Published by the IEEE Computer Society
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314

IEEE Computer Society Order Number P3675
ISBN 978-0-7695-3675-0
Library of Congress Number 2009902625
BMS Part Number CFP0999G-PRT

ISBN 978-0-7695-3675-0

SSBSE 2009 • 1st International Symposium on Search Based Software Engineering

1st International Symposium on Search Based Software Engineering

Cumberland Lodge, Windsor, UK
13 - 15 May, 2009

www.ssbse.org

SPONSORED BY

berner&mattner
optimizing your development

SOGETI

EPSRC Engineering and Physical Sciences Research Council

Proceedings

1st International Symposium on Search Based Software Engineering

SSBSE 2009

Proceedings

1st International Symposium on Search Based Software Engineering

SSBSE 2009

13-15 May 2009
Windsor, Berkshire, United Kingdom

Edited by
Massimiliano Di Penta and Simon Poulding

Sponsored by

berner&mattner
optimizing your development

SOGETI

EPSRC Engineering and Physical Sciences Research Council

CPS Conference Publishing Services

Los Alamitos, California

Washington • Tokyo

IEEE computer society

IEEE Computer Society Order Number P3675
ISBN-13: 978-0-7695-3675-0
BMS Part # CFP0999G-PRT
Library of Congress 2009902625

Additional copies may be ordered from:

IEEE Computer Society
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314
Tel: + 1 800 272 6657
Fax: + 1 714 821 4641
http://computer.org/cspress
csbooks@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: + 1 732 981 0060
Fax: + 1 732 981 9667
http://shop.ieee.org/store/
customer-service@ieee.org

IEEE Computer Society
Asia/Pacific Office
Watanabe Bldg., 1-4-2
Minami-Aoyama
Minato-ku, Tokyo 107-0062
JAPAN
Tel: + 81 3 3408 3118
Fax: + 81 3 3408 3553
tokyo.ofc@computer.org

Individual paper REPRINTS may be ordered at: <reprints@computer.org>

Editorial production by Silvia Ceballos
Cover art production by Mark Bartosik
Printed in the United Stated of America by Applied Digital Imaging

IEEE Computer Society
***Conference Publishing Services* (CPS)**
http://www.computer.org/cps

Table of Contents

1st International Symposium on Search Based Software Engineering SSBSE 2009

Session 3- Software Evolution

Session 4- Short Papers

Session 5- Testing

Message from the General Chair

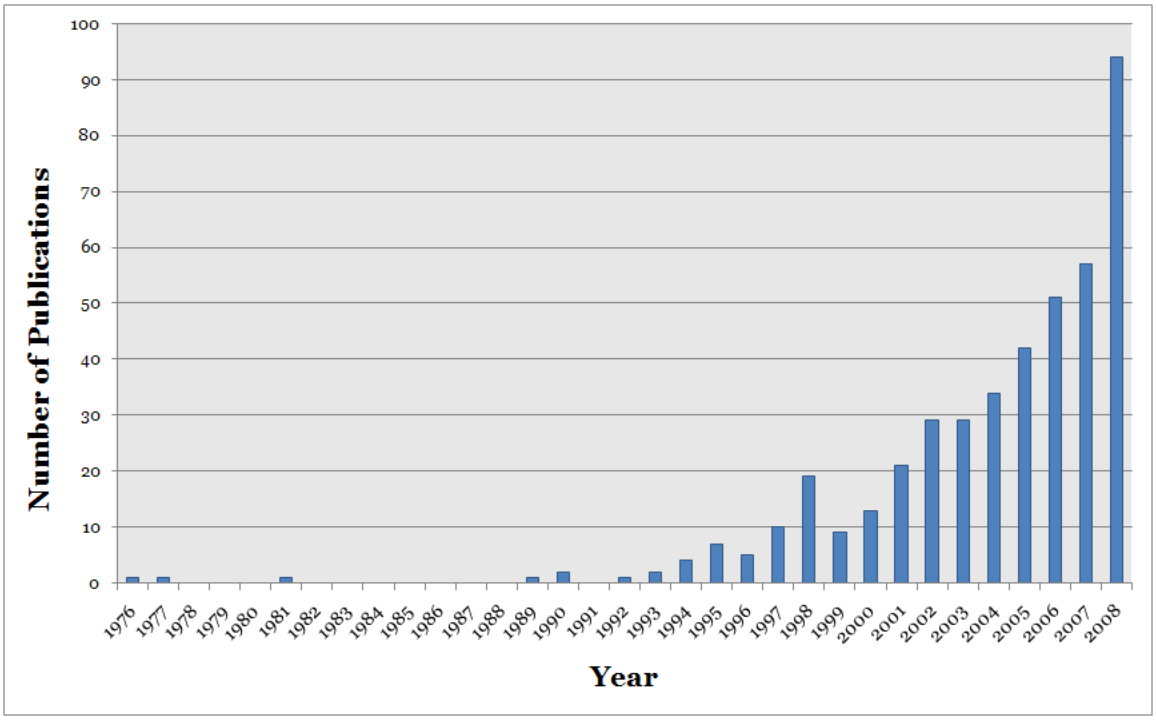
Welcome to SSBSE 2009, the 1st International Symposium on Search Based Software Engineering. I would like to take this opportunity to explain a little about SSBSE and its motivation and organisation and to thank those involved.

SSBSE 2009 is a unique event that combines aspects of a conference, a workshop and a retreat. Through this unique blend of attributes SSBSE seeks to nurture and develop the growing international research community working on Search Based Software Engineering. It is envisaged that SSBSE will become an annual event, sometimes standing alone and sometimes running alongside an existing conference in co-location to promote and encourage cross fertilization. This being the first SSBSE, the organising committee and I have paid particular attention to community building and outreach to the wider Software Engineering community.

The recent rapid growth in Search Based Software Engineering (SBSE) publications can be seen from the bar graph below, which is taken from the repository of papers on SBSE at

http://www.sebase.org/sbse/publications/

The repository includes analysis of topics covered, growth trends (such as this) as well as a searchable database of all publications on SBSE.



As this graph demonstrates there is a rapidly growing increase in interest in SBSE. The SBSE philosophy is that Software Engineering often considers problems that involve finding optimal or near optimal balances between competing and potentially conflicting goals. There is often a bewilderingly large set of choices and finding good solutions can be hard. For instance, the following is an illustrative list of software engineering optimization questions:

- What is the smallest set of test cases that cover all program branches?
- What is the best way to structure the architecture of the system?
- Which requirements best balance cost and customer satisfaction?
- What is the best allocation of resources to this development project?
- What is the best sequence of refactoring steps to apply to this system?

Answers to these questions might be expected from the literature on testing, design, requirements engineering, software engineering management and refactoring respectively. It would appear at first sight, that questions such as these involve different aspects of software engineering, that they would be covered by different conferences and specialized journals and that they would have little in common.

However, all of these questions are essentially *optimization* questions. As such, they are typical of the kinds of problem for which SBSE is well adapted, and with which each has been successfully formulated as a search based optimization problem. SSBSE therefore re-unites areas of Software Engineering that have long been studied in isolation. It seeks to overcome the potential for 'silo mentality' that the necessary expansion of Software Engineering has forced upon the growing Software Engineering community.

Through SBSE, many cross-cutting concerns that apply to several otherwise disparate sub-areas of Software Engineering can be considered together. These different areas are often united by SBSE nomenclature, such as the form of representation or the properties of the search landscape. It is this generic nature of SBSE that is one of its attractions and one of the key motivations for the SSBSE 2009 symposium.

The symposium will allow researchers and practitioners to explore commonalities in the underlying optimization problems and to generalize from these, allowing the development of deeper theoretical foundations and cross fertilization of solution methods and techniques. The symposium will cut across traditional software engineering sub-discipline boundaries with the common goal of seeking formulations that facilitate optimization.

Though this is the first International Symposium on Search Based Software Engineering, it builds upon a longer history of workshops, tracks, and special issues on SBSE. In December 2001, a special issue on Search Based Software Engineering of the Journal of Information and Software Technology was published by Elsevier. The papers reflected the (already, at the time) wide diversity of work within the embryonic field of SBSE. The inclusion of an overview paper and a survey paper by Darrell Whitley made the special issue a self-contained 'booklet' describing the nascent area of Search-Based Software Engineering.

In July 2002, the GECCO conference held its first track on Search-Based Software Engineering. The program chair for the track was Joachim Wegener. The adoption of SBSE as a track in its own right by the GECCO organisers demonstrated the high regard the international Evolutionary Computation community has for the new application area of SBSE. The SBSE track of the GECCO conference is now in its 7th year.

In 2003 there was an informal UK workshop on SBSE, also held here, at the Cumberland Lodge. This event had over 30 attendees from the UK, Europe, USA and New Zealand. The workshop proved that there was a growing international community of researchers who were prepared to travel to a standalone workshop (not co-located with any other event). In some ways, this was a pre-cursor to the SSBSE symposium. It is to be hoped that the lively discussion-centric format of the workshop will be retained for the symposium.

In 2007 there were two workshops on SBSE held in London, one at UCL as part of GECCO 2007 and one in the CREST centre at King's College, London. Despite being held on a Sunday in what there was of a UK summer in 2007, the CREST SBSE workshop attracted 39 researchers and practitioners of SBSE from many different locations, including North America and Europe. Presentations from the event are available on the web, under the auspices of the ASTReNet http://www.dcs.kcl.ac.uk/staff/zheng/astrenet.

The evidence for the future growth of SBSE research is also strong and promising. In October 2008, a special issue of the Journal Computers and Operations Research (Volume 35, issue 10) on SBSE appeared edited by Walter Gutjahr and Mark Harman. This is significant because the journal is an OR journal and so the publication of this special issue reflects the growing interest in SBSE, not just from the Software Engineering community, but also from the optimization community as well. Also in 2008, there was a special issue of the Journal of Software Maintenance and Evolution edited by Giulio Antoniol, Massimiliano Di Penta and Mark Harman.

