Yuri Gurevich
Bertrand Meyer (Eds.)

LNCS 4454

Tests and Proofs

First International Conference, TAP 2007
Zurich, Switzerland, February 2007
Revised Papers

Springer

Yuri Gurevich Bertrand Meyer (Eds.)

Tests and Proofs

First International Conference, TAP 2007
Zurich, Switzerland, February 12-13, 2007
Revised Papers

Springer

Volume Editors

Yuri Gurevich
Microsoft Research
Redmond, WA 98052, USA
E-mail: gurevich@microsoft.com

Bertrand Meyer
ETH Zurich
8092 Zurich, Switzerland
E-mail: Bertrand.Meyer@inf.ethz.ch

Library of Congress Control Number: 2007931908

CR Subject Classification (1998): D.2.4-5, F.3, D.4, C.4, K.4.4, C.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-73769-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-73769-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12095476 06/3180 5 4 3 2 1 0

Lecture Notes in Computer Science 4454

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Preface

To prove the correctness of a program is to demonstrate, through impeccable mathematical techniques, that it has no bugs. To test a program is to run it with the expectation of discovering bugs.

These two paths to software reliability seem to diverge from the very start: if you have proved your program correct, it is fruitless to comb it for bugs; and if you are testing it, that surely must be a sign that you have given up on any hope to prove its correctness.

Accordingly, proofs and tests have, since the onset of software engineering research, been pursued by distinct communities using different kinds of techniques and tools. Dijkstra's famous pronouncement that tests can only show the presence of errors — in retrospect, perhaps one of the best advertisements one can imagine for testing, as if "only" finding bugs were not already a momentous achievement! — didn't help make testing popular with provers, or proofs attractive to testers.

And yet the development of both approaches leads to the discovery of common issues and to the realization that each may need the other. The emergence of model checking was one of the first signs that apparent contradiction may yield to complementarity; in the past few years an increasing number of research efforts have encountered the need for combining proofs and tests, dropping earlier dogmatic views of incompatibility and taking instead the best of what each of these software engineering domains has to offer.

TAP — Tests And Proofs — results from an effort to present and discuss some of the most interesting of today's research projects at the convergence of proofs and tests. The first event of its kind, TAP 2007 was held at ETH Zurich on February, 12–13 2007. The conference demonstrated that this is indeed a vibrant topic with exciting developments and the potential for much further growth and cross-fertilization between the ideas pursued by many groups.

We hope that you will agree that TAP 2007 advanced the understanding of two equally promising approaches to software quality, and that you will find in the results, collected in this volume, a source of insight inspiration, and new challenges.

The success of TAP was the result of contributions by many people. We are particularly grateful to the authors who submitted excellent papers; to the keynote speakers, Yuri Gurevich, Jonathan Ostroff and Yannis Smaragdakis; to the Program Committee members and outside referees who made it possible to conduct an effective process leading to a selection of high-quality papers.

The conference was sponsored by IFIP; we are particularly grateful to the support of IFIP Working Group WG2.3 on Programming Methodology (through its Chairperson, Pamela Zave, and all the other members who supported the idea of IFIP sponsorship) as well as TC2 (the Technical Committee on Programming, especially its Chair Robert Meersman and its then secretary Judith Bishop). ETH Zurich provided excellent facilities and impeccable organization.

The financial support of Microsoft Research was particularly useful and is gratefully acknowledged.

The organization, including the preparation of these proceedings, was made possible by the work of the Organizing Committee: Ilinca Ciupa, Manuel Oriol, Andreas Leitner, Claudia Günthart, and Lisa Liu without whom the conference could not have taken place.

Yuri Gurevich
Bertrand Meyer

Organization

Committees

Conference Chair

Bertrand Meyer, ETH Zurich, Switzerland and Eiffel Software, California, USA

Program Chair

Yuri Gurevich, Microsoft Research, USA

Program Committee

Chandrasekhar Boyapati, University of Michigan, USA
Ed Clarke, Carnegie Mellon University, USA
Michael Ernst, MIT CSAIL, USA
Kokichi Futatsugi, JAIST, Japan
Tom Henzinger, EPFL, Switzerland
Daniel Kroening, ETH Zurich, Switzerland
Gary T. Leavens, Iowa State University, USA
Bertrand Meyer, ETH Zurich, Switzerland
Peter Müller, ETH Zurich, Switzerland
Huaikou Miao, Shanghai University, China
Jeff Offutt, George Mason University, USA
Jonathan Ostroff, York University, Canada
Benjamin Pierce, University of Pennsylvania, USA
Wolfram Schulte, Microsoft Research, USA
Yannis Smaragdakis, University of Oregon, USA
Tao Xie, North Carolina State University, USA
T.H. Tse, University of Hong Kong, China

