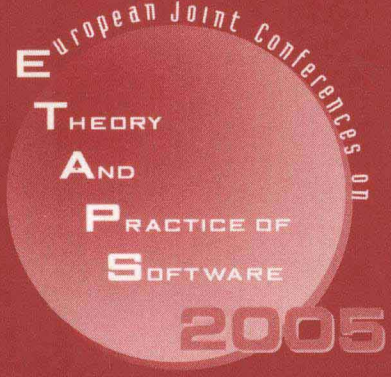
Rastislav Bodik (Ed.)

LNCS 3443

Compiler Construction

14th International Conference, CC 2005
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2005
Edinburgh, UK, April 2005, Proceedings



Springer

Rastislav Bodik (Ed.)

Compiler Construction

14th International Conference, CC 2005
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2005
Edinburgh, UK, April 4-8, 2005
Proceedings

Springer

Volume Editor

Rastislav Bodik
University of California
Computer Science Division, #1776
Berkeley, CA 94720-1776, USA
E-mail: bodik@cs.berkeley.edu

Library of Congress Control Number: 2005922868

CR Subject Classification (1998): D.3.4, D.3.1, F.4.2, D.2.6, F.3, I.2.2

ISSN 0302-9743
ISBN-10 3-540-25411-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-25411-9 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11406921 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 3443

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

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Massachusetts Institute of Technology, MA, USA
Demetri Terzopoulos
New York University, NY, USA
Doug Tygar
University of California, Berkeley, CA, USA
Moshe Y. Vardi
Rice University, Houston, TX, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Foreword

ETAPS 2005 was the eighth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (CC, ESOP, FASE, FOSSACS, TACAS), 17 satellite workshops (AVIS, BYTECODE, CEES, CLASE, CMSB, COCV, FAC, FESCA, FINCO, GCW-DSE, GLPL, LDTA, QAPL, SC, SLAP, TGC, UITP), seven invited lectures (not including those that were specific to the satellite events), and several tutorials. We received over 550 submissions to the five conferences this year, giving acceptance rates below 30% for each one. Congratulations to all the authors who made it to the final program! I hope that most of the other authors still found a way of participating in this exciting event and I hope you will continue submitting.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate program committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for "unifying" talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2005 was organized by the School of Informatics of the University of Edinburgh, in cooperation with

- European Association for Theoretical Computer Science (EATCS);
- European Association for Programming Languages and Systems (EAPLS);
- European Association of Software Science and Technology (EASST).

The organizing team comprised:

- Chair: Don Sannella
- Publicity: David Aspinall
- Satellite Events: Massimo Felici

此为试读,需要完整PDF请访问: www.ertongbook.com

- Secretariat: Dyane Goodchild
- Local Arrangements: Monika-Jeannette Lekuse
- Tutorials: Alberto Momigliano
- Finances: Ian Stark
- Website: Jennifer Tenzer, Daniel Winterstein
- Fundraising: Phil Wadler

ETAPS 2005 received support from the University of Edinburgh.

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Perdita Stevens (Edinburgh, Chair), Luca Aceto (Aalborg and Reykjavík), Rastislav Bodik (Berkeley), Maura Cerioli (Genoa), Evelyn Duesterwald (IBM, USA), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Roberto Gorrieri (Bologna), Reiko Heckel (Paderborn), Holger Hermanns (Saarbrücken), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Kim Larsen (Aalborg), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Hanne Riis Nielson (Copenhagen), Fernando Orejas (Barcelona), Mooly Sagiv (Tel Aviv), Don Sannella (Edinburgh), Vladimiro Sassone (Sussex), Peter Sestoft (Copenhagen), Michel Wermelinger (Lisbon), Igor Walukiewicz (Bordeaux), Andreas Zeller (Saarbrücken), Lenore Zuck (Chicago).

I would like to express my sincere gratitude to all of these people and organizations, the program committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, the many reviewers, and Springer for agreeing to publish the ETAPS proceedings. Finally, I would like to thank the organizer of ETAPS 2005, Don Sannella. He has been instrumental in the development of ETAPS since its beginning; it is quite beyond the limits of what might be expected that, in addition to all the work he has done as the original ETAPS Steering Committee Chairman and current ETAPS Treasurer, he has been prepared to take on the task of organizing this instance of ETAPS. It gives me particular pleasure to thank him for organizing ETAPS in this wonderful city of Edinburgh in this my first year as ETAPS Steering Committee Chair.