In 2009 there will be a special issue of IEEE Transactions on Software Engineering, edited by Afshin Mansouri and Mark Harman, for which the call is now closed and there is a current call for papers for a special issue of Software – Practice and Experience on SBSE, to be edited by Iain Bate and Simon Poulding. There will also be a special issue of the Journal of Empirical Software Engineering featuring extended selected papers from SSBSE 2009.

As can be seen the history of events and activities upon which SSBSE builds, stretches back over more than five years, while the future of this exciting field of study is assured by the continuing development of events and special issues devoted to SBSE.

Bringing about the first international Symposium on SSBSE has involved a lot of work from a lot of very talented and dedicated people to whom I would like to offer thanks in this message.

I would like to pay special thanks to Massimiliano Di Penta and Simon Poulding, the program co-chairs for all their work in bringing this imaginative and exciting program together. They also worked behind the scenes on many aspects of the event, including the proceedings, and supported by Per Kristian Lehre, David White and Paul Emberson the publicity and excellent website for SSBSE 2009. I would also like to extend a warm thanks to the program committee, whose expertise and rigour ensured high quality reviewing for the papers submitted.

Myra Cohen not only managed the PhD track for SSBSE 2009, but also was closely involved with the program chairs in the construction of the program for the event.

The industrial sponsors Berner and Mattner in Berlin and Sogeti UK in London have helped SSBSE 2009 to maintain strong and close links to industry and to reduce the cost of the event to the participants. Their involvement is a very welcome indication of industrial interest and uptake of SBSE and their sponsorship is particularly valued in the difficult economic climate in which we all now find ourselves.

As well as industrial sponsorship, SSBSE 2009 also received the kind and valuable support of the Engineering and Physical Sciences Research Council (the EPSRC). The EPSRC is the UK government funding body for the sciences and engineering.

I am very grateful to the international community for their participation and I would like to extend a very warm welcome to all our visitors to SSBSE. I hope that you find the technical content stimulating and the environment perfect for establishing and deepening research collaborations.

SSBSE 2009 has three outstanding keynote speakers, Enrique Alba, Lionel Briand and David Goldberg. It is also fortunate to have a panel of internationally leading Software Engineers who will provide feedback through the experts' panel. I would like to welcome and thank our three illustrious keynotes and our five expert panellists: Norman Fenton, Anthony Finkelstein, Paola Inverardi, Mary Lou Soffa and Ian Sommerville.

A complex event such as SSBSE 2009 cannot be organised without the help of a committed team of organisers. The staff at the Cumberland Lodge have been marvellous in their dedicated assistance. The event would not be nearly so appealing without their support, kindness and professionalism. I would like to extend a very warm thanks to Matthew Hancock, Jane Bailey and all the staff at the Cumberland Lodge for their hospitality.

The staff at the IEEE have also been very helpful in using their resources to produce the proceedings. I would like to thank Andrea Thibault-Sanchez and Silvia Ceballos and all the staff at the IEEE for this support. I am also very grateful to Lionel Briand, the Editor in Chief of the Journal of Empirical Software Engineering for agreeing to a special issue of extended selected papers from SSBSE 2009.

The local arrangements and planning of the event could not have been achieved without the help of Zheng Li and Jian Ren.

Thank you all very much for your hard work and support.

Next year, SSBSE 2010 will be held in Italy. The organising committee will be Lionel Briand, John Clark, Max Di Penta and Simon Poulding. After 2010, SBSE will seek co-location with other leading Software Engineering conferences. We plan to establish a charter for the SSBSE symposia series and a steering committee. If you would like to be involved, please talk to any of the organising committee for SSBSE 2009 or SSBSE 2010.

I hope you have a very productive, stimulating and optimal symposium at this and future symposia.

Mark Harman,
CREST: Centre for Research on Evolution, Search and Testing,
Department of Computer Science,
King's College London.
March 2009.

Message from the Program Co-chairs

On behalf of the SSBSE 2009 program committee, we welcome you to Cumberland Lodge and to the 1st International Symposium on Search Based Software Engineering.

It has been a privilege to be part of the organising committee for this, the first ever symposium dedicated to search based software engineering. We are particularly grateful for the widespread participation and support from the SBSE community. 26 papers were submitted to the research and PhD tracks, with authors from institutions in 14 countries: the community is truly international. We would like to thank all the authors who submitted such high quality manuscripts.

We wanted to ensure a thorough and constructive review process for the symposium. All submitted papers were reviewed by at least three experts in the field: in total, 89 reviews were performed. Authors were given the opportunity to clarify any misunderstandings and this enabled reviewers to refine their review comments. After extensive discussion, 9 manuscripts were accepted as full papers, 5 as short papers, and 3 were accepted to the PhD track. Providing detailed and constructive comments is a difficult task but a vitally important one if we, as a community, wish to encourage the publication of high quality, rigorous and relevant research. We would therefore like to express our gratitude to the members of the program committee for taking the time to do so.

We have designed the symposium program with the exchange of ideas in mind. In technical sessions, authors are encouraged to give concise presentations covering the novel ideas and key results that will be of particular interest to the audience. The presentations will be followed by a general discussion on the session topic, and we invite all attendees to actively participate in these discussions. We are providing facilities for ad-hoc presentations during the discussion, and will capture the points raised and distribute them to the attendees after the symposium.

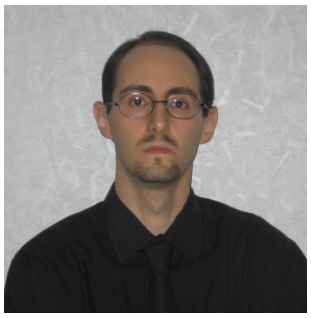
Technical papers are only part of the SSBSE 2009 program. We are honoured to have three outstanding keynote speakers: Lionel C. Briand, David E. Goldberg, and Enrique Alba. All are internationally renowned researchers in their respective fields, and we are sure that we speak for all attendees when we say that we are looking forward to learning from their keynote talks.

Last but not least, the program includes what promises to be an extremely stimulating panel session, involving five of the most prominent software engineering scientists: Norman Fenton, Anthony Finkelstein, Paola Inverardi, Mary Lou Soffa, and Ian Sommerville. It is a pleasure to be able to include such a distinguished panel of speakers in the symposium program. We believe that their insights on the status and future of search based software engineering will help to inform the direction that we take as a community.

Many people have been involved in organising the program for the symposium and we would like to take the time to acknowledge them here. We have benefited greatly from the support and guidance of the general chair, Mark Harman; without his knowledge and enthusiasm, this symposium would not have been possible. We would also like to extend our thanks to Myra Cohen who assisted us in developing the program, far beyond the role of PhD track chair that she signed up to. The publicity and website for the symposium are the result of the hard work of Paul Emberson, Per Kristian Lehre, and David White. Silvia Ceballos at the IEEE Computer Society has assisted us in preparing the proceedings, and Jian Ren has worked with us on incorporating local arrangements into the program. We would like to thank them all.

Please take advantage of the unique opportunity that this symposium affords: a coming-together of leading researchers and practitioners in a forum dedicated to SBSE. We also suggest you find time to explore the Lodge and the surrounding parkland. Please do not hesitate to give us comments on the symposium, and suggestions for the future.

Above all, we trust that you enjoy SSBSE 2009 and that we will see you again in 2010.



Simon Poulding, University of York, and Massimiliano Di Penta, University of Sannio.
March 2009

SSBSE 2009 Committees

Organising Committee

General Chair
Mark Harman, King's College London, UK

Program Co-chairs
Massimiliano Di Penta, University of Sannio, Italy
Simon Poulding, University of York, UK

PhD Student Track Chair
Myra B. Cohen, University of Nebraska – Lincoln, USA

Local Arrangements and Registration Chair
Jian Ren, King's College London, UK

Publicity Chair
Per Kristian Lehre, University of Birmingham, UK

Website
Paul Emberson, University of York, UK
David R. White, University of York, UK

Program Committee

Giuliano Antoniol, École Polytechnique de Montréal, Canada
Andrea Arcuri, University of Birmingham, UK
Rami Bahsoon, University of Birmingham, UK
Andre Baresel, Q-vi Tech GmbH, Germany
Iain Bate, University of York, UK
Fevzi Belli, University of Paderborn, Germany
Leonardo Bottaci, University of Hull, UK
Lionel C. Briand, Simula Research Laboratory, Norway
Gerardo Canfora, University of Sannio, Italy
Francisco Chicano, University of Málaga, Spain
John Clark, University of York, UK
José J. Dolado, University of the Basque Country, Spain
Paul Emberson, University of York, UK
Yaniv Eytan, University of Illinois at Urbana-Champaign, USA
Vahid Garousi, University of Calgary, Canada
Walter Gutjahr, University of Vienna, Austria
Youssef Hassoun, King's College London, UK
Rob Hierons, Brunel University, UK
Gregory Kapfhammer, Allegheny College, USA
Bogdan Korel, Illinois Institute of Technology, USA
Yvan Labiche, Carleton University, Canada
Kiran Lakhotia, King's College London, UK
Per Kristian Lehre, University of Birmingham, UK
José Carlos Maldonado, Universidade de São Paulo, Brazil
Spiros Mancoridis, Drexel University, USA