External Referees

Gerard Basler
Nicolas Blanc
Arindam Chakrabarti
Yuri Chebiriak
Adam Darvas
Weiqiang Kong
Masaki Nakamura
Martin Nordio
Kazuhiro Ogata
Joseph Ruskiewicz
Faraz Torshizi
Jianwen Xiang

Organizing Committee

Lisa (Ling) Liu, ETH Zurich, Switzerland
Ilinca Ciupa, ETH Zurich, Switzerland
Andreas Leitner, ETH Zurich, Switzerland
Claudia Günthart, ETH Zurich, Switzerland
Manuel Oriol, ETH Zurich, Switzerland

Sponsors

ETH Zurich
IFIP
Microsoft Research

Lecture Notes in Computer Science

For information about Vols. 1–4542

please contact your bookseller or Springer

Vol. 4671: V. Malyshkin (Ed.), Parallel Computing Technologies. XIV, 635 pages. 2007.

Vol. 4660: S. Džeroski, J. Todoroski (Eds.), Computational Discovery of Scientific Knowledge. X, 327 pages. 2007. (Sublibrary LNAI).

Vol. 4651: F. Azevedo, P. Barahona, F. Fages, F. Rossi (Eds.), Recent Advances in Constraints. VIII, 185 pages. 2007. (Sublibrary LNAI).

Vol. 4647: R. Martin, M. Sabin, J. Winkler (Eds.), Mathematics of Surfaces XII. IX, 509 pages. 2007.

Vol. 4643: M.-F. Sagot, M.E.M.T. Walter (Eds.), Advances in Bioinformatics and Computational Biology. IX, 177 pages. 2007. (Sublibrary LNBI).

Vol. 4632: R. Alhajj, H. Gao, X. Li, J. Li, O.R. Zaïane (Eds.), Advanced Data Mining and Applications. XV, 634 pages. 2007. (Sublibrary LNAI).

Vol. 4628: L.N. de Castro, F.J. Von Zuben, H. Knidel (Eds.), Artificial Immune Systems. XII, 438 pages. 2007.

Vol. 4624: T. Mossakoski, U. Mantanari, M. Haveraaen (Eds.), Algebra and Coalgebra in Computer Science. XI, 463 pages. 2007.

Vol. 4619: F. Dehne, J.-R. Sack, N. Zeh (Eds.), Algorithms and Data Structures. XVI, 662 pages. 2007.

Vol. 4618: S.G. Akl, C.S. Calude, M.J. Dinneen, G. Rozenberg, H.T. Wareham (Eds.), Unconventional Computation. X, 243 pages. 2007.

Vol. 4617: V. Torra, Y. Narukawa, Y. Yoshida (Eds.), Modeling Decisions for Artificial Intelligence. XII, 502 pages. 2007. (Sublibrary LNAI).

Vol. 4616: A. Dress, Y. Xu, B. Zhu (Eds.), Combinatorial Optimization and Applications. XI, 390 pages. 2007.

Vol. 4615: R. de Lemos, C. Gacek, A. Romanovsky (Eds.), Architecting Dependable Systems IV. XIV, 435 pages. 2007.

Vol. 4613: F.P. Preparata, Q. Fang (Eds.), Frontiers in Algorithmics. XI, 348 pages. 2007.

Vol. 4612: I. Miguel, W. Ruml (Eds.), Abstraction, Reformulation, and Approximation. XI, 418 pages. 2007. (Sublibrary LNAI).

Vol. 4611: J. Indulska, J. Ma, L.T. Yang, T. Ungerer, J. Cao (Eds.), Ubiquitous Intelligence and Computing. XXIII, 1257 pages. 2007.

Vol. 4610: B. Xiao, L.T. Yang, J. Ma, C. Muller-Schloer, Y. Hua (Eds.), Autonomic and Trusted Computing. XVIII, 571 pages. 2007.

