Patrice Godefroid (Ed.)

LNCS 3639

Model Checking Software

12th International SPIN Workshop
San Francisco, CA, USA, August 2005
Proceedings

Springer

TP311.5-53
S757
2005

Patrice Godefroid (Ed.)

Model Checking Software

12th International SPIN Workshop
San Francisco, CA, USA, August 22-24, 2005
Proceedings

Springer

Volume Editor

Patrice Godefroid
Bell Laboratories, Lucent Technologies
2701 Lucent Lane, Lisle, IL 60532, USA
E-mail: god@bell-labs.com

Library of Congress Control Number: 2005930636

CR Subject Classification (1998): F.3, D.2.4, D.3.1, D.2

ISSN 0302-9743
ISBN-10 3-540-28195-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28195-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11537328 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 12th International SPIN Workshop on Model Checking of Software, held in San Francisco, USA, on August 22–24, 2005. SPIN 2005 is a forum for practitioners and researchers interested in model-checking based techniques for the validation and analysis of communication protocols and software systems. The workshop focuses on topics including theoretical and algorithmic foundations and tools for software model checking, model derivation from code and code derivation from models, techniques for dealing with large and infinite state spaces, and applications. The workshop aims to foster interactions and exchanges of ideas with all related areas in software engineering. It has traditionally drawn contributions from both academia and industry.

The SPIN workshop series started 10 years ago, in 1995. Since then, SPIN workshops have been held on an annual basis at Montréal (1995), New Brunswick (1996), Enschede (1997), Paris (1998), Trento (1999), Toulouse (1999), Stanford (2000), Toronto (2001), Grenoble (2002), Portland (2003) and Barcelona (2004). All but the first SPIN workshop were organized as satellite events of larger conferences, in particular of CAV (1996), TACAS (1997), FORTE/PSTV (1998), FLOC (1999), the World Congress on Formal Methods (1999), FMOODS (2000), ICSE (2001, 2003) and ETAPS (2002, 2004). This year, SPIN was held as a satellite event of CONCUR 2005. The co-location of SPIN workshops with conferences has proven to be very successful and has helped to disseminate SPIN model checking technology to wider audiences. Since 1999, the proceedings of the SPIN workshops have appeared in Springer's Lecture Notes in Computer Science series.

The history of successful SPIN workshops is evidence for the maturing of software model-checking technology. While in earlier years the focus of the workshop series was algorithms and tool development around the SPIN model-checker, its scope was widened several years ago to include other software model-checking techniques, tools and applications.

This year, we received 45 regular paper submissions out of which 15 were selected. In addition, the SPIN 2005 program contained 4 tool presentation papers selected from 6 submissions. All submissions went through a rigorous reviewing process, where each paper received a mimimum of 3 referee reviews.

In addition to the refereed papers, three invited talks were given, by David Wagner (UC Berkeley) on *Pushdown Model Checking for Security*, Dawson Engler (Stanford University) on *Static Analysis Versus Model Checking for Bug Finding*, and Rajeev Alur (University of Pennsylvania) on *The Benefits of Exposing Calls and Returns* (invited talk of CONCUR/2005, and joint with SPIN/2005). Dawson Engler also contributed an original paper entitled *Execution Generated Test Cases: How to Make Systems Code Crash Itself* (co-authored with Cristian Cadar), which can be found in these proceedings. The program also included three invited tutorials, by Gerard J. Holzmann (NASA JPL) and Theo C. Ruys (University of Twente) on *Effective Bug Hunting with* SPIN *and* MODEX, by Thomas A. Henzinger (EPFL), Ranjit Jhala (UCSD) and Rupak Majumdar (UCLA) on *The BLAST Software Verification System*, and by Willem Visser (NASA Ames) on *Model Checking Programs with Java PathFinder*.

I would like to thank the Program Committee members, as well as all the external reviewers who assisted them in their work. My thanks also go to the Steering Committee members and last year's organizers, Susanne Graf and Laurent Mounier, for their helpful advice. I would also like to thank the invited speakers and invited tutorial speakers, the authors of submitted papers, and the workshop participants. Special thanks go to Springer for providing us with the possibility of using a Conference Online Service free of charge and to the METAFrame team for their support. Last but not least, I would like to thank the organizers of CONCUR 2005, in particular Luca de Alfaro, for inviting us to hold SPIN 2005 as a satellite event and for their support and flexibility in accommodating the particular needs of the SPIN workshop.