Phil McMinn, University of Sheffield, UK
Mel Ó Cinnéide, University College Dublin, Ireland
Marek Reformat, University of Alberta, Canada
Marc Roper, University of Strathclyde, UK
Guenther Ruhe, University of Calgary, Canada
Ramón Sagarna, University of Birmingham, UK
Martin Shepperd, Brunel University, UK
Paolo Tonella, Fondazione Bruno Kessler – IRST, Italy
Amund Tveit, Google, Norway
Tanja Vos, Instituto Tecnológico de Informática, Spain
Joachim Wegener, Berner & Mattner, Germany
David R. White, University of York, UK
Tao Xie, North Carolina State University, USA
Shin Yoo, King's College London, UK
Yuanyaun Zhang, King's College London, UK

Additional Reviewers

Arthur Baars
Rosana Braga
Shanshan Hou
Adenilso Simão
Kunal Taneja
Suresh Thummalapenta

Program Schedule – Wednesday, May 13

12:30	Lunch
14:00 – 14:30	Welcome and Introduction
14:30 – 16:00	Keynote Model-driven Development and Search-based Software Engineering: An Opportunity for Research Synergy *Lionel C. Briand, Simula Research Lab and University of Oslo, Norway*
16:00	Refreshments
16:30 – 18:00	Session 1 – Testing and Defect Prediction Chair: *Rob Heirons, Brunel University, UK* Search-Based Testing of Ajax Web Applications *Alessandro Marchetto and Paolo Tonella* An Improved Meta-Heuristic Search for Constrained Interaction Testing *Brady J. Garvin, Myra B. Cohen and Matthew B. Dwyer* Evolution and Search Based Metrics to Improve Defects Prediction *Segla Kpodjedo, Filippo Ricca, Giuliano Antoniol and Philippe Galinier*
19:00	Dinner
	Drinks in the bar, sponsored by Berner and Mattner, Germany

Program Schedule – Thursday, May 14

08:15	Breakfast
09:00 – 10:30	Keynote Not Your Grandmother's Genetic Algorithm *David E. Goldberg, University of Illinois at Urbana-Champaign, USA*
10:30	Refreshments
11:00 – 12:30	Session 2 – PhD Student Track Chair: *Myra B. Cohen, University of Nebraska – Lincoln, USA* Search-Based Prediction of Fault Count Data *Wasif Afzal, Richard Torkar and Robert Feldt* On Search Based Software Evolution *Andrea Arcuri* Local Search-Based Refactoring as Graph Transformation *Fawad Qayum and Reiko Heckel*
12:30	Lunch
14:00 – 15:30	Session 3 – Software Evolution Chair: *Filippo Ricca, University of Genoa, Italy* A Study of the Multi-Objective Next Release Problem *J. J. Durillo, Y. Zhang, E. Alba and A. J. Nebro* Dynamic Architectural Selection: A Genetic Algorithm Based Approach *Dongsun Kim and Sooyong Park* On the Use of Discretized Source Code Metrics for Author Identification *Maxim Shevertalov, Jay Kothari, Edward Stehle and Spiros Mancoridis*
15:30	Refreshments
16:00 – 18:00	Software Engineering Experts' Panel – The Status and Future of SBSE in the Software Engineering Community *Norman Fenton, Queen Mary, University of London, UK* *Anthony Finkelstein, University College London, UK* *Paola Inverardi, Università dell'Aquila, Italy* *Mary Lou Soffa, University of Virginia, USA* *Ian Sommerville, University of St Andrews, UK*
19:00	Dinner
	Drinks in the bar, sponsored by Sogeti, UK

Program Schedule – Friday, May 15

08:00	Breakfast
08:45 – 10:15	Keynote How Can Metaheuristics Help Software Engineers *Enrique Alba, Universidad de Málaga, Spain*
10:15	Refreshments
10:45 – 12:15	Session 4 – Short Papers Chair: *Iain Bate, University of York, UK* Search-Based Testing with in-the-loop Systems *Joachim Wegener and Peter M. Kruse* A Testability Transformation Approach for State-Based Programs *AbdulSalam Kalaji, Robert M. Hierons and Stephen Swift* Searching for Rules to find Defective Modules in Unbalanced Datasets *D. Rodríguez, J.C. Riquelme, R. Ruiz, and J.S. Aguilar-Ruiz* Formal model simulation: can it be guided? *Thang H. Bui and Albert Nymeyer* How can optimization models support the maintenance of component-based software? *Vittorio Cortellessa and Pasqualina Potena*
12:15	Lunch
13:00 – 14:30	Session 5 – Testing Chair: *Phil McMinn, University of Sheffield, UK* WCET Analysis of Modern Processors Using Multi-Criteria Optimisation *Usman Khan and Iain Bate* Full Theoretical Runtime Analysis of Alternating Variable Method on the Triangle Classification Problem *Andrea Arcuri* Widening the Goal Posts: Program Stretching to Aid Search Based Software Testing *Kamran Ghani and John A. Clark*
14:30	Refreshments
15:00 – 16:00	Session 6 – Fast Abstracts Chair: *Mark Harman, King's College London, UK*
16:00 – 16:15	Concluding Remarks

Keynote Speakers

Program Schedule – Wednesday, May 13

12:30	Lunch
14:00 – 14:30	Welcome and Introduction
14:30 – 16:00	Keynote Model-driven Development and Search-based Software Engineering: An Opportunity for Research Synergy *Lionel C. Briand, Simula Research Lab and University of Oslo, Norway*
16:00	Refreshments
16:30 – 18:00	Session 1 – Testing and Defect Prediction Chair: *Rob Heirons, Brunel University, UK* Search-Based Testing of Ajax Web Applications *Alessandro Marchetto and Paolo Tonella* An Improved Meta-Heuristic Search for Constrained Interaction Testing *Brady J. Garvin, Myra B. Cohen and Matthew B. Dwyer* Evolution and Search Based Metrics to Improve Defects Prediction *Segla Kpodjedo, Filippo Ricca, Giuliano Antoniol and Philippe Galinier*
19:00	Dinner
	Drinks in the bar, sponsored by Berner and Mattner, Germany

Program Schedule – Thursday, May 14

08:15	Breakfast
09:00 – 10:30	Keynote Not Your Grandmother's Genetic Algorithm *David E. Goldberg, University of Illinois at Urbana-Champaign, USA*
10:30	Refreshments
11:00 – 12:30	Session 2 – PhD Student Track Chair: *Myra B. Cohen, University of Nebraska – Lincoln, USA* Search-Based Prediction of Fault Count Data *Wasif Afzal, Richard Torkar and Robert Feldt* On Search Based Software Evolution *Andrea Arcuri* Local Search-Based Refactoring as Graph Transformation *Fawad Qayum and Reiko Heckel*
12:30	Lunch
14:00 – 15:30	Session 3 – Software Evolution Chair: *Filippo Ricca, University of Genoa, Italy* A Study of the Multi-Objective Next Release Problem *J. J. Durillo, Y. Zhang, E. Alba and A. J. Nebro* Dynamic Architectural Selection: A Genetic Algorithm Based Approach *Dongsun Kim and Sooyong Park* On the Use of Discretized Source Code Metrics for Author Identification *Maxim Shevertalov, Jay Kothari, Edward Stehle and Spiros Mancoridis*
15:30	Refreshments
16:00 – 18:00	Software Engineering Experts' Panel – The Status and Future of SBSE in the Software Engineering Community *Norman Fenton, Queen Mary, University of London, UK* *Anthony Finkelstein, University College London, UK* *Paola Inverardi, Università dell'Aquila, Italy* *Mary Lou Soffa, University of Virginia, USA* *Ian Sommerville, University of St Andrews, UK*
19:00	Dinner
	Drinks in the bar, sponsored by Sogeti, UK

Program Schedule – Friday, May 15

08:00	Breakfast
08:45 – 10:15	Keynote How Can Metaheuristics Help Software Engineers *Enrique Alba, Universidad de Málaga, Spain*
10:15	Refreshments
10:45 – 12:15	Session 4 – Short Papers Chair: *Iain Bate, University of York, UK* Search-Based Testing with in-the-loop Systems *Joachim Wegener and Peter M. Kruse* A Testability Transformation Approach for State-Based Programs *AbdulSalam Kalaji, Robert M. Hierons and Stephen Swift* Searching for Rules to find Defective Modules in Unbalanced Datasets *D. Rodríguez, J.C. Riquelme, R. Ruiz, and J.S. Aguilar-Ruiz* Formal model simulation: can it be guided? *Thang H. Bui and Albert Nymeyer* How can optimization models support the maintenance of component-based software? *Vittorio Cortellessa and Pasqualina Potena*
12:15	Lunch
13:00 – 14:30	Session 5 – Testing Chair: *Phil McMinn, University of Sheffield, UK* WCET Analysis of Modern Processors Using Multi-Criteria Optimisation *Usman Khan and Iain Bate* Full Theoretical Runtime Analysis of Alternating Variable Method on the Triangle Classification Problem *Andrea Arcuri* Widening the Goal Posts: Program Stretching to Aid Search Based Software Testing *Kamran Ghani and John A. Clark*
14:30	Refreshments
15:00 – 16:00	Session 6 – Fast Abstracts Chair: *Mark Harman, King's College London, UK*
16:00 – 16:15	Concluding Remarks

Not Your Grandmother's Genetic Algorithm

David E. Goldberg
University of Illinois at Urbana-Champaign, USA

Genetic algorithms (GAs)—search procedures inspired by the mechanics of natural selection and genetics—have been increasingly applied across the spectrum of human endeavor, but some researchers mistakenly think of them as slow, unreliable, and without much theoretical support.

This talk briefly introduces GAs, but quickly shifts to a line of work that has succeeded in supporting GA mechanics with bounding design theory that has been used to demonstrate GA scalability, speed, and range of reliable applicability. Key elements of this theory are discussed to give insight into this accomplishment and to make the point that fast, scalable GAs may also be viewed as first-order models of human innovative or inventive processes. The talk highlights recent results in breaking the billion-variable optimization barrier for the first time, and points to a variety of opportunities for efficiency enhancement that should be useful in the application of genetic algorithms to a variety of software engineering problems.