Vol. 4609: E. Ernst (Ed.), ECOOP 2007 – Object-Oriented Programming. XIII, 625 pages. 2007.

Vol. 4608: H.W. Schmidt, I. Crnkovic, G.T. Heineman, J.A. Stafford (Eds.), Component-Based Software Engineering. XII, 283 pages. 2007.

Vol. 4607: L. Baresi, P. Fraternali, G.-J. Houben (Eds.), Web Engineering. XVI, 576 pages. 2007.

Vol. 4606: A. Pras, M. van Sinderen (Eds.), Dependable and Adaptable Networks and Services. XIV, 149 pages. 2007.

Vol. 4605: D. Papadias, D. Zhang, G. Kollios (Eds.), Advances in Spatial and Temporal Databases. X, 479 pages. 2007.

Vol. 4604: U. Priss, S. Polovina, R. Hill (Eds.), Conceptual Structures: Knowledge Architectures for Smart Applications. XII, 514 pages. 2007. (Sublibrary LNAI).

Vol. 4603: F. Pfenning (Ed.), Automated Deduction – CADE-21. XII, 522 pages. 2007. (Sublibrary LNAI).

Vol. 4602: S. Barker, G.-J. Ahn (Eds.), Data and Applications Security XXI. X, 291 pages. 2007.

Vol. 4600: H. Comon-Lundh, C. Kirchner, H. Kirchner (Eds.), Rewriting, Computation and Proof. XVI, 273 pages. 2007.

Vol. 4599: S. Vassiliadis, M. Berekovic, T.D. Hämäläinen (Eds.), Embedded Computer Systems: Architectures, Modeling, and Simulation. XVIII, 466 pages. 2007.

Vol. 4598: G. Lin (Ed.), Computing and Combinatorics. XII, 570 pages. 2007.

Vol. 4597: P. Perner (Ed.), Advances in Data Mining. XI, 353 pages. 2007. (Sublibrary LNAI).

Vol. 4596: L. Arge, C. Cachin, T. Jurdziński, A. Tarlecki (Eds.), Automata, Languages and Programming. XVII, 953 pages. 2007.

Vol. 4595: D. Bošnački, S. Edelkamp (Eds.), Model Checking Software. X, 285 pages. 2007.

Vol. 4594: R. Bellazzi, A. Abu-Hanna, J. Hunter (Eds.), Artificial Intelligence in Medicine. XVI, 509 pages. 2007. (Sublibrary LNAI).

Vol. 4592: Z. Kedad, N. Lammari, E. Métais, F. Meziane, Y. Rezgui (Eds.), Natural Language Processing and Information Systems. XIV, 442 pages. 2007.

Vol. 4591: J. Davies, J. Gibbons (Eds.), Integrated Formal Methods. IX, 660 pages. 2007.

Vol. 4590: W. Damm, H. Hermanns (Eds.), Computer Aided Verification. XV, 562 pages. 2007.

Vol. 4589: J. Münch, P. Abrahamsson (Eds.), Product-Focused Software Process Improvement. XII, 414 pages. 2007.

Vol. 4588: T. Harju, J. Karhumäki, A. Lepistö (Eds.), Developments in Language Theory. XI, 423 pages. 2007.

Vol. 4587: R. Cooper, J. Kennedy (Eds.), Data Management. XIII, 259 pages. 2007.

Vol. 4586: J. Pieprzyk, H. Ghodosi, E. Dawson (Eds.), Information Security and Privacy. XIV, 476 pages. 2007.

Vol. 4585: M. Kryszkiewicz, J.F. Peters, H. Rybinski, A. Skowron (Eds.), Rough Sets and Intelligent Systems Paradigms. XIX, 836 pages. 2007. (Sublibrary LNAI).

Vol. 4584: N. Karssemeijer, B. Lelieveldt (Eds.), Information Processing in Medical Imaging. XX, 777 pages. 2007.

Vol. 4583: S.R. Della Rocca (Ed.), Typed Lambda Calculi and Applications. X, 397 pages. 2007.

Vol. 4582: J. Lopez, P. Samarati, J.L. Ferrer (Eds.), Public Key Infrastructure. XI, 375 pages. 2007.

