the modifications of the `sent` fields and the `toSendQueue` to be atomic. While the `sentPool` data structure is also a shared variable, since the network can already re-order messages, it is not necessary for the sequence of updates to `sentPool` by each `send` method to be atomic. The `KBytesSentThisSec` static field is shared (read and written to) by all threads. It is used for rate control and occasionally causes a `send` method to abort, but otherwise, in non-exceptionally-terminating executions of `send`, it does not affect the functionality of the method. This field is reset every second, and is incremented by all threads manipulating `Msg` objects. The user does not need the complete sequence of updates to this field within a single execution of `send` to be atomic. Also, the read-write dependencies between concurrent executions of `send` caused by this field are not really significant. They do not really point to a data dependency between the atomic blocks.

```
 0: class Msg {
 1:   long msgId;
 2:   static long KBSentThisSec = 0; /* @Periph */
 3:   boolean sent = false;          /* @Focus  */
 4:   byte[] contents;               /* @Focus  */
 5:
 6:   static synchronized long getKBSentThisSecIncr() {
 7:     return ++KBSentThisSec;
 8:   }
 9:
10:
11:   synchronized atomic void send() {
12:
13:     if ( sent  || !toSendQueue.isIn(this))
14:       abort;  // Caller must retry
15:
16:     if (Msg.getKBSentThisSec() > MaxRate)
17:       abort;  // Caller must retry
18:
19:     int i = 0;
20:     while (i < contents.length) {
21:
22:       sendPool.insert(msgId, i, content[i]);
23:         if ( (i++ % 1000) == 0 )
24:           if (Msg.getKBSentThisSecIncr() > MaxRate)
25:              abort; // Caller must retry
26:     }
27:
28:     sent = true;
29:     toSendQueue.remove(this);
30:   } //Commit point
31: }
```

Figure 1. Example 1: Focus variables and rollback atomicity

Consider two normally-terminating concurrent executions of `send` for `Msg` objects `m1` and `m2`. Suppose that an increment that `m1.send()` performs on `Msg.KBytesSentThisSec` is interleaved between two updates to the same static field by `m2.send()` running on another thread. Conflict serializability does not allow the reads and updates to `Msg.KBytesSentThisSec` by `m1.send()` and `m2.send()` to be re-ordered. Likewise, view-serializability does not allow such a re-ordering either, as it requires that the value of `Msg.KBytesSentThisSec` seen by each increment operation to be the same in the concurrent and serial executions. Therefore, these two criteria declare such an execution unserializable. It is possible to construct other interleavings for which commit atomicity would flag a warning because the value of `Msg.KBytesSentThisSec` at the end of a serial execution does not match that at the end of a concurrent one. However, in terms of the other shared variables (`toSendQueue`, `contents` and `Msg.sent`) these concurrent executions can be serialized in the order that `m1` and `m2` are removed from `toSendQueue`. Declaring this latter set of variables as our focus variables while designating the rest of the shared variables as peripheral variables, rollback atomicity gives us a way of expressing the requirement that `toSendQueue`, `contents` and `sent` be updated atomically by each

execution of `send` while `Msg.KBytesSentThisSec` only have a consistent value that allows these executions to complete normally.

In the full paper, we provide examples where view serializability and commit atomicity of a single execution may not provide enough observation points along the execution to reveal bugs. In these examples, rollback atomicity provides the necessary early warning.

1. Rollback Atomicity

We focus only on well-synchronized Java programs whose executions are free of race conditions and thus sequentially consistent. We suppose that the programmer has annotated certain code blocks as atomic. We work with *strong atomicity*, where all other actions modifying or reading focus variables are considered to be atomic blocks as well. Our definition is based on a partition of the set of shared variables into focus and peripheral variables: $F \cup P$.