How Can Metaheuristics Help Software Engineers

Enrique Alba
Universidad de Málaga, Spain

Among the many fields of interest appearing in recent years, research in search based software engineering (SBSE) seems to have a very large potential. The base idea of modeling software engineering problems as optimization or search tasks opens the door to the utilization of a vast set of solvers. These techniques have been there around in Computer Science from its very beginning, since searching and optimizing are in the core of almost every aspect of life, and for sure in any sort of engineering application.

Metaheuristics are approximate algorithms that can be used to face complex (e.g. NP-hard) problems of large dimensions, highly constrained, with mixed types of variables, and in general with complex search landscapes. Either in isolation or hybridized, evolutionary algorithms, ant colony optimization, particle swarm optimization, and traditional simulated annealing are powerful search procedures that an engineer can use to find high quality solutions to his/her problem.

This keynote will be devoted to present the basic behavior of metaheuristics and to show how they can be applied to actual software engineering problems. We will discuss the main features of both, techniques and applications, to draw an efficient and accurate cross fertilization among them. Intelligent systems, operations research, computer science, and software engineering, all working together form a new field where many past difficult problems can be solved and where the present frontiers can be expanded by using new tools and research approaches.

Search-Based Testing of Ajax Web Applications

Alessandro Marchetto and Paolo Tonella
Fondazione Bruno Kessler - IRST
38050 Povo, Trento, Italy
marchetto|tonella@fbk.eu

Abstract

Ajax is an emerging Web engineering technology that supports advanced interaction features that go beyond Web page navigation. The Ajax technology is based on asynchronous communication with the Web server and direct manipulation of the GUI, taking advantage of reflection. Correspondingly, new classes of Web faults are associated with Ajax applications.

In previous work, we investigated a state-based testing approach, based on semantically interacting events. The main drawback of this approach is that exhaustive generation of semantically interacting event sequences limits quite severely the maximum achievable length, while longer sequences would have higher fault exposing capability. In this paper, we investigate a search-based algorithm for the exploration of the huge space of long interaction sequences, in order to select those that are most promising, based on a measure of test case diversity.

Keywords: Web Testing, Ajax Applications and Search-based Software Engineering.

1 Introduction

Web applications developed for the so-called "Future Internet" are expected to offer a richer user experience than current hypermedia navigation. The client is supposed to become a rich, interactive and highly responsive environment, offering information through advanced, adaptive means (e.g., 3D graphics). Among the technologies that are being developed to implement this vision, Ajax is one of the most promising and mature. Ajax overcomes the synchronous request-response protocol used by traditional Web applications by supporting asynchronous requests, which leave the user interface active and responsive. Combined with the possibility to update a Web page dynamically through the DOM (Document Object Model, [1]), Ajax promises to be a core component of the Future Internet technologies.

While improving the functionalities and interaction offered to the users, Ajax poses novel, additional problems with respect to those already known in the Web testing area [2, 4, 10]. Asynchronous requests and responses may interleave in an unexpected way, so Ajax programmers need to carefully program the concurrency provided by Ajax. Dynamic page update is another potential source of faults that are specific of Ajax applications. In fact, the code may be written according to wrong assumptions about the DOM state (e.g., an HTML element, such as a form or a table, is assumed to exist in the current page, while it is not there).

Recently, Marchetto *et al.* [8] investigated the use of state-based testing for AJAX Web applications and particularly focused on the specific faults introduced by this technology. The technique is based on dynamic extraction of a finite state machine for an Ajax application and its analysis with the aim of identifying sets of test cases based on *semantically interacting events* (Yuan *et al.* [13]). Empirical evidence [8, 12, 13] shows the effectiveness of this kind of technique in finding faults. Automatically generated test cases are comparable to those obtained by careful functional testing, manually performed by expert testers. Unfortunately, one of the main drawbacks of the technique based on semantically interacting events is that it generates testing suites composed of a very large number of test cases and this can limit its usefulness.

Semantically interacting event sequences are generated up to a maximum length k_{max}. Since there is an upper bound to the number of test cases that can reasonably compose a test suite, k_{max} is in practice quite small. In the case study documented by Marchetto *et al.* [8] more than 5000 test cases of length $k \leq 4$ are generated from the semantically interacting event sequences. On the other hand, the experiments performed by Marchetto *et al.* and by Yaun *et al.* show that the capability to reveal faults tends to grow at increasing event sequence length k.

In this paper, we investigate the use of a search-based approach (based on the hill-climbing algorithm) to address the problem of generating long semantically interacting event sequences while keeping the test suite size reason-

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.13

ably small. In order to preserve a fault revealing power comparable to that of the exhaustive test suite, we maximize the diversity of the test cases. We re-formulate the problem as an optimization problem that can be solved by applying a heuristic search algorithm guided by an objective function. Specifically, we introduce a measure of test case diversity and instead of generating exhaustively all test cases up to a given length k_{max}, we select only the most diverse test cases, without any constraint on their length k. A case study has been conducted on two real Ajax applications. The results indicate that test suites consisting of long interaction sequences generated by means of the proposed search-based approach are the most effective.

The paper is organized as follows: Sections 2 and 3 provide some background respectively on Ajax and Ajax testing (summarizing the approach by Marchetto *et al.* [8]). Our search-based approach to test case generation is presented in Section 4. Experimental results are discussed in Section 5. Related works (Section 6) and conclusions (Section 7) terminate the paper.

2 Ajax

Ajax (Asynchronous Javascript And XML) is a bundle of technologies used to simplify the implementation of rich and dynamic Web applications. It employs HTML and CSS for information presentation. The Document Object Model [1] is used to access and modify the displayed information. The *XMLHttpRequest* object is exploited to retrieve data from the Web server asynchronously. XML is used to wrap data and Javascript code is executed upon callback activation.

With Ajax, developers can implement asynchronous communications between client and server. To this aim, the Ajax object *XMLHttpRequest* wraps any service request or data traveling between client and server. The asynchronous response from the server is handled at the client side by Javascript code which is part of a callback triggered by the response. Upon arrival of the server message, the handler method associated with such an event is run. Depending on the received data, the message handler can change the structure or content of the current Web page through the DOM.

Since Ajax Web applications are heavily based on asynchronous messages and DOM manipulation, we expect the faults associated with these two features to be relatively more common and widespread than in other kinds of applications. So, first of all, Ajax testing should be directed toward revealing faults related to incorrect manipulation of the DOM. For example, this may be due to assumptions about the DOM structure which become invalid during the execution, because of page manipulation by Javascript code. Another example is an inconsistency between code and DOM, which makes the code reference an incorrect or nonexistent part of the DOM. Second, Ajax testing should deal with asynchronous message passing, a well-known source of trouble in software development [5]. Often Ajax programmers make the assumption that each server response comes immediately after the request, with nothing occurring in-between. While this is a reasonable assumption under good network performance, when the network performance degrades, we may occasionally observe unintended interleaving of server messages, swapped callbacks, and executions occurring under incorrect DOM state. All such faults are hard to reveal and require dedicated techniques [7].

3 Ajax Testing

An Ajax Web application is constructed around the structure of the DOM to be manipulated by the message handlers associated with user events or server messages. Correspondingly, we model an Ajax application by means of a Finite State Machine (FSM), the states of which represent DOM instances and the transitions of which represent the effects of callback executions. This model captures precisely the two distinctive features of Ajax, namely reflective DOM manipulation and asynchronous messages. Given this model of an Ajax application, test cases can be derived according to available techniques for state-based testing [11], such as path coverage criteria. However, such approaches tend to generate a high number of test cases involving unrelated events. Hence, we restricted the considered event sequences to those containing chains of semantically interacting events [8], similarly to the technique proposed for GUI-testing [12, 13]. To make the paper self contained, we now summarize how to obtain the FSM model and how to extract semantic event interactions which are turned into test cases.

3.1 Model extraction

We extract the FSM of an Ajax application through dynamic analysis, complemented by information coming from static code analysis. Dynamic analysis is by definition partial, hence a manual validation or refinement step may be required after model extraction, to ensure that the extracted model is not under-approximating the set of admissible behaviors.

The starting point for our dynamic analysis is a set of execution traces, such as the ones shown in Figure 1 for a hypothetical *Cart* application written in Ajax. *Cart* allows users to add and remove items from a cart, or to empty the cart. Execution traces may be obtained from log files generated by real user interactions, following an approach similar to the one proposed by Elbaum *et al.* [4]. We can also take

Trace	Event sequence
1	add
2	rem
3	add, add, rem
4	add, add, rem, rem, rem
5	add, empty
6	add, empty, rem

Figure 1. Traces for Cart (events only)

advantage of existing test cases, if any, produced by previous testing phases or in previous development iterations.

DOM element	Abstraction
`DIV | SPAN | P`	*null* | *empty*
`TEXTAREA`	*null* | *empty* | *notEmpty*
`OL | UL`	*null* | `#LI=0` | `#LI>0`
`TABLE`	*null* | `(#TD | #TR)=0` | `(#TD | #TR)>0`
`INPUT type=text`	*null* | *empty* | *notEmpty*
`A`	*null* | *notNull*
`LI`	*null* | *empty* | *notEmpty*
`SELECT`	*null* | *empty* | *sel=1* | ...
`INPUT type=text name=total`	*null* | *total=0* | *total>0*

Figure 2. Fragment of the default state abstraction function (top) and Cart-specific abstraction (bottom)

Traces contain information about the DOM states and the callbacks causing transitions from state to state (Figure 1 shows only the callbacks for space reasons). DOM states are abstracted from the concrete states by means of a state abstraction function, such as the one shown in Figure 2. Since the number of possible concrete DOM states is usually huge and unbounded, we do not represent them directly in the FSM model. We consider abstract states only instead, following an approach similar to the one implemented in the tool Adabu [3]. The state abstraction function shown in Figure 2 (top) is a default, generic function that can be used as a first approximation with any Ajax application. However, it is often necessary to refine or extend it with application-specific abstraction mechanisms. Figure 2 (bottom) contains one such example, where the text field *total* is known to actually contain a number, hence its abstraction can be defined upon the numeric value, being either 0 or greater than zero. On the contrary, the default abstraction of a text field is *null*, *empty*, or *notEmpty*. In *Cart*, *total* is always *notEmpty*, hence the default abstraction is not adequate. With regards to transitions, we represent those user events that have an effect on the DOM. All other method invocations have no effect on the DOM state, thus we can safely ignore them. We determine the set of methods reacting to events and possibly affecting the DOM by means of a static code analysis. The output of this analysis determines the set (actually, a safe superset) of the methods that need to be traced. The DOM state is logged after the invocation of each such methods.