Edinburgh, January 2005

Perdita Stevens
ETAPS Steering Committee Chair

Preface

The program committee is pleased to present the proceedings of the 14th International Conference on Compiler Construction (CC 2005) held April 4–5, 2005, in Edinburgh, UK, as part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2005).

Traditionally, CC had been a forum for research on compiler construction. Starting this year, CC has expanded its mission to a broader spectrum of programming tools, from refactoring editors to program checkers to compilers to virtual machines to debuggers. The submissions we received reflected the new scope of the conference.

The Program Committee received 91 submissions (one was later withdrawn), a significant increase from previous years. From the 90 submissions, the Program Committee selected 21 papers, for an acceptance rate of 23%. Four of the accepted papers were tool demonstrations; the submission pool included eight such papers. I believe this is the first CC conference that includes tool demos.

The Program Committee included 15 members representing 10 countries on three continents. Each committee member reviewed (or delegated) roughly 19 papers. Each paper received three reviews. Sixty-eight external reviewers participated in the review process. Committee members were allowed to submit papers, although no paper by a committee member was selected. The Program Committee met on December 4, 2004, in New York for a one-day meeting. All but one of the members participated in the meeting; three members attended via teleconference.

The work of many contributed to the success of this conference. First of all, I want to thank the authors for the care they put into their submissions. My gratitude also goes to Program Committee members and external reviewers for their insightful reviews. IBM generously provided the teleconference service for the Program Committee meeting; thanks to Kemal Ebcioglu for arranging this service. Special thanks go to Manu Sridharan for helping to prepare and run the Program Committee meeting. CC 2005 was made possible by the ETAPS Steering Committee, in particular by the hard work of Don Sannella, the ETAPS 2005 Organizing Committee chair, and José Luiz Fiadeiro and Perdita Stevens, ETAPS chairs. I would also like to thank Evelyn Duesterwald, Görel Hedin, Nigel Horspool and Reinhard Wilhelm, all recent CC chairs, for our many discussions on CC's future directions. Finally, we are grateful to Andreas Zeller for accepting the invitation to give a keynote talk.

Berkeley, January 2005 Rastislav Bodík

Conference Organization

Program Chair

Rastislav Bodík	*UC Berkeley, USA*

Program Committee

Charles Consel	*LABRI, France*
Grzegorz Czajkowski	*Sun Labs, USA*
Angela Demke-Brown	*University of Toronto, Canada*
Amy Felty	*University of Ottawa, Canada*
Antonio Gonzalez	*UPC, Spain*
Thomas Gross	*ETH, Switzerland*
Jan Heering	*CWI, Netherlands*
Roberto Ierusalimschy	*PUC-Rio, Brazil*
Chandra Krintz	*UC Santa Barbara, USA*
Rustan Leino	*Microsoft Research, USA*
Eduard Mehofer	*University of Vienna, Austria*
Michael Philippsen	*Universität Erlangen-Nürnberg, Germany*
G. Ramalingam	*IBM Research, USA*
Michael I. Schwartzbach	*University of Aarhus, Denmark*
Andreas Zeller	*Saarland University, Germany*

Referees

Ali Adl-Tabatabai, Alex Aleta, Erik Altman, Lars Bak, Siegfried Benkner, Claus Brabrand, Mark van der Brand, Joachim Buhmann, Vipin Chaudhary, Aske Christensen, Josep M. Codina, Jesus Corbal, Matteo Corti, Laurent Daynes, David Detlefs, Danny Dubé, Jan van Eijck, Robert van Engelen, Xiaobo Fan, Steve Fink, Ingrid Fischer, Kyle Gallivan, Enric Gibert, David Gregg, Dan Grossman, David Grove, Selim Gurun, Laurie Hendren, Michael Hind, Douglas Howe, Daniel Jackson, Timothy Jones, Mick Jordan, Wayne Kelly, Michael Klemm, Jens Knoop, Gabriella Kokai, Andreas Krall, Geoff Langdale, Julia Lawall, Sorin Lerner, Chuck Liang, Christian Lindig, Josep Llosa, Lukas Loehrer, Guei-Yuan Lueh, Carlos Madriles, Hidehiko Masuhara, Hussam Mousa, Priya Nagpurkar, Krzysztof Palacz, Carlos Garcia Quiñones, Ramshankar Ramanarayana, Jesus Sanchez, Dominic Schell, Florian Schneider, Bernhard Scholz, Christian Sigg, Glenn Skinner, Sunil Soman, Walid Taha, Mads Torgersen, David Ungar, Ronald Veldema, Xavier Vera, Jurgen Vinju, Phillip Yelland, Lingli Zhang

Lecture Notes in Computer Science

For information about Vols. 1–3342

please contact your bookseller or Springer

Vol. 3452: F. Baader, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning. XI, 562 pages. 2005. (Subseries LNAI).