Vol. 4581: A. Petrenko, M. Veanes, J. Tretmans, W. Grieskamp (Eds.), Testing of Software and Communicating Systems. XII, 379 pages. 2007.

Vol. 4580: B. Ma, K. Zhang (Eds.), Combinatorial Pattern Matching. XII, 366 pages. 2007.

Vol. 4579: B. M. Hämmerli, R. Sommer (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment. X, 251 pages. 2007.

Vol. 4578: F. Masulli, S. Mitra, G. Pasi (Eds.), Applications of Fuzzy Sets Theory. XVIII, 693 pages. 2007. (Sublibrary LNAI).

Vol. 4577: N. Sebe, Y. Liu, Y.-t. Zhuang (Eds.), Multimedia Content Analysis and Mining. XIII, 513 pages. 2007.

Vol. 4576: D. Leivant, R. de Queiroz (Eds.), Logic, Language, Information and Computation. X, 363 pages. 2007.

Vol. 4575: T. Takagi, T. Okamoto, E. Okamoto, T. Okamoto (Eds.), Pairing-Based Cryptography – Pairing 2007. XI, 408 pages. 2007.

Vol. 4574: J. Derrick, J. Vain (Eds.), Formal Techniques for Networked and Distributed Systems – FORTE 2007. XI, 375 pages. 2007.

Vol. 4573: M. Kauers, M. Kerber, R. Miner, W. Windsteiger (Eds.), Towards Mechanized Mathematical Assistants. XIII, 407 pages. 2007. (Sublibrary LNAI).

Vol. 4572: F. Stajano, C. Meadows, S. Capkun, T. Moore (Eds.), Security and Privacy in Ad-hoc and Sensor Networks. X, 247 pages. 2007.

Vol. 4571: P. Perner (Ed.), Machine Learning and Data Mining in Pattern Recognition. XIV, 913 pages. 2007. (Sublibrary LNAI).

Vol. 4570: H.G. Okuno, M. Ali (Eds.), New Trends in Applied Artificial Intelligence. XXI, 1194 pages. 2007. (Sublibrary LNAI).

Vol. 4569: A. Butz, B. Fisher, A. Krüger, P. Olivier, S. Owada (Eds.), Smart Graphics. IX, 237 pages. 2007.

Vol. 4568: T. Ishida, S. R. Fussell, P. T. J. M. Vossen (Eds.), Intercultural Collaboration. XIII, 395 pages. 2007.

Vol. 4566: M.J. Dainoff (Ed.), Ergonomics and Health Aspects of Work with Computers. XVIII, 390 pages. 2007.

Vol. 4565: D.D. Schmorrow, L.M. Reeves (Eds.), Foundations of Augmented Cognition. XIX, 450 pages. 2007. (Sublibrary LNAI).

Vol. 4564: D. Schuler (Ed.), Online Communities and Social Computing. XVII, 520 pages. 2007.

Vol. 4563: R. Shumaker (Ed.), Virtual Reality. XXII, 762 pages. 2007.

Vol. 4562: D. Harris (Ed.), Engineering Psychology and Cognitive Ergonomics. XXIII, 879 pages. 2007. (Sublibrary LNAI).

Vol. 4561: V.G. Duffy (Ed.), Digital Human Modeling. XXIII, 1068 pages. 2007.

Vol. 4560: N. Aykin (Ed.), Usability and Internationalization, Part II. XVIII, 576 pages. 2007.

Vol. 4559: N. Aykin (Ed.), Usability and Internationalization, Part I. XVIII, 661 pages. 2007.

Vol. 4558: M.J. Smith, G. Salvendy (Eds.), Human Interface and the Management of Information, Part II. XXIII, 1162 pages. 2007.

Vol. 4557: M.J. Smith, G. Salvendy (Eds.), Human Interface and the Management of Information, Part I. XXII, 1030 pages. 2007.

Vol. 4556: C. Stephanidis (Ed.), Universal Access in Human-Computer Interaction, Part III. XXII, 1020 pages. 2007.

Vol. 4555: C. Stephanidis (Ed.), Universal Access in Human-Computer Interaction, Part II. XXII, 1066 pages. 2007.

Vol. 4554: C. Stephanidis (Ed.), Universal Acess in Human Computer Interaction, Part I. XXII, 1054 pages. 2007.