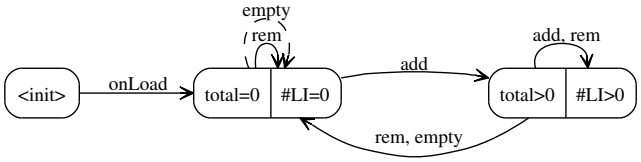


Figure 3. FSM for Cart application

Figure 3 shows the FSM obtained from the traces in Figure 1 by means of the state abstraction function in Figure 2. One admissible transition (dashed in Figure 3) is missing, the reason being that no execution trace gathered in this example covers it (dynamic analysis is partial), hence it was added manually to the model. Furthermore, the built FSM can be manually enriched by the user with conditions constraining transitions activation.

3.2 Semantic interactions -based Testing

Definition 1 (Semantically interacting events) *Events e_1 and e_2 interact semantically if there exists a state S_0 such that their execution in S_0 does not commute, i.e., the following conditions hold:*

$$S_0 \Rightarrow_{e_1;e_2} S_1$$
$$S_0 \Rightarrow_{e_2;e_1} S_2$$
$$S_1 \neq S_2$$

where S_0 is any state in the FSM of the Ajax application. We have a semantic interaction whenever the effects on the DOM state of the two callbacks c_1 and c_2, associated with e_1 and e_2, are not independent, i.e., swapping the order of execution brings the application to a different state. This definition can be easily extended to sequences of events, instead of single events, by just replacing e_1 and e_2 with two event sequences. In the *Cart* example, the two events *rem, add* interact semantically. In fact, the execution order $\langle$*rem, add*$\rangle$ results in a cart containing one item (state `#LI>0`). The sequence $\langle$*add, rem*$\rangle$ produces an empty cart (state `#LI=0`).

Two different criteria [8, 12, 13] can be used to generate suites of test cases based on semantically interacting events.

The first testing criterion (called *SEM*) is based on the concatenation of pairs of semantically interacting events. In the sequence $\langle e_1, \ldots, e_n \rangle$ obtained by applying this criterion, every pair of events e_i, e_{i+1} in the sequence (with i ranging between 1 and $n-1$) is a pair of semantically interacting events.

The second criterion (called *ALT*) considers the semantic interaction of the event sequence prefix with the last event in the sequence. In the sequence $\langle e_1, \ldots, e_n \rangle$ obtained by applying this criterion, the prefix $\langle e_1, \ldots, e_{n-1} \rangle$ interacts semantically (i.e., does not commute) with e_n.

Once event sequences are generated according to either of the criteria above, input values must be provided. To this aim, we take advantage of a database of input values, collected together with the traces that are used for model extraction. Values in the database are typed, so that it is possible to randomly pick-up one of the values available in the database for the particular data type required by a given callback. Each obtained event sequence is converted into a testing script that can be executed by Selenium[1]. The PASS/FAIL result of a test case execution is determined by: (1) checking the consistency of the concrete state sequence w.r.t. the related state sequence in the FSM model of the application; (2) checking any output value (e.g., DOM elements, their attributes and the textual content in the resulting page) against a manually provided oracle; (3) determining whether the application is crashed by the test case.

4 Hill-Climbing Testing

We propose a search-based test case derivation technique for Ajax (called *HILL*), based on the hill climbing algorithm. It uses an objective function (also called fitness) to evolve an initial population of test cases (a test suite) with the aim of producing eventually a test suite which optimizes the objective function (i.e., maximizes the fitness). Since hill climbing is a heuristic method, the final solution will be in general a local, not the global, optimum. Even if not globally optimal, the solution found through hill climbing may be a good approximation for the problem considered.

Hill climbing is a local search algorithm that, given an initial solution (e.g., pairs of semantically iteracting events), uses the fitness function to evaluate the neighborhood solutions. A solution is a neighborhood solution if it can be reached from the current one by applying small changes to it. In our case, a neighborhood solution is produced by concatenating a semantically interacting event at the end of an existing test case. Among the neighboring solutions which improve the fitness, the one with highest fitness improvement is selected and used to form the next solution. Then, the algorithm iterates, evaluating neighborhood solutions and selecting new ones, until no improving neighborhood can be found.

```
(st1) Input:
- N_max: max test suite size (default: 100)
- FSM: Finite State Machine model
- k: initial event sequence length (default: 2)
- K_max: max event sequence length (default: 100)
(st2) Output:
- S: optimized test suite
(st3) Initialization of the test suite  S
S = semInteractEventSeqs(FSM, k);
if sizeOf(S) > N_max
 S = randomSample(S, N_max);
end if
(st4) Evolution of the suite from length  k to k+1
repeat
 iniFit = computeFitness(S);
 S_best = S;
 for each tc in S such that length(tc) == k
  m = getLastEvent(tc);
  for each n in getNextSemInteractEvents(FSM, m);
   tc' = tc + n;
   if sizeOf(S) < N_max
    S' = addTestCase(S, tc');
   else
    S' = replaceTestCase(S, tc, tc');
   end if
   if computeFitness(S') > computeFitness(S_best)
    S_best = S';
   end if
  end for
 end for
 S = S_best;
until iniFit == computeFitness(S)
(st5) Repeat (st4) with k = k + 1 until:
- no fitness improvement occurs:
  computeFitness(S) after (st4)
  gives the same value for k and for k + 1
- K_max iterations have been performed
  (k == K_max)
```

Figure 4. Hill-Climbing based test suite generation

Figure 4 shows the pseudo-code of the algorithm used by HILL to generate the sequences of semantically interacting events composing the final testing suite. The test suite S being generated contains initially short sequences of events obtained by concatenating pairs of semantically interacting events. By default, the length k of such initial sequences is 2, but any arbitrary value can be used. In some cases, the number of sequences may exceed the maximum test suite size N_{max}. In such cases, sequences are sampled up to the size N_{max}.

Then, the main iteration of the hill climbing algorithm is entered (step *st4*). The purpose of this iteration is extending event sequences from length k to length $k+1$, hence improving the test suite fitness. When no improvement can be

[1] http://www.openqa.org/selenium

achieved by means of extensions to length $k+1$, step *st4* terminates and the algorithm continues with step *st5*. In this step, k is incremented, in order to consider longer event sequences. With this new value of k, step *st4* is re-executed if the following conditions hold: (1) the maximum sequence length K_{max} has not been reached yet; and (2) the previous execution of step *st4* has successfully performed some fitness improvement. The algorithm terminates when the maximum length is reached or the fitness does not improve any more.

Let us now consider step *st4* in more detail. Each test case of length k is extended to length $k+1$ by concatenating at the end of it an event that interacts semantically (according to the *SEM* definition) with the last event in the test case. In this way a new test case of length $k+1$ is formed. It can be either added to the existing suite or it can replace the test case selected for extension, depending on the size of S, compared to N_{max}. The fitness of the resulting test suite (S') is then evaluated. If it is greater than the best extension considered so far, it is recorded as the current best improvement, into S_{best}. After examining all test cases and all possible extensions for each test case, the best possible improvement is made by replacing S with S_{best}. It should be noticed that S and S_{best} differ by exactly one test case: the one which, once extended, produces the highest fitness increase. The surrounding repeat-until loop (instructions 14-32) takes care of ensuring that all fitness improvement actions that apply to test cases of length k are actually made. After at most N_{max} iterations the loop terminates and step *st4* is over: all fitness improving extensions from length k to $k+1$ have been made. As described above, step *st5* is in charge of repeating this process at increasing event sequence length.

4.1 Fitness function

As the objective function, we propose three fitnesses based on the notion of test diversity [9]: *EDiv*, test suite diversity based on the execution frequency of each event that labels a transition in the FSM exercised by the test cases of the suite; *PDiv*, test suite diversity based on the execution frequency of each pair of semantically interacting events labeling FSM contiguous transitions exercised by each test case of the suite; and *TCov*, test suite diversity based on the FSM coverage reached by the test cases in the suite.

Test diversity is a notion related to the test distribution, which can be obtained by analyzing the execution frequency of specific software elements or artifacts (e.g., GUI events and program branches) for sets of test cases. In other words, test diversity is the degree to which a set of test cases executes a given software in diverse ways with respect to each other. HILL computes the diversity degree of a test suite according to the exercised events (*EDiv* and *TCov*) or pairs of semantically interacting events (*PDiv*).

In detail, given a test suite S, composed of a set of test cases based on semantically interacting sequences of events, its events-based diversity (*EDiv*) is computed as follows:

$$EDiv_{min}(S) = \sum_{tc \in TCS} \min_{tc' \neq tc} \sqrt{\frac{\sum_{e \in Events} (F_e^{tc} - F_e^{tc'})^2}{|Events|}}$$

$$EDiv_{avg}(S) = \sum_{tc \in TCS} \sqrt{\frac{\sum_{e \in Events} (F_e^{tc} - \overline{F_e})^2}{|Events|}}$$

where: TCS are the test cases of S; $Events$ are the FSM events that are exercised by the test cases; F_e^{tc} and $F_e^{tc'}$ are the execution frequencies of the event e in test cases tc and tc', while $\overline{F_e}$ is the average frequency of event e computed over the entire test suite S.