June 2005 Patrice Godefroid

Organization

Program Committee

George Avrunin (U. Mass. Amherst, USA)
Dennis Dams (Bell Labs, USA)
Stefan Edelkamp (U. Dortmund, Germany)
Cormac Flanagan (UC Santa Cruz, USA)
Jaco Geldenhuys (Tampere U., Finland)
Patrice Godefroid (Bell Labs, USA; Chair)
Susanne Graf (Verimag, France)
Gerard Holzmann (NASA JPL, USA)
Sarfraz Khurshid (UT Austin, USA)
Stefan Leue (U. Konstanz, Germany)
Rupak Majumdar (UCLA, USA)
Laurent Mounier (Verimag, France)
Shaz Qadeer (Microsoft, USA)
Theo Ruys (U. Twente, Netherlands)
Willem Visser (NASA Ames, USA)
Pierre Wolper (U. Liège, Belgium)

Advisory Committee

Gerard Holzmann (NASA JPL, USA; Chair)
Amir Pnueli (Weizmann Inst., Israel)

Steering Committee

Thomas Ball (Microsoft, USA)
Susanne Graf (Verimag, France)
Stefan Leue (U. Konstanz, Germany)
Moshe Vardi (Rice U., USA)
Pierre Wolper (U. Liège, Belgium; Chair)

Additional Reviewers

Husain Aljazzar
Nina Amla
Dragan Boshnachki
Marius Bozga
Richard Chang
Pieter de Villiers
Paul Grisham
Henri Hansen
Klaus Havelund
Shahid Jabbar
Ranjit Jhala
Rajeev Joshi
Alberto Lluch-Lafuente
Oded Maler
Tilman Mehler

Kedar Namjoshi	Nicolas Rouquette	Evghenia Stegantova
Iulian Ober	Hassen Saïdi	Brink Van der Merwe
Corina Pasareanu	Danhua Shao	Wei Wei
Michael Perin	Natalia Sidorova	

Lecture Notes in Computer Science 3639

¥424.80元

Table of Contents

Tool Papers

Pushdown Model Checking for Security

David Wagner

U.C. Berkeley
`daw@cs.berkeley.edu`

Abstract. One of the key challenges for computer security is the problem of software security: how to build software that is free of implementation vulnerabilities. In this talk, I will present experience with pushdown model checking for software security. First, I will survey simple methods for pushdown model checking, and I will introduce MOPS, a tool for pushdown model checking of C programs. Then, I will show many security properties of interest may be encoded as temporal safety properties that are well-suited to analysis with model checking. I will report on our experience applying MOPS to tens of millions of lines of C code. Finally, I will discuss some possible directions for future research.

Execution Generated Test Cases: How to Make Systems Code Crash Itself

Cristian Cadar and Dawson Engler*

Computer Systems Laboratory,
Stanford University,
Stanford, CA 94305, U.S.A

Abstract. This paper presents a technique that uses code to automatically generate its own test cases at run-time by using a combination of symbolic and concrete (i.e., regular) execution. The input values to a program (or software component) provide the standard interface of any testing framework with the program it is testing, and generating input values that will explore all the "interesting" behavior in the tested program remains an important open problem in software testing research. Our approach works by turning the problem on its head: we lazily generate, from within the program itself, the input values to the program (and values derived from input values) as needed. We applied the technique to real code and found numerous corner-case errors ranging from simple memory overflows and infinite loops to subtle issues in the interpretation of language standards.

1 Introduction

Systems code is difficult to test comprehensively. Externally, systems interfaces tend towards the baroque, with many different possible behaviors based on tricky combinations of inputs. Internally, their implementations tend towards heavily entangling nests of conditionals that are difficult to enumerate, much less exhaust with test cases. Both features conspire to make comprehensive, manual testing an enormous undertaking, so enormous that empirically, many systems code test suites consist only of a handful of simple cases or, perhaps even more commonly, none at all.

Random testing can augment manual testing to some degree. A good example is the fuzz [3, 4] tool, which automatically generates random inputs, which is enough to find errors in many applications. Random testing has the charm that it requires no manual work, other than interfacing the generator to the tested code. However, random test generation by itself has several severe drawbacks. First, blind generation of values means that it misses errors triggered by narrow ranges of inputs. A trivial example: if a function only has an error if its 32-bit integer argument is equal to "12345678" then random will most likely have to generate billions of test cases before it hits this specific case. Second, and

* This paper is a shortened version of [1], which was in simultaneous submission with similar but independent work by Patrice Godefroid et al [2]. Our thanks to Patrice for graciously accepting this version as an invited paper.

similarly, random testing has difficulty hitting errors that depend on several different inputs being within specific (even wide) ranges of values. Third, the ability of random testing to effectively generate random noise is also its curse. It is very poor at generating input that has structure, and as a result will miss errors that require some amount of correct structure in input before they can be hit. A clear example would be using random test generation to find bugs in a language parser. It will find cases where the parser cannot handle garbage inputs. However, because of the extreme improbability of random generation constructing inputs that look anything like legal programs it will miss almost all errors cases where the parser mishandles them.