Vol. 3448: G.R. Raidl, J. Gottlieb (Eds.), Evolutionary Computation in Combinatorial Optimization. XI, 271 pages. 2005.

Vol. 3447: M. Keijzer, A. Tettamanzi, P. Collet, J.v. Hemert, M. Tomassini (Eds.), Genetic Programming. XIII, 382 pages. 2005.

Vol. 3443: R. Bodik (Ed.), Compiler Construction. XI, 305 pages. 2005.

Vol. 3442: M. Cerioli (Ed.), Fundamental Approaches to Software Engineering. XIII, 373 pages. 2005.

Vol. 3441: V. Sassone (Ed.), Foundations of Software Science and Computational Structures. XVIII, 521 pages. 2005.

Vol. 3440: N. Halbwachs, L.D. Zuck (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. XVII, 588 pages. 2005.

Vol. 3436: B. Bouyssounouse, J. Sifakis (Eds.), Embedded Systems Design. XV, 492 pages. 2005.

Vol. 3434: L. Brun, M. Vento (Eds.), Graph-Based Representations in Pattern Recognition. XII, 384 pages. 2005.

Vol. 3433: S. Bhalla (Ed.), Databases in Networked Information Systems. VII, 319 pages. 2005.

Vol. 3432: M. Beigl, P. Lukowicz (Eds.), Systems Aspects in Organic and Pervasive Computing - ARCS 2005. X, 265 pages. 2005.

Vol. 3427: G. Kotsis, O. Spaniol, Wireless Systems and Mobility in Next Generation Internet. VIII, 249 pages. 2005.

Vol. 3423: J.L. Fiadeiro, P.D. Mosses, F. Orejas (Eds.), Recent Trends in Algebraic Development Techniques. VIII, 271 pages. 2005.

Vol. 3422: R.T. Mittermeir (Ed.), From Computer Literacy to Informatics Fundamentals. X, 203 pages. 2005.

Vol. 3421: P. Lorenz, P. Dini (Eds.), Networking - ICN 2005, Part II. XXXV, 1153 pages. 2005.

Vol. 3420: P. Lorenz, P. Dini (Eds.), Networking - ICN 2005, Part I. XXXV, 933 pages. 2005.

Vol. 3419: B. Faltings, A. Petcu, F. Fages, F. Rossi (Eds.), Constraint Satisfaction and Constraint Logic Programming. X, 217 pages. 2005. (Subseries LNAI).

Vol. 3418: U. Brandes, T. Erlebach (Eds.), Network Analysis. XII, 471 pages. 2005.

Vol. 3416: M. Böhlen, J. Gamper, W. Polasek, M.A. Wimmer (Eds.), E-Government: Towards Electronic Democracy. XIII, 311 pages. 2005. (Subseries LNAI).

Vol. 3415: P. Davidsson, B. Logan, K. Takadama (Eds.), Multi-Agent and Multi-Agent-Based Simulation. X, 265 pages. 2005. (Subseries LNAI).

Vol. 3414: M. Morari, L. Thiele (Eds.), Hybrid Systems: Computation and Control. XII, 684 pages. 2005.

Vol. 3412: X. Franch, D. Port (Eds.), COTS-Based Software Systems. XVI, 312 pages. 2005.

Vol. 3411: S.H. Myaeng, M. Zhou, K.-F. Wong, H.-J. Zhang (Eds.), Information Retrieval Technology. XIII, 337 pages. 2005.

Vol. 3410: C.A. Coello Coello, A. Hernández Aguirre, E. Zitzler (Eds.), Evolutionary Multi-Criterion Optimization. XVI, 912 pages. 2005.

Vol. 3409: N. Guelfi, G. Reggio, A. Romanovsky (Eds.), Scientific Engineering of Distributed Java Applications. X, 127 pages. 2005.