$EDiv_{min}$ measures how a testing suite is diverse by considering the minimun distance between the event frequencies of each test case and the event frequencies of the other test cases in the same suite. $EDiv_{avg}$ measures how a testing suite is diverse by considering the distance from the average frequencies.

For instance, in the *Cart* example, we can consider the three events add, rem, and $empty$, exercised by two test suites, $S1$ composed of two test cases $tc1 = \langle rem, add, rem \rangle$ and $tc2 = \langle rem, add, empty \rangle$, and $S2$ composed of two other test cases $tc3 = \langle rem, add, rem \rangle$ and $tc4 = \langle rem, add, rem \rangle$. The execution frequency of the events exercised by these test cases are: add=1, rem=2, and $empty$=0 for $tc1, tc3$ and $tc4$; while for $tc2$ add=1, rem=1, and $empty$=1. By considering these execution frequencies, the events-based diversities are $EDiv_{min}(S1) = 1.63$ and $EDiv_{min}(S2) = 0$; $EDiv_{avg}(S1) = 0.81$ and $EDiv_{avg}(S2) = 0$. Therefore, in this example the test cases composing the suite $S1$ are more diverse (i.e., more spread across the events of the FSM) than the test cases composing $S2$.

Similarly to the $EDiv$ measure, the diversity based on pairs of semantically interacting events ($PDiv$) can be measured as follows:

$$PDiv_{min}(S) = \sum_{tc \in TCS} \min_{tc' \neq tc} \sqrt{\frac{\sum_{(n,m) \in P} (F_{(n,m)}^{tc} - F_{(n,m)}^{tc'})^2}{|P|}}$$

$$PDiv_{avg}(S) = \sum_{tc \in TCS} \sqrt{\frac{\sum_{(n,m) \in P} (F_{(n,m)}^{tc} - \overline{F_{(n,m)}})^2}{|P|}}$$

where: TCS is the set of test cases of the suite, P is the set of pairs of events that interact semantically, $F^{tc}_{(n,m)}$ and $F^{tc'}_{(n,m)}$ are the execution frequencies of the pair (n, m) respectively in tc and tc'; $\overline{F_{(n,m)}}$ is the average execution frequency for the pair (n, m).

The third fitness function, *TCov*, is based on event coverage. It selects as the most adequate set of test cases those with the highest coverage of the FSM transitions. Note that this coverage measure is a special case of the events-based diversity, in which the frequency counter for each event is just a boolean value indicating if an event has been covered (exercised) by a given test case or not.

$$TCov(S) = |\bigcup_{tc \in TCS} CoveredTransitions^{tc}|$$

where: TCS is the set of test cases of the suite and $CoveredTransitions^{tc}$ the set of transitions of the FSM covered by the current test case tc.

5 Case Study

We conducted a case study experiment using two AJAX Web applications: Tudu[2] and Oryx[3]. Tudu is an application supporting management of personal todo lists. By using Tudu, the user can create, remove or change her/his todo lists and events, and can share them with other users. The application consists of more than 10k lines of codes written in Java/JSP and it uses a huge set of frameworks (e.g., Struts, Spring, Oro, Aspectj, Log4j, Velocity, Xalan, DWR). Oryx is an editor for modeling business processes in BPMN (Business Process Modeling Notation). It is distributed in three versions: demo, client and developers and, in the experiment, we just considered the client one. The application consists of around 200k lines of codes, written in Javascript/Java/JSP and it uses some libraries (Batik, HtmlParser, Xerces, Xalan).

We developed three tools to support the proposed testing technique: (1) *FSMInstrumentor*, a Javascript module able to trace the execution of the Web application under test; (2) *FSMExtractor*, a Java module to analyze the execution traces and build the application FSM; and (3) *FSMTestCaseGenerator*, a Java module to analyze the built FSM and generate test suites according to SEM, ALT, and $HILL$ (using five fitness functions: $EDiv_{min}$, $EDiv_{avg}$, $PDiv_{min}$, $PDiv_{avg}$ and $TCov$).

[2] http://tudu.sourceforge.net

[3] http://bpt.hpi.uni-potsdam.de/Oryx

5.1 Research questions and Used Metrics

The aim of the experiment is to investigate adequacy and effectiveness of $HILL$ in generating test suites, compared to those obtained by applying the SEM/ALT criteria. In detail, the experiment goal is to answer to the following research questions:

RQ1: *What is the test case distribution over sequence length k produced by HILL, with respect to SEM/ALT?*

RQ2: *What is the effectiveness in finding faults of HILL compared to SEM/ALT?*

RQ1 deals with the ability of each testing criterion to define test suites of a "reasonable" size and composed of test cases of different lengths k, including big values of k. A carefully generated test suite composed of a limited number of test cases of different lengths is potentially more adequate than a huge suite composed of test cases of small length. RQ1 can be answered by computing the composition of the testing suites in terms of length of their test cases (i.e., the number of test cases per length k). This helps in understanding how the different testing criteria explore the space of all test cases that can be generated for a given set of lengths k. We will evaluate the distribution of the generated test cases over their length k by plotting and visually inspecting the distribution plots for the test suites produced by $HILL$ and by SEM/ALT.

RQ2 deals with the effectiveness in finding faults. It can be answered by computing the number of faults revealed by each testing suite ($HILL$ vs. SEM/ALT) and comparing it with the number of faults that are known to be in the applications (e.g., injected faults). The effectiveness can be also evaluated by computing the fault detection capability ratio $rFDC(S)$:

$$rFDC(S) = \frac{\sum_{f \in F} \frac{|FR^f(S)|}{|TCS|}}{|F|}$$

where: $FR^f(S)$ is the set of test cases of the suite S that reveal fault f; F is the set of all (known) faults; and TCS is the set of test cases in the current suite of S.

5.2 Procedure

The following steps have been performed to execute the experiment for both AJAX applications:

1. we instrumented the source code of the target application by using *FSMInstrumentor*;

Id	Fault Description
Tudu	
1	The function for adding a todo duplicates the todo element
2	*Delete completed Todos* doesn't delete todos
3	Delete completed Todos delete also the last todo edited (if any)
4	*Advanced Add todos* doesn't work when the number of todos is greater than 3
5	*Advanced Add todos* doesn't work when the number of todos is greater than 1
6	*Delete todo* doesn't work when Delete completed Todos is clicked before add
7	*Delete current list* works only if add to do list is clicked two times
8	*Delete completed Todos* doesn't work if after the click, *quickDelete* is clicked in 5 seconds
9	*Delete todo* doesn't work correctly but, when it is selected, it performs the *edit todo* function
10	The *edit list* action works correctly but, during its execution, it deletes the same list
11	*Edit list* has an unexpected delay before to do its task. This leads to an unexpected behaviour if other operations are executed during that while
12	The todo ordering operation works correctly but if some todos *actionCheckbox* are checked, these todos are deleted
13	*Advanced Add todos* works correctly but, if a note is filled, an additional item is added to the list
14	The *Share list* add also an empty item to the todos list (when the *Hide* button is clicked)
15	*Advanced Add todos* works correctly but if a note is filled for a todo then its DOM-representation doesn't contain the *ActionDelete* button
Oryx	
1	The save function does not work correctly and reliably, when a new element is added to an already saved workflow it is not save correctly
2	The main canvas is not deactivated when a model export window is shown
3	The save function stores wrong information and properties about the saved workflow
4	The workflow stencil set does not work
5	The PNML-export function does not work, it is not correctly activated when the canvas is not empty
6	The exported file is not correctly generated, it is corrupted
7	The BPMN syntax-check does not work correctly, some kinds of element are not considered
8	Missing attributes: in the BPMN stencil set, some properties and assignments are missing
9	The export function is not action-independent
10	When export an existing workflow, its relative URLs to the stencil-sets are not correctly defined and built
11	Handle terminate end events and exception throwing in PNML-export function
12	Handling the task-flow in PNML-export function and syntax check

Table 1. Faults injected into Tudu and Oryx

2. we executed the instrumented application for a long execution time, with the aim of exercising at least each AJAX event of the application GUI. The execution has been traced in log files by *FSMInstrumentor*;

3. the recorded traces have been analyzed by *FSMExtractor* to extract the application FSM;

4. the built FSM has been analyzed by *FSMTestCaseGenerator* to extract suites of test cases based on SEM, ALT, and $HILL$ (considering all the proposed fitness functions);

5. we generated some mutated versions of the target application by injecting several faults (see below);

6. we executed each generated test suite with the aim of revealing the injected faults.

Injected Faults

Confidence in the results of the experimental study can be achieved only if the faults generated artificially resemble the real faults. In order to seed both Tudu and Oryx with faults as similar as possible to real faults, we retrieved a set of real failure descriptions from the bug tracking systems[4] of such applications and we manually analyzed them. Whenever possible, we reproduced such failures by seeding faults that we regarded as reasonable causes for such failures.

The result of this process is that 15 (state-based) failures have been injected into Tudu and 12 into Oryx. Some of them are briefly described in Table 1. For instance, the fault number 2 of Tudu introduces an error in the functionality *Delete completed Todos*, which prevents invocation of the Java function that actually deletes the completed todos from the database. This fault is a state-based fault since, after the injection, the click of the "Delete completed Todos" button does not change the initial state of the application (i.e., selected todos are not deleted). An interesting example of fault injected into Oryx is number 4. The injection of this fault implies that when the user selects the BPMN stencil set, some design elements (e.g., specific kinds of task) are not correctly loaded in the list provided to the user by the tool.