Vol. 4553: J.A. Jacko (Ed.), Human-Computer Interaction, Part IV. XXIV, 1225 pages. 2007.

Vol. 4552: J.A. Jacko (Ed.), Human-Computer Interaction, Part III. XXI, 1038 pages. 2007.

Vol. 4551: J.A. Jacko (Ed.), Human-Computer Interaction, Part II. XXIII, 1253 pages. 2007.

Vol. 4550: J.A. Jacko (Ed.), Human-Computer Interaction, Part I. XXIII, 1240 pages. 2007.

Vol. 4549: J. Aspnes, C. Scheideler, A. Arora, S. Madden (Eds.), Distributed Computing in Sensor Systems. XIII, 417 pages. 2007.

Vol. 4548: N. Olivetti (Ed.), Automated Reasoning with Analytic Tableaux and Related Methods. X, 245 pages. 2007. (Sublibrary LNAI).

Vol. 4547: C. Carlet, B. Sunar (Eds.), Arithmetic of Finite Fields. XI, 355 pages. 2007.

Vol. 4546: J. Kleijn, A. Yakovlev (Eds.), Petri Nets and Other Models of Concurrency – ICATPN 2007. XI, 515 pages. 2007.

Vol. 4545: H. Anai, K. Horimoto, T. Kutsia (Eds.), Algebraic Biology. XIII, 379 pages. 2007.

Vol. 4544: S. Cohen-Boulakia, V. Tannen (Eds.), Data Integration in the Life Sciences. XI, 282 pages. 2007. (Sublibrary LNBI).

Vol. 4543: A.K. Bandara, M. Burgess (Eds.), Inter-Domain Management. XII, 237 pages. 2007.

¥452.00元

Table of Contents

Combining Static and Dynamic Reasoning for Bug Detection 1
Yannis Smaragdakis and Christoph Csallner

Testable Requirements and Specifications 17
Jonathan S. Ostroff and Faraz Ahmadi Torshizi

Proving Programs Incorrect Using a Sequent Calculus for Java Dynamic Logic 41
Philipp Rümmer and Muhammad Ali Shah

Testing and Verifying Invariant Based Programs in the SOCOS Environment 61
Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen

Testing and Proving Distributed Algorithms in Constructive Type Theory 79
Qiao Haiyan

Automatic Testing from Formal Specifications 95
Manoranjan Satpathy, Michael Butler, Michael Leuschel, and S. Ramesh

Using Contracts and Boolean Queries to Improve the Quality of Automatic Test Generation 114
Lisa (Ling) Liu, Bertrand Meyer, and Bernd Schoeller

Symbolic Execution Techniques for Refinement Testing 131
Pascale Le Gall, Nicolas Rapin, and Assia Touil

Test-Sequence Generation with Hol-TestGen with an Application to Firewall Testing 149
Achim D. Brucker and Burkhart Wolff

Generating Unit Tests from Formal Proofs 169
Christian Engel and Reiner Hähnle

Using Model Checking to Generate Fault Detecting Tests 189
Angelo Gargantini

White-Box Testing by Combining Deduction-Based Specification Extraction and Black-Box Testing 207
Bernhard Beckert and Christoph Gladisch

Author Index 217

Combining Static and Dynamic Reasoning for Bug Detection

Yannis Smaragdakis[1] and Christoph Csallner[2]

[1] Department of Computer Science
University of Oregon, Eugene, OR 97403-1202, USA
yannis@cs.uoregon.edu
[2] College of Computing
Georgia Institute of Technology, Atlanta, GA 30332, USA
csallner@gatech.edu

Abstract. Many static and dynamic analyses have been developed to improve program quality. Several of them are well known and widely used in practice. It is not entirely clear, however, how to put these analyses together to achieve their combined benefits. This paper reports on our experiences with building a sequence of increasingly more powerful combinations of static and dynamic analyses for bug finding in the tools JCrasher, Check 'n' Crash, and DSD-Crasher. We contrast the power and accuracy of the tools using the same example program as input to all three.

At the same time, the paper discusses the philosophy behind all three tools. Specifically, we argue that trying to detect program errors (rather than to certify programs for correctness) is well integrated in the development process and a promising approach for both static and dynamic analyses. The emphasis on finding program errors influences many aspects of analysis tools, including the criteria used to evaluate them and the vocabulary of discourse.