Vol. 3408: D.E. Losada, J.M. Fernández-Luna (Eds.), Advances in Information Retrieval. XVII, 572 pages. 2005.

Vol. 3407: Z. Liu, K. Araki (Eds.), Theoretical Aspects of Computing - ICTAC 2004. XIV, 562 pages. 2005.

Vol. 3406: A. Gelbukh (Ed.), Computational Linguistics and Intelligent Text Processing. XVII, 829 pages. 2005.

Vol. 3404: V. Diekert, B. Durand (Eds.), STACS 2005. XVI, 706 pages. 2005.

Vol. 3403: B. Ganter, R. Godin (Eds.), Formal Concept Analysis. XI, 419 pages. 2005. (Subseries LNAI).

Vol. 3401: Z. Li, L.G. Vulkov, J. Waśniewski (Eds.), Numerical Analysis and Its Applications. XIII, 630 pages. 2005.

Vol. 3399: Y. Zhang, K. Tanaka, J.X. Yu, S. Wang, M. Li (Eds.), Web Technologies Research and Development - APWeb 2005. XXII, 1082 pages. 2005.

Vol. 3398: D.-K. Baik (Ed.), Systems Modeling and Simulation: Theory and Applications. XIV, 733 pages. 2005. (Subseries LNAI).

Vol. 3397: T.G. Kim (Ed.), Artificial Intelligence and Simulation. XV, 711 pages. 2005. (Subseries LNAI).

Vol. 3396: R.M. van Eijk, M.-P. Huget, F. Dignum (Eds.), Agent Communication. X, 261 pages. 2005. (Subseries LNAI).

Vol. 3395: J. Grabowski, B. Nielsen (Eds.), Formal Approaches to Software Testing. X, 225 pages. 2005.

Vol. 3394: D. Kudenko, D. Kazakov, E. Alonso (Eds.), Adaptive Agents and Multi-Agent Systems III. VIII, 313 pages. 2005. (Subseries LNAI).

Vol. 3393: H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, G. Taentzer (Eds.), Formal Methods in Software and Systems Modeling. XXVII, 413 pages. 2005.

Vol. 3391: C. Kim (Ed.), Information Networking. XVII, 936 pages. 2005.

Vol. 3390: R. Choren, A. Garcia, C. Lucena, A. Romanovsky (Eds.), Software Engineering for Multi-Agent Systems III. XII, 291 pages. 2005.

Vol. 3389: P. Van Roy (Ed.), Multiparadigm Programming in Mozart/OZ. XV, 329 pages. 2005.

Vol. 3388: J. Lagergren (Ed.), Comparative Genomics. VII, 133 pages. 2005. (Subseries LNBI).

Vol. 3387: J. Cardoso, A. Sheth (Eds.), Semantic Web Services and Web Process Composition. VIII, 147 pages. 2005.

Vol. 3386: S. Vaudenay (Ed.), Public Key Cryptography - PKC 2005. IX, 436 pages. 2005.

Vol. 3385: R. Cousot (Ed.), Verification, Model Checking, and Abstract Interpretation. XII, 483 pages. 2005.

Vol. 3383: J. Pach (Ed.), Graph Drawing. XII, 536 pages. 2005.

Vol. 3382: J. Odell, P. Giorgini, J.P. Müller (Eds.), Agent-Oriented Software Engineering V. X, 239 pages. 2005.

Vol. 3381: P. Vojtáš, M. Bieliková, B. Charron-Bost, O. Sýkora (Eds.), SOFSEM 2005: Theory and Practice of Computer Science. XV, 448 pages. 2005.

Vol. 3380: C. Priami, Transactions on Computational Systems Biology I. IX, 111 pages. 2005. (Subseries LNBI).

Vol. 3379: M. Hemmje, C. Niederee, T. Risse (Eds.), From Integrated Publication and Information Systems to Information and Knowledge Environments. XXIV, 321 pages. 2005.

Vol. 3378: J. Kilian (Ed.), Theory of Cryptography. XII, 621 pages. 2005.

Vol. 3377: B. Goethals, A. Siebes (Eds.), Knowledge Discovery in Inductive Databases. VII, 190 pages. 2005.

Vol. 3376: A. Menezes (Ed.), Topics in Cryptology – CT-RSA 2005. X, 385 pages. 2005.

Vol. 3375: M.A. Marsan, G. Bianchi, M. Listanti, M. Meo (Eds.), Quality of Service in Multiservice IP Networks. XIII, 656 pages. 2005.