[4] (Tudu) http://sourceforge.net/tracker/?group_id=131842&atid=722407 (Oryx) http://code.google.com/p/oryx-editor/issues/list

Criteria	Test cases length k											Total
	2	3	4	5	6	7	8	9	10	11	12	
Tudu												
SEM	73	385	2309	-	-	-	-	-	-	-	-	2767
ALT	92	71	63	61	46	6	-	-	-	-	-	339
$HILL_{EDiv_{min}}$	5	7	16	24	10	19	7	5	2	4	1	100
	17	57	72	36	18	0	0	0	0	0	0	200
$HILL_{EDiv_{avg}}$	2	11	32	32	19	4	0	0	0	0	0	100
	8	13	14	17	28	44	18	28	13	11	6	200
$HILL_{PDiv_{min}}$	19	42	17	6	4	1	4	4	0	2	1	100
	39	90	38	9	5	2	4	8	15	11	17	200
$HILL_{PDiv_{avg}}$	11	14	18	15	9	5	3	3	9	11	2	100
	4	16	26	39	37	13	14	8	15	11	17	200
$HILL_{TCov}$	12	16	14	25	7	16	9	0	0	1	0	100
	25	49	16	35	11	35	20	6	1	2	0	200
Oryx												
SEM	93	193	670	2081	-	-	-	-	-	-	-	3037
ALT	103	98	87	56	9	2	-	-	-	-	-	355
$HILL_{EDiv_{min}}$	2	8	21	24	14	10	11	6	4	0	0	100
	6	26	31	34	46	29	20	4	2	2	0	200
$HILL_{EDiv_{avg}}$	1	1	10	21	22	22	15	4	1	3	0	100
	56	36	44	16	14	16	9	7	1	1	0	200
$HILL_{PDiv_{min}}$	11	40	35	9	2	0	0	2	0	1	0	100
	104	66	18	3	2	0	7	0	0	0	0	200
$HILL_{PDiv_{avg}}$	28	18	25	6	10	8	5	0	0	0	0	100
	53	39	46	20	17	21	4	0	0	0	0	200
$HILL_{TCov}$	10	43	38	7	2	0	0	0	0	0	0	100
	107	70	14	9	0	0	0	0	0	0	0	200

Table 2. Test suite size

5.3 Results and Discussion

FSMExtractor analyzed the 28 and 17 execution traces used to exercise Tudu and Oryx respectively and available for producing their FSMs. The built FSMs contain 14 and 15 states and 37 and 39 transitions, respectively for Tudu and Oryx. No customization of the abstraction function was necessary for analyzing Tudu and build its FSM. A customization was needed for analyzing Oryx to model a `DIV` HTML tag filled with one or more element of the same type `DIV`. The `DIV` element is abstracted in terms of: "$null|empty(\#DIV = 0)|!empty(\#DIV > 0)$". Since the size of the initial suites considered in both applications is less than N_{max} the applied hill climbing algorithm is deterministic.

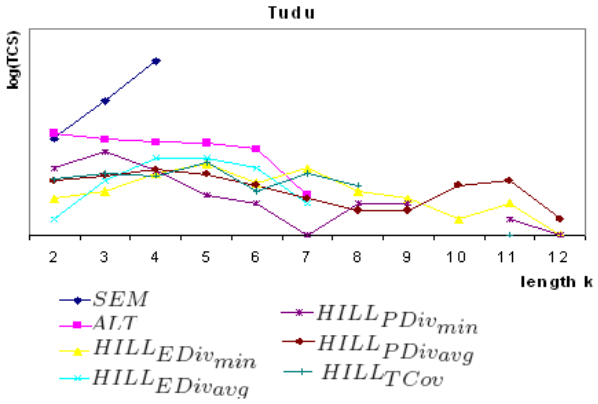


Figure 5. Test cases per k

RQ1: Testing suite size

As shown in Table 2, the test suite size for the $HILL$ criteria has been fixed (100, 200 test cases), so that a high size reduction is obtained for the $HILL$ test suites with respect to SEM (3%, 6%) and a moderate size reduction is obtained with respect to ALT (28.5%, 57%).

Table 2 shows also the number of test cases per testing criterion, divided according to their sequence length k. Figure 5 shows the plot obtained for Tudu, for the $HILL$ criteria. Only the results obtained for 100 test cases are plotted in the figure. They are plotted in a logarithmic scale to make them more readable. The test cases more evenly distributed across sequence lengths k are those generated

Criteria	Revealed Fault	rFDC (%)
Tudu		
SEM	10 (66.6%)	1.5
ALT	2 (13.3%)	0.2
$HILL_{EDiv_{min}}$	10 (66.6%)	3
	9 (60%)	1.9
$HILL_{EDiv_{avg}}$	6 (40%)	3.3
	6 (40%)	1.8
$HILL_{PDiv_{min}}$	5 (33.3%)	1.6
	8 (53.3%)	1.9
$HILL_{PDiv_{avg}}$	8 (53.3%)	2.4
	8 (53.3%)	2.2
$HILL_{TCov}$	9 (60%)	2.1
	9 (60%)	1.8
Oryx		
SEM	11 (91%)	1.7
ALT	7 (58.3%)	2.1
$HILL_{EDiv_{min}}$	5 (41.6%)	2.7
	9 (75%)	1.6
$HILL_{EDiv_{avg}}$	9 (75%)	3.1
	9 (75%)	1.8
$HILL_{PDiv_{min}}$	6 (50%)	2.5
	8 (66.6%)	1.9
$HILL_{PDiv_{avg}}$	6 (50%)	2.4
	8 (66.6%)	1.6
$HILL_{TCov}$	6 (50%)	3
	8 (66.6%)	2

Table 3. Revealed faults

by $HILL_{EDiv_{min}}$, $HILL_{PDiv_{min}}$ and $HILL_{PDiv_{avg}}$ for Tudu; $HILL_{EDiv_{min}}$ and $HILL_{EDiv_{avg}}$ for Oryx. The other criteria generate suites of test cases that are more concentrated in a specific subset of the considered *k*. For instance, for Tudu, $HILL_{EDiv_{avg}}$ generates test cases from length $k = 2$ to 7. Sometimes a smaller test suite size limit has the effect of removing shorter sequences early. This forces the selection of alternative (sub-optimal) sequences eventually resulting in longer test cases (e.g., $HILL_{EDiv_{min}}$ for Tudu 100 vs 200).

The use of the hill climbing algorithm seems to be effective in increasing the length of the test cases of the generated suite, while keeping the test suite size small. In our case studies, $HILL_{EDiv_{min}}$ produced better distributed test cases across lengths.

RQ2: Faults revealed

Table 3 shows the faults revealed by each testing suite. The number of faults revealed by $HILL_{EDiv_{min}}(100)$ is the same as those revealed by SEM for Tudu (66%). A comparable, but slightly worse performance (60%) is exhibited on Tudu by $HILL_{EDiv_{min}}(200)$ and $HILL_{TCov}$. On Oryx, all search based methods perform worse than SEM, with $HILL_{EDiv_{min}}(200)$ and $HILL_{EDiv_{avg}}$ getting quite close to it (75% faults revealed by $HILL$ vs. 91% faults revealed by SEM). On the other hand, the fault detection ratio (see column $rFDC$) is generally improved by search based methods, i.e., search based methods produce test suites containing a higher ratio of fault revealing test cases. This is especially true when the test suite size is 100. For example, $HILL_{EDiv_{min}}(100)$ produces suites with 3% of test cases revealing each fault (on average) for Tudu and 2.7% for Oryx, compared to 1.5% and 1.75% produced by *SEM* in the two respective cases. The fault detection capability is approximately doubled. On this metrics, $HILL_{EDiv_{avg}}(100)$ performs even better (3.3% and 3.1% respectively).

Overall Considerations

According to the overall result of the experiment, we conclude that $HILL_{EDiv}$ (both *min* and *avg* versions) is the most effective and adequate testing technique for generating test suites based on semantic sequences of events. Using $HILL_{EDiv_{min}}$ the generated test suites achieve a comparable (actually, slightly lower) number of revealed faults, with respect to *SEM*, with a substantial reduction of the test suite size (3% and 6% of the *SEM* suite size for the 100 or 200 test suites, respectively). $HILL_{EDiv_{avg}}$ is the one obtaining very high fault detection ratio in Table 3 for both AJAX applications. The reduced test suite size comes with an increased fault detection ratio, which makes it easier for the tester to reveal a fault early during the testing process.

Our experiment confirms the intuitions [8, 13] that: (a) length *k* and distribution over *k* of the event sequences taken into account for generating test cases deeply affect their fault revealing capability; and, (b) increasing the suite diversity and the number of long interaction sequences is a suitable strategy to improve the effectiveness of the automatically generated test suites.

Threats to validity

We considered only two applications, so our results cannot be easily generalized to arbitrary Ajax Web applications. However, the tested applications (Tudu and Oryx) are quite typical Ajax applications, in terms of their size, technology and the implemented functionalities. This means that the considered bugs, the recovered FSMs and the generated test suites have probably features that can be found in other similar applications. As happening with any experimental study in software engineering, it is only by replication that we can build a sound body of knowledge on this topic.

6 Related works

Several techniques and a few tools have been presented in the literature to support testing of Web applications.

Functional testing tools of Web applications (e.g., LogiTest, Maxq, Badboy, Selenium) are based on capture/replay facilities: they record the interactions that a user has with the graphical interface and repeat them during regression testing. Another approach to functional testing is based on HttpUnit. HttpUnit is a Java API providing all the building blocks necessary to emulate the behavior of a browser. When combined with a framework such as JUnit, HttpUnit permits testers to write test cases that verify the expected behavior of a Web application.