Of course, random can be augmented with some amount of guidance to more intelligently generate inputs, though this comes at the cost of manual intervention. A typical example would be writing a tool to take a manually-written language grammar and use it to randomly generate legal and illegal programs that are fed to the tested program. Another would be having a specification or model of what a function's external behavior is and generate test cases using this model to try to hit "interesting" combinations. However, all such hybrid approaches require manual labor and, more importantly, a willingness of implementors to provide this labor at all. The reluctance of systems builders to write specifications, grammars, models of what their code does, or even assertions is well known. As a result, very few real systems have used such approaches.

This paper's first contribution is the observation that code can be used to *automatically* generate its own potentially highly complex test cases. At a high level, the basic idea is simple. Rather than running the code on manually-constructed concrete input, we instead run it on symbolic input that is initially allowed to be "anything." As the code observes this input, these observations tell us what legal values (or ranges of values) the input could be. Each time the code makes a decision based on an observation we conceptually fork the execution, adding on one branch the constraint that the input satisfies the observation, and on the other that it does not. We can then generate test cases by solving these constraints for concrete values. We call such tests *execution generated testing* (EGT).

This process is most easily seen by example. Consider the following contrived routine **bad_abs** that incorrectly implements absolute value:

```
int bad_abs(int x) {
    if(x < 0)
        return -x;
    if(x == 12345678)
        return -x;
    return x;
}
```

As mentioned before, even such a simple error will probably take billions of random-generated test cases to hit. In contrast, finding it with execution generated testing it is straightforward. Symbolic execution would proceed as follows:

1. Initial state: set x to the symbolic value of "anything." In this case, before any observations at all, it can be any value between INT_MIN and INT_MAX. Thus we have the constraints $x \geq INT\_MIN \wedge x \leq INT\_MAX$.
2. Begin running the code.
3. At the first conditional (line 1) fork the execution, setting x to the symbolic constraint $x < 0$ on the true path, and to $x \geq 0$ on the false path.
4. At the return (line 2) solve the constraints on x for a concrete value (such as `x == -1`). This value is later used used as a test input to bad_abs.
5. At the second conditional (line 3) fork the execution, setting x to the constraints $x \equiv 12345678 \wedge x \geq 0$ on the true path and $x \neq 12345678 \wedge x \geq 0$ on the false path.
6. At the second return (line 4) solve the symbolic constraints $x \equiv 12345678 \wedge x \geq 0$. The value is 12345678 is our second test case.
7. Finally, at line 5, solve x's constraints for a concrete value (e.g., `x = 1`). This value is used as our third, final case.

We can then test the code on the three generated values for x. Of course, this sketch leaves many open questions — when to generate concrete values, how to handle system calls, how to tell what is correct, etc. The rest of the paper discusses these issues in more detail.

There are a couple of ways to look at the approach. From one point of view, implementation code has a "grammar" of the legal inputs it accepts and acts on, or rejects. EGT is an automatic method to extract this grammar (and the concrete sentences it accepts and rejects) from the implementation rather than from a hand-written specification. From another viewpoint, it can be seen as a way to turn code "inside out" so that instead of consuming inputs becomes a generator of them. Finally, and perhaps only half-vacuously, it can be viewed as a crude analogue of the Heisenberg effect in the sense that unlike observations perturbing experiments from a set of potential states into a variety of concrete ones, observations in this case perturb a set of possible inputs into a set of increasingly concrete ones. The more precise the observation the more definitively it perturbs the input. The most precise observation, an equality comparison, fixes the input to a specific concrete value. The least precise, an inequality, simply disallows a single value but leaves all others as possibilities.

This paper has three main contributions:

1. A simple conceptual approach to automatically generate test cases by running code on symbolic inputs.
2. A working prototype EGT system.
3. Experimental results showing that the approach is effective on real code.

The paper is organized as follows. Section 2 gives an overview of the method. Section 3 discusses concrete implementation issues. The next four sections give four case studies of applying the approach to systems code. Finally, Section 7 discusses related work and Section 8 concludes.