1 Introduction

Programming is hard. As an intellectual task, it attempts to approximate real-world entities and conditions as abstract concepts. Since computers are unforgiving interpreters of our specifications, and since in software we can build up complexity with no physical boundaries, it is easy to end up with artifacts that are very hard to comprehend and reason about. Even moderate size programs routinely surpass in detail and rigor the most complex laws, constitutions, and agreements in the "real world". Not only can individual program modules be complex, but the interactions among modules can be hardly known. Most programmers work with only a partial understanding of the parts of the program that their own code interacts with. Faced with this complexity, programmers need all the help they can get. In industrial practice, testing has become significantly more intense and structured in the past decade. Additionally, numerous static analyses attempt to automatically certify properties of a program, or detect errors in it.

B. Meyer and Y. Gurevich (Eds.): TAP 2007, LNCS 4454, pp. 1–16, 2007.

In the past few years, we have introduced three program analysis tools for finding program defects (bugs) in Java applications. *JCrasher* [3] is a simple, mostly dynamic analysis that generates JUnit test cases. Despite its simplicity it can find bugs that would require complex static analysis efforts. *Check 'n' Crash* [4] uses JCrasher as a post-processing step to the powerful static analysis tool ESC/Java. As a result, Check 'n' Crash is more precise than ESC/Java alone and generates better targeted test cases than JCrasher alone. *DSD-Crasher* [5] adds a reverse engineering step to Check 'n' Crash to rediscover the program's intended behavior. This enables DSD-Crasher to suppress false positives with respect to the program's informal specification. This property is more useful for bug-finding than for proving correctness, as we argue later.

In this paper, we report on our experience with these tools and present their comparative merits through a simple example. At the same time, we discuss in detail our philosophy in building them. All three tools are explicitly geared towards finding program errors and not towards certifying program correctness. Viewed differently, program analyses (regardless of the artificial static/dynamic distinction) can never accurately classify with full confidence all programs as either correct or incorrect. Our claim is that analyses that choose to be confident in their incorrectness classification (*sound for incorrectness*) are gaining ground over analyses that choose to be confident in their correctness classification (*sound for correctness*). We discuss this point next in more detail.

2 Bug Finding Musings

There are several dichotomies in program analysis. Clearly, analyses are often classified as *static* or *dynamic*. Additionally, analyses are often classified as *sound* or *complete*, or as *over-* and *under-approximate*. We next present some thoughts on these distinctions as well as the terminology they introduce.

2.1 Static and Dynamic Analysis

At first glance it may seem simple to classify an analysis as static or dynamic. The definition in the popular Wikipedia archive claims that:

> *Static code analysis* is the analysis of computer software that is performed without actually executing programs built from that software (analysis performed on executing programs is known as *dynamic analysis*).

This definition is not quite satisfying, however. Program execution only differs from program reasoning at the level of accuracy. This distinction is fairly artificial. First, there are languages where reasoning and execution are often thought of in the same terms (e.g., static analyses of Prolog programs often include steps such as "execute the program in a universe that only includes these values"). Second, even in imperative languages, it is often hard to distinguish between a virtual machine that executes the program and tools that reason about it

at some level of abstraction (e.g., model checking tools, or symbolic execution analyses). Finally, it is hard to classify analyses that execute a program with known inputs. Known inputs are by definition "static", in standard terminology, and these analyses give information about the program without executing it under "real" conditions. Yet at the same time, since the program is executed, it is tempting to call such analyses "dynamic".

We believe that there is a continuum of analyses and the static vs. dynamic classification is not always easy to make. Our working definition is as follows:

> An analysis is "dynamic" if it emphasizes control-flow accuracy over data-flow richness/generality, and "static" if it emphasizes data-flow richness/generality over control-flow accuracy.

There is always a trade-off between these trends. The undecidability of most useful program properties entails that one cannot make statements about infinitely many inputs without sacrificing some control-flow accuracy.

Although the definition is approximate, we believe that it serves a useful purpose. It reflects the intuitive understanding of the two kinds of analyses, while emphasizing that the distinction is arbitrary. A more useful way to classify analyses is in terms of *what* they claim not *how* they maintain the information that leads to their claims.