Consider a concurrent execution σ^{conc} of a program with a set of atomic code blocks $AtBlk$. Let us suppose that a every execution of an atomic block that occurs in σ^{conc} is given a unique id from the set XId. We say that σ^{conc} is *rollback atomic* iff there exists an execution σ^{ser} of the program with the following properties

- For each thread t, the projection of the two executions onto t, $proj(\sigma^{ser}, t)$ and $proj(\sigma^{conc}, t)$, consist of the same sequence of atomic blocks for each thread id t. Exploiting this fact, we use the same id from XId to refer to corresponding occurrences of an atomic block execution by the same thread in σ^{ser} and σ^{conc}. Let us define the *commit order* on XId as follows: $\alpha \leq_{cmt} \beta$ iff $\alpha = \beta$ or α occurs before β in σ^{conc}. If $\alpha \leq_{cmt} \beta$, we say that α *commits before* β.
- Let σ^{ser}_α denote the state of σ^{ser} right after the block with id α has completed executing. Let $\mathbf{s}^{\mathbf{conc}}_\alpha = proj(\sigma^{ser}_\alpha, F)$ be the projection of the state σ^{ser}_α onto the focus variables. Let σ^{conc}_α be the state in σ^{conc} right after atomic block execution with id α has performed the last write access to a variable in F. Let $\mathbf{s}^{\mathbf{ser}}_\alpha = proj(\sigma^{conc}_\alpha, F)$ be the projection of the state σ^{conc}_α onto the focus variables. Let $RlBk(\mathbf{s}^{\mathbf{ser}}_\alpha)$ be obtained from $\mathbf{s}^{\mathbf{ser}}_\alpha$ as follows:
  - Let $v \in F$. If v was last written by a transaction $\beta \leq_{cmt} \alpha$ in σ^{conc} then $RlBk(\mathbf{s}^{\mathbf{ser}}_\alpha)(v) = \mathbf{s}^{\mathbf{ser}}_\alpha(v)$.
  - Otherwise, if v was last modified before σ^{conc}_α by an atomic block that commits before α, then find the most recent write ω to v in σ^{conc} by a block that has committed before α. $RlBk(\mathbf{s}^{\mathbf{ser}}_\alpha)(v)$ is assigned the value written by this most recent committed write. If no such write exists, the value of v is set to its initial value.

  We require that for each α, $RlBk(\mathbf{s}^{\mathbf{ser}}_\alpha)(v) = \mathbf{s}^{\mathbf{conc}}_\alpha(v)$.

2. Checking Rollback Atomicity

Using the infrastructure built for the Vyrd tool [6] we track the accesses to the shared variables throughout the execution. We perform a view refinement check as described in [6] where the abstraction function is given by $RlBk$ as described above. The view refinement check requires that the order of atomic blocks in σ^{ser} be explicitly provided by the user. In order to allow more flexibility in the choice of this order, we instead try to infer it from causality relationships. We construct two graphs of causality dependencies between accesses in order to infer this order: $\mathbf{CG}_F$ and $\mathbf{CG}_{F\cup P}$. The rules for constructing the two graphs are the same. The former is constructed only using accesses to F variables while the latter uses accesses to all shared variables.

In this directed graph, called the causality graph, each `atomic` block in the execution and each individual read and write action correspond to a unique node. The graph has the following sets of edges:

- For each read action r and the write that it sees, $W(r)$, an edge from the node representing $W(r)$ to the node representing r. If any of these actions happen inside some atomic block, the edge starts/ends at the node representing that atomic block.
- For each read action r and the write action w to the same variable that happens immediately after r, there is an edge from the node representing r to the node representing w. If any of these actions happen inside some atomic block, the edge starts/ends at the node representing that atomic block.
- Within each atomic block, if the block contains a write to and a subsequent read of the same variable, there is an edge to each read action from the last write to the same variable in the same atomic block.
- For each pair of nodes α and β representing actions or atomic blocks ordered by program order, there is an edge from α to β.