Vol. 3374: D. Weyns, H.V.D. Parunak, F. Michel (Eds.), Environments for Multi-Agent Systems. X, 279 pages. 2005. (Subseries LNAI).

Vol. 3372: C. Bussler, V. Tannen, I. Fundulaki (Eds.), Semantic Web and Databases. X, 227 pages. 2005.

Vol. 3371: M.W. Barley, N. Kasabov (Eds.), Intelligent Agents and Multi-Agent Systems. X, 329 pages. 2005. (Subseries LNAI).

Vol. 3370: A. Konagaya, K. Satou (Eds.), Grid Computing in Life Science. X, 188 pages. 2005. (Subseries LNBI).

Vol. 3369: V.R. Benjamins, P. Casanovas, J. Breuker, A. Gangemi (Eds.), Law and the Semantic Web. XII, 249 pages. 2005. (Subseries LNAI).

Vol. 3368: L. Paletta, J.K. Tsotsos, E. Rome, G.W. Humphreys (Eds.), Attention and Performance in Computational Vision. VIII, 231 pages. 2005.

Vol. 3367: W.S. Ng, B.C. Ooi, A. Ouksel, C. Sartori (Eds.), Databases, Information Systems, and Peer-to-Peer Computing. X, 231 pages. 2005.

Vol. 3366: I. Rahwan, P. Moraitis, C. Reed (Eds.), Argumentation in Multi-Agent Systems. XII, 263 pages. 2005. (Subseries LNAI).

Vol. 3365: G. Mauri, G. Păun, M.J. Pérez-Jiménez, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing. IX, 415 pages. 2005.

Vol. 3363: T. Eiter, L. Libkin (Eds.), Database Theory - ICDT 2005. XI, 413 pages. 2004.

Vol. 3362: G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, T. Muntean (Eds.), Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. IX, 257 pages. 2005.

Vol. 3361: S. Bengio, H. Bourlard (Eds.), Machine Learning for Multimodal Interaction. XII, 362 pages. 2005.

Vol. 3360: S. Spaccapietra, E. Bertino, S. Jajodia, R. King, D. McLeod, M.E. Orlowska, L. Strous (Eds.), Journal on Data Semantics II. XI, 223 pages. 2005.

Vol. 3359: G. Grieser, Y. Tanaka (Eds.), Intuitive Human Interfaces for Organizing and Accessing Intellectual Assets. XIV, 257 pages. 2005. (Subseries LNAI).

Vol. 3358: J. Cao, L.T. Yang, M. Guo, F. Lau (Eds.), Parallel and Distributed Processing and Applications. XXIV, 1058 pages. 2004.

Vol. 3357: H. Handschuh, M.A. Hasan (Eds.), Selected Areas in Cryptography. XI, 354 pages. 2004.

Vol. 3356: G. Das, V.P. Gulati (Eds.), Intelligent Information Technology. XII, 428 pages. 2004.

Vol. 3355: R. Murray-Smith, R. Shorten (Eds.), Switching and Learning in Feedback Systems. X, 343 pages. 2005.

Vol. 3354: M. Margenstern (Ed.), Machines, Computations, and Universality. VIII, 329 pages. 2005.

Vol. 3353: J. Hromkovič, M. Nagl, B. Westfechtel (Eds.), Graph-Theoretic Concepts in Computer Science. XI, 404 pages. 2004.

Vol. 3352: C. Blundo, S. Cimato (Eds.), Security in Communication Networks. XI, 381 pages. 2005.

Vol. 3351: G. Persiano, R. Solis-Oba (Eds.), Approximation and Online Algorithms. VIII, 295 pages. 2005.

Vol. 3350: M. Hermenegildo, D. Cabeza (Eds.), Practical Aspects of Declarative Languages. VIII, 269 pages. 2005.

Vol. 3349: B.M. Chapman (Ed.), Shared Memory Parallel Programming with Open MP. X, 149 pages. 2005.

Vol. 3348: A. Canteaut, K. Viswanathan (Eds.), Progress in Cryptology - INDOCRYPT 2004. XIV, 431 pages. 2004.

Vol. 3347: R.K. Ghosh, H. Mohanty (Eds.), Distributed Computing and Internet Technology. XX, 472 pages. 2004.

Vol. 3346: R.H. Bordini, M. Dastani, J. Dix, A.E.F. Seghrouchni (Eds.), Programming Multi-Agent Systems. XIV, 249 pages. 2005. (Subseries LNAI).