2.2 Soundness for Incorrectness

Analyses can be classified with respect to the set of properties they can establish with confidence. In mathematical logic, reasoning systems are often classified as *sound* and *complete*. A sound system is one that proves only true sentences, whereas a complete system proves all true sentences. In other words, an analysis is sound iff $provable(p) \Rightarrow true(p)$ and complete iff $true(p) \Rightarrow provable(p)$. Writing the definitions in terms of what the analysis claims, we can say:

Definition 1 (Sound). $claim_{true}(p) \Rightarrow true(p)$.

Definition 2 (Complete). $true(p) \Rightarrow claim_{true}(p)$.

When we analyze programs we use these terms in a qualified way. For instance, a type system (the quintessential "sound" static analysis) only proves correctness with respect to certain errors.

In our work, we like to view program analyses as a way to prove programs *incorrect*—i.e., to find bugs, as opposed to certifying the absence of bugs. If we escape from the view of program analysis as a "proof of correctness" and we also allow the concept of a "proof of incorrectness", our terminology can be adjusted. Useful program analyses give an answer for all programs (even if the analysis does not terminate, the programmer needs to interpret the non-termination-within-time-bounds in some way). In this setting, an analysis is sound for showing program correctness iff it is complete for showing program incorrectness. Similarly, an analysis is sound for showing program incorrectness iff it is complete for showing program correctness.

These properties are easily seen from the definitions. We have:

Lemma 1. *Complete for program correctness* $\equiv$ *Sound for program incorrectness.*

Proof. Complete for program correctness
$\equiv correct(p) \Rightarrow claim_{cor}(p)$
$\equiv \neg incorrect(p) \Rightarrow \neg claim_{incor}(p)$
$\equiv claim_{incor}(p) \Rightarrow incorrect(p)$
$\equiv$ Sound for program incorrectness

Lemma 2. *Complete for program incorrectness* $\equiv$ *Sound for program correctness.*

Proof. Complete for program incorrectness
$\equiv incorrect(p) \Rightarrow claim_{incor}(p)$
$\equiv \neg correct(p) \Rightarrow \neg claim_{cor}(p)$
$\equiv claim_{cor}(p) \Rightarrow correct(p)$
$\equiv$ Sound for program correctness

In the above, we considered the complementary use of the analysis, such that it claims incorrectness whenever the original analysis would not claim correctness. Note that the notion of "claim" is external to the analysis. An analysis either passes or does not pass programs, and "claim" is a matter of interpretation. Nevertheless, the point is that the same base analysis can be used to either soundly show correctness or completely show incorrectness, depending on how the claim is interpreted.

The interesting outcome of the above reasoning is that we can abolish the notion of "completeness" from our vocabulary. We believe that this is a useful thing to do for program analysis. Even experts are often hard pressed to name examples of "complete" analyses and the term rarely appears in the program analysis literature (in contrast to mathematical logic). Instead, we can equivalently refer to analyses that are "sound for correctness" and analyses that are "sound for incorrectness". An analysis does not have to be either, but it certainly cannot be both for interesting correctness properties.

Other researchers have settled on different conventions for classifying analyses, but we think our terminology is preferable. For instance, Jackson and Rinard call a static analysis "sound" when it is sound for correctness, yet call a dynamic analysis "sound" when it is sound for incorrectness [12]. This is problematic, since, as we argued, static and dynamic analyses form a continuum. Furthermore, the terminology implicitly assumes that static analyses always attempt to prove correctness. Yet, there are static analyses whose purpose is to detect defects (e.g., FindBugs by Hovemeyer and Pugh [11]). Another pair of terms used often are "over-" and "under-approximate". These also require qualification (e.g., "over-approximate for incorrectness" means the analysis errs on the safe side, i.e., is sound for correctness) and are often confusing.

2.3 Why Prove a Program Incorrect?

Ensuring that a program is correct is the Holy Grail of program construction. Therefore analyses that are sound for correctness have been popular, even if limited. For instance, a static type system guarantees the absence of certain kinds of bugs, such as attempting to perform an operation not defined for our data. Nevertheless, for all interesting properties, soundness for correctness implies that the analysis has to be pessimistic and reject perfectly valid programs. For some kinds of analyses this cost is acceptable. For others, it is not—for instance, no mainstream programming language includes sound static checking to ensure the lack of division-by-zero errors, exactly because of the expected high rejection rate of correct programs.