We update $\mathbf{CG}_F$ and $\mathbf{CG}_{F\cup P}$ as we processes each access in order, by iteratively adding nodes and edges. We search for cycles in each graph after adding an edge that starts/ends at a node representing an `atomic` block [4]. At each such point, there are three possibilities:

- Neither $\mathbf{CG}_F$ nor $\mathbf{CG}_{F\cup P}$ have a cycle containing an atomic block. In this case, we obtain a commit order of atomic blocks by applying the algorithm in [4] to $\mathbf{CG}_{F\cup P}$. In this case the entire execution is conflict-serializable and it is not necessary to perform a rollback atomicity check.
- $\mathbf{CG}_{F\cup P}$ has a cycle containing an atomic block but $\mathbf{CG}_F$ does not. In this case, we obtain a witness order using $\mathbf{CG}_F$ only.
- $\mathbf{CG}_F$ and $\mathbf{CG}_{F\cup P}$ both have cycles. In this case, we take as the commit order the order of the last focus variable writes by atomic blocks.

If there is a read action r to which there is more than one causality edge from write actions in $\mathbf{CG}_F$, an error is declared. This latter warning captures a form of interference in the atomic block that will be discussed in the full paper.

In case our algorithm declares a warning, we were not able to obtain a serialized execution satisfying the rollback atomicity check. In this case, the implementation could truly have undesired behavior, or it could be the case that we were not able to find the right ordering of the atomic blocks. If the latter is the case, however, the witness ordering conflicts certain causality dependencies between focus variables. The programmer can aid our atomicity check by explicitly providing commit point annotations.

References

[1] F. Chen and G. Roşu. Predicting Concurrency Errors at Runtime using Sliced Causality. Technical Report: UIUCDCS-R-2005-2660. Department of Computer Science, University of Illinois at Urbana-Champaign. 2005.

[2] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In Proc. 31st ACM Symposium on Principles of Programming Languages. pp. 256–267, 2004.

[3] C. Flanagan Verifying Commit-Atomicity Using Model Checking Model Checking Software. 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings. Lecture Notes in Computer Science 2989, pp. 252–266.

[4] D. J. Pearce, P. H. Kelly, and C. Hankin. Online Cycle Detection and Difference Propagation: Applications to Pointer Analysis. *Software Quality Control* 12, 4 (Dec. 2004), 311-337.

[5] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs In *PPoPP '06: Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 137–146.

[6] T. Elmas, S. Tasiran, and S. Qadeer. Vyrd: verifying concurrent programs by runtime refinement-violation detection. In *PLDI '05: Proc. of the 2005 ACM SIGPLAN Conference on Programming language Design and Implementation*, pp. 27–37.

On the Semantics of Matching Trace Monitoring Patterns

Pavel Avgustinov Oege de Moor Julian Tibble
[pavel.avgustinov, oege.de.moor, julian.tibble]@comlab.ox.ac.uk
Programming Tools Group, University of Oxford, United Kingdom

1. INTRODUCTION

Many runtime verification properties can be expressed as a pattern that is matched against the trace of runtime events. In our previous work on so-called tracematches, the pattern language consists of regular expressions over events; furthermore the regular expressions contain free variables [1]. The latter feature is indispensable when tracking the behaviour of a clique of objects. The formal semantics of tracematches was worked out in [1]: that paper both gives a declarative semantics and proves it is equivalent to an operational semantics, which in turn is the basis of an efficient implementation.

When comparing tracematches against other formalisms in runtime verification (for instance the PQL system [3]), an important difference is the use of an *exact-match* semantics (where every relevant event must be matched by the pattern) instead of a *skipping* semantics (any event may be skipped during matching). Under skipping semantics, '$A\,B$' means 'A implies eventually B', while under exact-match, '$A\,B$' means 'A implies next B'. It appears that the exact-match semantics are favoured by systems originating in aspect-oriented programming [2, 4, 5], while skipping semantics are more common for systems with roots in runtime verification. Indeed, one could argue that skipping is more natural for specifying temporal properties.

This paper investigates the formal relation between these two styles of specification. In particular, we show that a skipping semantics is no more expressive than exact-match.

2. SEMANTICS OF TRACE PATTERNS

2.1 Exact-match languages

Since the pattern language of *tracematches* [1] is the only one that comes equipped with a formal semantics, we shall make it the starting point of our investigation.