Vol. 3345: Y. Cai (Ed.), Ambient Intelligence for Scientific Discovery. XII, 311 pages. 2005. (Subseries LNAI).

Vol. 3344: J. Malenfant, B.M. Østvold (Eds.), Object-Oriented Technology. ECOOP 2004 Workshop Reader. VIII, 215 pages. 2005.

Vol. 3343: C. Freksa, M. Knauff, B. Krieg-Brückner, B. Nebel, T. Barkowsky (Eds.), Spatial Cognition IV. Reasoning, Action, and Interaction. XIII, 519 pages. 2005. (Subseries LNAI).

Table of Contents

Invited Talk

Compilation

Parallelism

此为试读,需要完整PDF请访问: www.ertongbook.com

Memory Management

Program Transformations

Tool Demonstrations

Pointer Analysis

When Abstraction Fails

Andreas Zeller

Saarland University, Saarbrücken, Germany
zeller@cs.uni-sb.de

Abstract. Reasoning about programs is mostly deduction: the reasoning from the abstract model to the concrete run. Deduction is useful because it allows us to predict properties of future runs—up to the point that a program will never fail its specification. However, even such a 100% correct program may still show a problem: the specification itself may be problematic, or deduction required us to abstract away some relevant property. To handle such problems, deduction is not the right answer—especially in a world where programs reach a complexity that makes them indistinguishable from natural phenomena. Instead, we should enrich our portfolio by methods proven in natural sciences, such as observation, induction, and in particular experimentation. In my talk, I will show how systematic experimentation automatically reveals the causes of program failures—in the input, in the program state, or in the program code.

1 Introduction

I do research on how to debug programs. It is not that I am particularly fond of bugs, or debugging. In fact, I hate bugs, and I have spent far too much time on chasing and eradicating them. People might say: So, why don't you spend your research time on improving your specification, model checker, software process, architecture, or whatever the latest and greatest advance in science is. I answer: All of these help *preventing* errors, which is fine. But none can prevent surprises. And I postulate that surprises are unavoidable, that we have to teach people how to deal with them and to set things straight after the fact.

As one of my favorite examples, consider the *sample* program in Fig. 1 on the following page. Ideally, the `sample` program sorts its arguments numerically and prints the sorted list, as in this run ($r_✓$):

```
sample 9 8 7   ⇒   7 8 9
```

With certain arguments, `sample` fails (run $r_✗$):

```
sample 11 14   ⇒   0 11
```

Surprise! While the output of `sample` is still properly sorted, the output is not a permutation of the input—somehow, a zero value has sneaked in. What is the defect that causes this failure?

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 1–9, 2005.
© Springer-Verlag Berlin Heidelberg 2005

```
1  /* sample.c -- Sample C program to be debugged */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  static void shell_sort(int a[], int size)
7  {
8      int i, j;
9      int h = 1;
10     do {
11         h = h * 3 + 1;
12     } while (h <= size);
13     do {
14         h /= 3;
15         for (i = h; i < size; i++)
16         {
17             int v = a[i];
18             for (j = i; j >= h && a[j - h] > v; j -= h)
19                 a[j] = a[j - h];
20             if (i != j)
21                 a[j] = v;
22         }
23     } while (h != 1);
24 }
25
26 int main(int argc, char *argv[])
27 {
28     int  i = 0;
29     int *a = NULL;
30
31     a = (int *)malloc((argc - 1) * sizeof(int));
32     for (i = 0; i < argc - 1; i++)
33         a[i] = atoi(argv[i + 1]);
34
35     shell_sort(a, argc);
36
37     for (i = 0; i < argc - 1; i++)
38         printf("%d ", a[i]);
39     printf("\n");
40
41     free(a);
42     return 0;
43 }
```

Fig. 1. The *sample* program (almost) sorts its arguments

In principle, debugging a program like `sample` is easy. Initially, some programmer has created a *defect*—an error in the code. When executed, this defect causes an *infection*—an error in the program state. (Other people call this a *fault*, but I prefer the term *infection*, because the error propagates across later states, just like an infection.) When the infection finally reaches a point where it can be observed, it becomes a *failure*—in our case, the zero in the output. Given that a failure has already occurred, it is the duty of the programmer to trace back this cause-effect chain of infections back to the defect where it originated—the defect that caused the failure.