Instead, it is perfectly valid to try to be sound for incorrectness. That is, we may want to show that a program fails with full confidence. This is fairly expected for dynamic analysis tools, but it is worth noting that even static analyses have recently adopted this model. For instance, Lindahl and Sagonas's *success typings* [14] are an analogue of type systems but with the opposite trade-offs. Whereas a type system is sound for correctness and, hence, pessimistic, a success typing is sound for incorrectness and, thus, optimistic. If a success typing cannot detect a type clash, the program might work and is permitted. If the system does report a problem, then the problem is guaranteed to be real. This is a good approach for languages with a tradition of dynamic typing, where users will likely complain if a static type system limits expressiveness in the name of preventing unsafety.

Yet the most important motivation for analyses that are sound for incorrectness springs from the way analyses are used in practice. For the author of a piece of code, a sound-for-correctness analysis may make sense: if the analysis is too conservative, then the programmer probably knows how to rewrite the code to expose its correctness to the analysis. Beyond this stage of the development process, however, conservativeness stops being an asset and becomes a liability. A tester cannot distinguish between a false warning and a true bug. Reporting a non-bug to the programmer is highly counter-productive if it happens with any regularity. Given the ever-increasing separation of the roles of programmer and tester in industrial practice, high confidence in detecting errors is paramount.

This need can also be seen in the experience of authors of program analyses and other researchers. Several modern static analysis tools [10, 8, 11] attempt to find program defects. In their assessment of the applicability of ESC/Java, Flanagan et al. write [10]:

> "[T]he tool has not reached the desired level of cost effectiveness. In particular, users complain about an annotation burden that is perceived to be heavy, and about excessive warnings about non-bugs, particularly on unannotated or partially-annotated programs."

The same conclusion is supported by the findings of other researchers. Notably, Rutar et al. [19] examine ESC/Java2, among other analysis tools, and conclude

that it can produce many spurious warnings when used without context information (method annotations). For five testees with a total of some 170 thousand non commented source statements, ESC warns of a possible null dereference over nine thousand times. Rutar et al., thus, conclude that "there are too many warnings to be easily useful by themselves."

To summarize, it is most promising to use analyses that are sound for correctness at an early stage of development (e.g., static type system). Nevertheless, for analyses performed off-line, possibly by third parties, it is more important to be trying to find errors with high confidence or even certainty. This is the goal of our analysis tools. We attempt to increase the soundness of existing analyses by combining them in a way that reduces the false error reports. Just like analyses that are sound for correctness, we cannot claim full correctness, yet we can claim that our tools are sound for incorrectness with respect to specific kinds of errors. Such soundness-for-incorrectness topics are analyzed in the next section.

3 Soundness of Automatic Bug Finding Tools

In practice, there are two levels of soundness for automatic bug finding tools. The lower level is being sound with respect to the execution semantics. This means that a bug report corresponds to a possible execution of a program module, although the input that caused this execution may not be one that would arise in normal program runs. We call this *language-level soundness* because it can be decided by checking the language specification alone. Many bug finding tools concern themselves only with this soundness level and several of them do not achieve it. A stronger form of soundness consists of also being sound with respect to the intended usage of the program. We call this *user-level soundness*, as it means that a bug report will be relevant to a real user of the program. This is an important distinction because developers have to prioritize their energy on the cases that matter most to their users. From their perspective, a language-level sound but user-level unsound bug report may be as annoying as one that is unsound at the language level.

We next examine these concepts in the context of the ESC/Java tool. Analysis with ESC/Java is an important step for our tools, and we can contrast them well by looking at what need they fill over the base ESC/Java bug finding ability.

3.1 Background: ESC/Java

The Extended Static Checker for Java (ESC/Java) [10] is a compile-time program checker that detects potential invariant violations. ESC/Java compiles the Java source code under test to a set of predicate logic formulae [10]. ESC/Java checks each method `m` in isolation, expressing as logic formulae the properties of the class to which the method belongs, as well as Java semantics. Each method call or invocation of a primitive Java operation in `m`'s body is translated to a check of the called entity's precondition followed by assuming the entity's post-condition. ESC/Java recognizes invariants stated in the Java Modeling Language