In a nutshell, each tracematch declares a set of symbols, which pick out the events of interest (in fact, they are AspectJ pointcuts). We can think of these as predicates on events, following the intuition suggested by temporal logics. Matching such a predicate (*i.e.* symbol) to an event either results in failure, or in success (possibly subject to a certain instantiation of the tracematch's formal variables).

The pattern itself is a regular expression over the alphabet of symbols, and is matched against *all suffixes* of the *filtered, instantiated trace* — that is, first, the sequence of program events is filtered to a sequence of symbols (*i.e.* propositions that are true at each event), and this is then instantiated for each possible set of variable bindings, resulting in a sequence of *ground symbols*. Since tracematches use an exact-match semantics, matching the regular expression against strings of ground symbols is well-understood.

2.2 Skipping languages

As a first step, we shall provide a formal definition of a simple tracematch-like skipping language: We require an alphabet of declared symbols over which patterns will range, and we shall try to follow the syntax of regular expressions as closely as possible.

It is clear that explicit negation is needed in a skipping language, as otherwise it would be impossible to specify patterns that preclude certain events from occurring in the middle of a match. However, if negation is unrestricted and can apply to general patterns, then a matcher (and hence a semantics) for the language must be capable of arbitrary backtracking, which significantly complicates matters. PQL [3] solves this problem by only permitting negation on a single declared symbol; we go somewhat beyond that by allowing negations of *alternations* of symbols (or, equivalently, symbol sets).

Note that in order to ensure this, we have to use negated terms with care. In particular, they cannot be used freely in sequential composition: since ~a ~b would match exactly the same traces as ~(a b), it is equivalent to a negated compound pattern. We refer to terms that do not start or end in negation as *closed* (cf. Figure 1), and only such terms may be freely sequentially composed.

```
pattern := closedTerm
         | closedTerm '|' pattern

closedTerm := simplePattern
            | simplePattern closedTerm
            | simplePattern '~' symbolSet
                                closedTerm

simplePattern := symbol
               | '(' pattern ')'

symbolSet := symbol
           | '(' symbol '|' symbolSet ')'
```

Figure 1: The grammar of our simple skipping language

It is interesting to observe that the Kleene star is not present in our skipping language definition. This is not an oversight: Under a skipping semantics, a b* c would match

$$[\![\ \text{cTerm} \mid \text{pat}\ ]\!] \Longrightarrow [\![\ \text{cTerm}\ ]\!] \mid [\![\ \text{pat}\ ]\!]$$

$$[\![\ \text{sPat cTerm}\ ]\!] \Longrightarrow [\![\ \text{sPat}\ ]\!]\ \Sigma* [\![\ \text{cTerm}\ ]\!]$$

$$[\![\ \text{sPat}\ \tilde{}\alpha\ \text{cTerm}\ ]\!] \Longrightarrow [\![\ \text{sPat}\ ]\!]\ (\Sigma \setminus \alpha)* [\![\ \text{cTerm}\ ]\!]$$

$$[\![\ \text{symbol}\ ]\!] \Longrightarrow \text{symbol}$$

Figure 2: Rewrite rules translating into the tracematch language

precisely when a c matches, since any events are permitted between a and c. Thus, Kleene closure does not add expressiveness, since the matcher could always choose to match it against the empty program trace, and then skip over an arbitrary sequence.

To specify the semantics of this language, we proceed by providing a set of simple syntax-directed rewriting rules that translate a skipping pattern into a standard tracematch pattern (the semantics of which is well-understood). The translation proceeds by structural induction on the skipping pattern; full details are given in Figure 2 (where cTerm is of type closedTerm, pat is a pattern, sPat is a simplePattern, α is an alternation of symbols, interpreted as a set, and Σ is the set of all declared symbols).

As expected, the basic structure of the pattern carries over. Alternation is mapped to alternation, and each individual symbol is mapped to the same symbol. The interesting cases concern sequential composition: either with or without an intervening negated set of symbols.