As an experienced programmer, you may be able to walk your way through the source code in Fig. 1 and spot the defect. When it comes to doing so in a general, systematic, maybe even automated way, we quickly run into trouble, though. The difficulty begins with the terms. What do we actually mean when we say "the defect that caused the failure"? What are we actually searching for?

2 Errors are Easy to Detect, But Generally Impossible to Locate

An *error* is a deviation from what is correct, right, or true. To tell that something is erroneous thus requires a specification of what is correct, right, or true. This can be applied to output, input, state, and code:

Errors in the output. An externally visible error in the program behavior is called a *failure*. Our investigation starts when we determine (or decide) that this is the case.

Errors in the input. For the program *input*, we typically know what is valid and what not, and therefore we can determine whether an input is erroneous or not. If the program shows a failure, and if the input was correct, we know the program as a whole is incorrect.

Errors in the program state. It is between input and output that things start to get difficult. When it comes to the program *state*, we frequently have specifications that allow us to catch infections—for instance, when a pre- or postcondition is violated. Types can be seen as specifications that detect and prevent illegal variable values. Common programming errors, such as buffer overflows or null pointer dereferences, can be specified and detected at compile time.

Errors in the code. Unfortunately, specifications apply only to *parts* of the program state: conditions apply to selected moments in time; types allow a wide range of values; tools can only check for common errors. Therefore, there will always be parts of the state for which correctness is not specified. But if we do not know whether a variable value is correct, we cannot tell whether the code that generated this value is correct. Therefore, we cannot exactly track down the moment the value got infected, and therefore, we cannot locate the defect that caused the failure.

Of course, we can catch errors by simply specifying more. A specification that covers each and every aspect of a program state would detect every single error. Unfortunately, such a specification would ne no less complex and error-prone than the program itself.

In practice, it is the programmer who *decides* what is right upon examining the program—and fixes the program according to this *implied* specification. In such a cases, deciding which part of a program is in error can only be told after the decision has been made and the error has been fixed. Once we know the correct, right, and true code, we can thus tell the defect as a deviation from the corrected code. In other words, *locating a defect is equivalent to writing a correct program.* And we know how hard this is.

3 Causes Need Not be Errors, But Can Easily be Located

While it may be hard to pinpoint an error, the concept of *causality* is far less ambiguous. In general, a *cause* is an event that precedes another event (the

effect), such that the effect would not have occurred without the cause. For programs, this means that any aspect of an execution causes a failure if it can be altered such that the failure no longer occurs. This applies to input, state, and code:

Causes in the input. We can change the input of the `sample` program from `11 14` (run $r_✗$) to `7 8 9` (run $r_✓$), and the failure no longer occurs. Hence, we know that the input causes the failure.

One may argue that in any program, the input determines the behavior and thus eventually causes any failure. However, it may be only parts of the input that are relevant. For instance, if we run `sample` with `11`, we find that it is the additional `14` argument which causes the failure:

```
sample 11   ⇒   11
```

Causes in program state. If we can change some variable during execution such that the failure no longer occurs, we know that the variable caused the failure.

Again, consider the failing `sample` run $r_✗$. We could use an interactive debugger and stop the program at `main()` (Line 28), change `argc` from 2 to 1, and resume execution. We would find an output of `11`, and thus find out that the value of `argc` caused the failure.

As we can see from this example, a cause does not imply an error: The value of `argc` probably is correct with respect to some implied specification; yet, it is tied to the failure.

Causes in the code. All variable values are created by some statement in the code; and thus, there are statements which cause values which again cause failures.

In the `sample` program, there is a statement which exactly does that, and which can (and should) be changed to make the failure no longer occur. The interesting aspect is that we can find that statement from the causes in the program state. If we can find a failure cause in the program state, we can trace it back to the statement which generated it.

Once again, it is important to note that causes and errors are two orthogonal concepts. We can tell an error without knowing whether it is a cause for the failure at hand, and we can tell a cause without knowing whether it is an error. In the absence of a detailed specification, though, we must rely on causality to narrow down those statements which caused the error—in the hope that the defect is among them.

4 Isolating Failure Causes with Automatic Experiments

Verifying that something is a cause cannot be done by deduction. We need at least two *experiments:* One with the cause, and one without; if the effect occurs only with the cause, we're set. This implies that we need two runs of the

此为试读,需要完整PDF请访问: www.ertongbook.com