In the negation-free case, we want to capture the fact that "an arbitrary number of events of any kind are allowed in between consecutive matched statements" [3]. The natural way to ensure this is to add $\Sigma*$ between the two patterns: Since Σ is the entire alphabet of symbols, this has the desired effect.

When there is a negated set of symbols, the translation is similar, but we allow any number of events matching symbols in $\Sigma\backslash\alpha$ to occur. If we interpret the Kleene closure of the empty set to only match the empty trace, then this has exactly the effect of prohibiting the symbols in α under a skipping semantics.

This simple set of rewrite rules suffices to pin down the semantics of our small skipping language; we therefore conclude that changing the interpretation of patterns from exact-match to skipping does not by itself increase expressiveness.

3. EXPRESSIVENESS OF SKIPPING LANGUAGES

Our translation showed that a regular expressions-based skipping language is not more expressive than the corresponding exact-match language. At the same time, we were forced to do without Kleene closure, since any Kleene-starred expression can be dropped from a skipping pattern without altering matching behaviour.

Note that the claim above is only true if we insist that Kleene-starred terms are not *closed terms* in the sense of the condition on sequential composition imposed above. What would happen if we were to consider them closed?

Unfortunately, the answer is that we end up with a non-compositional semantics, meaning that it is not valid to substitute equivalent subexpressions for each other. Consider the following two patterns: a b* c and a c. They clearly match the same set of traces, so (a b*) and (a) are equivalent subexpressions. Now consider the context C(X) := (X ~b c), defined for all closed instantiations of X. Since both our subexpressions would be closed, we can plug each of them into the context to obtain the patterns a b* b~ c and a ~b c, respectively. But the former matches the trace A B C, while the latter doesn't.

Thus, it would seem that the skipping language is strictly less expressive than the tracematch language. This notion is formalised in the paper by giving a backwards translation from a *subset* of the tracematch language to the skipping language (it turns out that to get an equivalent language, we need to restrict Kleene closure to symbol sets, mirroring the restriction on explicit negation).

4. CONCLUSIONS AND FUTURE WORK

This work examines the semantics of skipping-based trace monitoring languages, and gives a formal semantics for a simple regular expressions-based skipping language, as well as an argument of equivalence to a subset of the trace language of tracematches [1].

Since it was shown that a skipping interpretation precludes the use of Kleene closure, it is interesting to investigate in how far the results presented here carry over when we consider more expressive languages (context-free grammars, or their closure under intersection — this is the class of languages accepted by [3]).

The small trace language described above has been implemented as an extension to the tracematches system and will be made available in the very near future.

5. REFERENCES

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.

[2] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura, and Mario Südholt. An expressive aspect language for system applications with arachne. In *Aspect-Oriented Software Development*, pages 27–38. ACM Press, 2005.

[3] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 365–383. ACM Press, 2005.

[4] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *ECOOP*, 2005.

[5] Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159–169. ACM Press, 2004.

Statistical Runtime Checking of Probabilistic Properties

Usa Sammapun Insup Lee Oleg Sokolsky
University of Pennsylvania

Abstract

Probabilistic correctness is another important aspect of reliable systems. A soft real-time system, for instance, exhibits probabilistic behaviors from tolerating some degrees of deadline misses. Since probabilistic systems may behave differently from their probabilistic models depending on their current environments, checking the systems at runtime can provide another level of assurance for their probabilistic correctness.

Runtime verification is a technique for checking correctness of a system at runtime by observing a system execution and checking it against its property specification. To check probabilistic properties, runtime verification can adopt a statistical technique used in model checking to check probabilistic properties. The statistical technique simulates, samples many execution paths, and estimates probabilities by counting successful samples against all samples. One particular difficulty in using this technique in runtime verification, however, is that runtime verification has only one execution path and cannot simply collect many different execution paths as in statistical probabilistic model checking. Therefore, this one execution path, usually in a form of a trace of states or events, needs to be broken down into different individual samples, which can be done only if a probabilistic system being observed has repeated or periodic behaviors such as soft real-time schedulers or network protocols.

To break apart one execution, runtime verification must be able to distinguish one repeated behavior from another by just looking at the system trace. The trace may also contain other states or events unrelated to probabilistic properties. Nonetheless, breaking apart one execution can be done via conditional probabilities. Written in terms of probabilistic properties, one can specify as given a condition A, does the probability that an outcome B occurs fall within a given range? For example, given that a real-time task has started, is the probability that it finishes within its deadline greater than 0.8? If we can relate the condition A to the outcome B within the trace, a set of A and B can be collected as one individual sample, and a sequence of the set can be collected as many different individual samples. The probability observed from the system can be estimated by counting the condition A with the outcome B against all A. After the probabilities are estimated, runtime verification uses statistical analysis such as hypothesis testing to provide a systematic procedure with an adequate level of confidence to determine statistically whether a system satisfies a probabilistic property.

The above statistical technique for checking probabilistic properties is applied to a runtime verification framework called MaC or Monitoring and Checking. MaC provides expressive specification languages based on Linear Temporal Logics to specify system properties. Once the properties are specified, MaC observes the system by retrieving system information from

probes instrumented into the system prior to the execution. MaC then checks the execution against the system properties and reports any violations. The main aspect of MaC is its formal property specification. MaC specification is built upon two elements: events and conditions. Events occur instantaneously during execution, whereas conditions represent system states that hold for a duration of time and can be $true$ or $false$. For example, an event denoting a call to a method $init$ occurs at the instant the control is passed to the method, and a condition $v < 5$ holds as long as the value v is less than *5*. Events and conditions can be composed using boolean operators such as negation, conjunction, disjunction, and other temporal operators.

Probabilistic properties can be specified by quantifying these events with probabilities. Using conditional probabilities to break one execution path into several samples, the probabilistic events are defined as follows. Given that an event a occurs, does the probability that an event b occurs fall within a given range? The probabilities are estimated by counting the number of an event b that occurs in response to an event a against the total number of an event a. The hypothesis testing is then used to decide whether the estimated probability falls within the given range of a probability quantified in a probabilistic event. A common statistics technique of z-score is used to tell statistically how far apart the estimated probability from the quantified probability. A threshold is set up with some levels of confidence to test whether the estimated probability falls with the quantified probability. If not, a violation is reported. We have applied this technique to check probabilistic properties in wireless sensor network applications.

The implementation is done using sliding windows by collecting a fixed number of samples and shifting the window appropriately over time. The sliding windows can detect different probabilistic behaviors over time. For example, a soft real-time system may behave well during normal loads but poorly during overloaded periods. When using all samples without the sliding windows, the difference in these situations may not be detected.

MaC does not provide probabilistic conditions because of possible inconsistency in the results of hypothesis testing. To understand the reason, we give possible definitions of probabilistic conditions. One can specify them as given that a MaC condition a is true, what is the probability that a MaC condition b is true? The sample can be collected by counting the number of states (or the time duration) where a and b are true against the number of states (or the time duration) where a is true. These definitions are sensitive to the way the system is instrumented or the time metric used to measure the duration of time. Fine-grained instrumentation (or time metric) gives more states (or larger duration) than coarse-grained ones. In either case, the resulted probability estimation using different levels of instrumentation or time metric may still be the same, only the numbers of samples that are different. However, z-score is sensitive to the number of samples and may produce different results even though the estimated probabilities are the same leading to possible inconsistency in reporting violations.

Other existing runtime verification frameworks that provide probabilistic properties do not use any statistical analysis to support the estimated probabilities. Most of their semantics also uses propositions, equivalent to MaC conditions, as their basic elements and are subject to the possible inconsistency of hypothesis testing results. MaC however provides events in addition to conditions and can use the statistical technique to check probabilistic events.

Our contributions are: 1) we provide a general statistical technique for checking probabilistic properties at runtime, 2) the technique is applied to and implemented in an existing runtime verification framework called MaC, and 3) a discussion is given about possible inconsistency when using the statistical technique with conditions or other runtime verification frameworks.