Model-based testing of Web applications was proposed by Ricca and Tonella [10]. Coverage criteria (e.g. page and link coverage) are defined with reference to the navigational model, i.e. a model containing Web pages, links and forms. Another proposal of model-based testing of Web applications was made by Andrews et al. [2]. In this case, the navigational model is a finite state machine with constraints recovered by hand by the test engineers directly from the Web application.

Testing of Web applications employing new technologies (e.g., Ajax, Flash, ActiveX plug-in components, Struts, Ruby on rails, etc.) is an area that has not been investigated thoroughly so far. A first step in this direction has been made in the Selenium TestRunner tool [6], which supports the "waitForTextPresent" condition, necessary whenever asynchronous messages received by the client determine the next execution step of a test case.

The work presented in this paper is based on the state-based testing approach, originally defined for object-oriented programs [11]. The approach was recently applied to GUI [13] and Ajax [8] testing. In fact, similarly to Ajax applications, GUI code is event driven and processing depends on callback execution. Hence, we share with the GUI testing techniques notions such as event sequences and semantically interacting events. However, Ajax applications have specific features (asynchronous communications and DOM manipulation) which make them different from GUI applications.

7 Conclusions and future work

We have proposed an improvement of the original state-based testing technique designed to address the features of Ajax applications. In the proposed technique, the DOM manipulated by Ajax code is abstracted into an FSM and sequences of semantically interacting events are extended by a hill climbing algorithm to generate testing suites of reasonably small size, comprising long interaction sequences and maximizing a specific objective function, focused on the test suite diversity.

The experiment that we have conducted confirms the intuition that longer interaction sequences have higher fault exposing potential. By selecting long sequences that maximize the test suite diversity, it is possible to keep their number small, with a limited degradation in number of revealed faults and a substantial increase of the fault detection ratio.

Our future work will be devoted to the improvement of the FSM recovery step, in order to automatically infer proper abstraction functions. We will experiment with alternative search based algorithms and we will apply them to a larger benchmark of Ajax applications. We will also investigate the role of input selection and infeasible paths in the FSM during test case generation.

References

[1] Document Object Model (DOM). *http://www.w3.org/DOM*.

[2] A. Andrews, J. Offutt, and R. Alexander. Testing Web Applications by Modeling with FSMs. *Software and System Modeling, Vol 4, n. 3*, July 2005.

[3] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Proc. of the Fourth International Workshop on Dynamic Analysis (WODA)*, Shangai, China, May 2006.

[4] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher. Leveraging user session data to support web application testing. *IEEE Transactions of Software Engineering*, 31(3):187–202, March 2005.

[5] K. Gatlin. Trials and tribulations of debugging concurrency. *ACM Queue*, 2(7), October 2004.

[6] J. Larson. Testing ajax applications with selenium. InfoQ magazine, 2006.

[7] A. Marchetto, P. Tonella, and F. Ricca. A case study-based comparison of web testing techniques applied to ajax web applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(6):477–492, December 2008.

[8] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *Proc. of IEEE International Conference on Software Testing (ICST)*, Lillehammer, Norway, April 2008.

[9] B. Nikolik. Test diversity. *Information and Software Technology*, 48:1083–1094, April 2006.

[10] F. Ricca and P. Tonella. Analysis and testing of Web applications. In *Proc. of ICSE 2001, International Conference on Software Engineering, Toronto, Ontario, Canada, May 12-19*, pages 25–34, 2001.

[11] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. *IEEE Conference on Software Maintenance (ICSM)*, September 1993.

[12] X. Yuan and A. Memon. Alternating gui test generation and execution. In *Proc. of IEEE Testing: Academic and Industrial Conference (TAIC PART)*, Washington, DC, USA, 2008.

[13] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 396–405, Washington, DC, USA, May 23–25, 2007. IEEE Computer Society.

An Improved Meta-Heuristic Search for Constrained Interaction Testing

Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer
Department of Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, NE 68588-0115
{*bgarvin, myra, dwyer*}*@cse.unl.edu*

Abstract

Combinatorial interaction testing (CIT) is a cost-effective sampling technique for discovering interaction faults in highly configurable systems. Recent work with greedy CIT algorithms efficiently supports constraints on the features that can coexist in a configuration. But when testing a single system configuration is expensive, greedy techniques perform worse than meta-heuristic algorithms because they produce larger samples. Unfortunately, current meta-heuristic algorithms are inefficient when constraints are present.

We investigate the sources of inefficiency, focusing on simulated annealing, a well-studied meta-heuristic algorithm. From our findings we propose changes to improve performance, including a reorganized search space based on the CIT problem structure. Our empirical evaluation demonstrates that the optimizations reduce run-time by three orders of magnitude and yield smaller samples. Moreover, on real problems the new version compares favorably with greedy algorithms.

1. Introduction

Software development is shifting to producing families of related products [2, 19]. Developing these families as integrated, highly-configurable systems offers significant opportunities for cost reduction through systematic component reuse. Unfortunately, highly configurable systems are even more difficult to validate than traditional software of comparable scale and complexity; faults may lie in the interactions between features. These *interaction faults* only appear when the corresponding features are combined, and it is generally impractical to validate every feature combination as that means testing all possible configurations [5, 6, 24].

Combinatorial interaction testing (CIT) tempers the cost of validation by limiting the degree of feature interaction considered. Only a sample of the set of possible configurations is tested, but that sample contains at least one occurrence of every t-way interaction, where t is called the strength of testing [4].

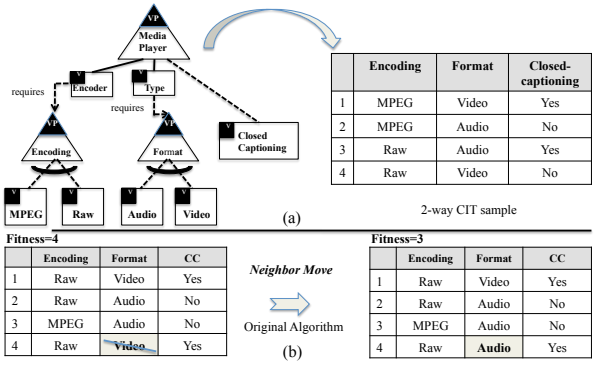
CIT sampling is dominated by approximating algorithms of two varieties: greedy techniques such as the Automatic Efficient Test Case Generator (AETG) [4] or the In Parameter Order (IPO[G]) algorithm [14, 23] and meta-heuristic algorithms like simulated annealing [8, 21, 22]. Both greedy and meta-heuristic approaches construct solutions by making small, elementary decisions, but a greedy algorithm's selections are permanent, whereas meta-heuristic search may revisit its choices. Intuitively, greedy techniques should be faster because each decision occurs just once; for the same reason, meta-heuristic strategies should discover better answers when the consequences of selections are difficult to anticipate. In CIT this impression is accurate: greedy algorithms tend to run faster, but meta-heuristic searches usually yield smaller sample sizes [8]. Thus, the former are better when building and testing a configuration is inexpensive, but the latter become superior as these costs increase.

However, these observations ignore an inherent problem: highly-configurable systems typically have feature constraints—restrictions on the features that can coexist in a valid configuration. While earlier work efficiently handled constraints in a greedy algorithm [7], they remain a roadblock to meta-heuristic search [6]. Hence, if testing is expensive and feature constraints are present, neither approach is cost-effective.

In this paper we study the sources of inefficiency for the only published meta-heuristic algorithm for constrained CIT—a variation on simulated annealing [6] described in Section 3. In Section 4 we identify eight algorithmic improvements, two of which show significant promise: (1) modifying the global strategy for selecting a sample size and (2) changing the neighbor-

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.25

hood of the search. Then, in Section 5, we evaluate the benefits of these improvements on five real highly-configurable systems and 30 synthesized systems that share their characteristics. Our results show that simulated annealing can be cost-effective for constrained CIT problems: the changes reduce sample generation time by three orders of magnitude. Perhaps more importantly, when each configuration takes non-trivial time to test (for example, more than 30 seconds), the results suggest that simulated annealing's better sample sizes more than compensate for its longer run time, compared to a greedy algorithm. This makes it the superior choice for CIT on real systems where executing a test suite typically takes tens or hundreds of minutes [16].



	Encoding	Format	Closed-captioning
1	MPEG	Video	Yes
2	MPEG	Audio	No
3	Raw	Audio	Yes
4	Raw	Video	No

	Encoding	Format	CC
1	Raw	Video	Yes
2	Raw	Audio	No
3	MPEG	Audio	No
4	Raw	~~Video~~	Yes

	Encoding	Format	CC
1	Raw	Video	Yes
2	Raw	Audio	No
3	MPEG	Audio	No
4	Raw	**Audio**	Yes

Figure 1. Example of SPL and Neighbors

2. Background

We begin by introducing a small example of a software product line (SPL). An SPL is one type of configurable system, a family of related products defined through a well managed set of commonalities and points of variability [2, 20]. SPLs are used for development in many large companies and present unique challenges for validation [5, 17]. Our example is an SPL for a media player and appears on the left side of Figure 1(a) in the Orthogonal Variability Modeling language (OVM) [20]. In OVM variation points are shown as triangles, and rectangles represent each variant (or feature). Required variants are connected by solids lines, optional features use dashed lines, and arcs indicate alternative choices. Additionally, dashed arrows labeled "requires" indicate hierarchical relationships between variants and variation points. Because the modeling language is hierarchical, the selections of leaf features uniquely determine the final products.

There are two variation points of interest in this family, *encoding* and *format*, and a single optional variant, *closed-captioning*. The *encoding* is either MPEG or RAW, and *format* is AUDIO or VIDEO. Because *closed-captioning* is optional we represent its inclusion or exclusions with YES or NO. Hence, there are a total of eight possible products in this SPL.

If we plan to test the *family* for interaction faults then each of the eight products must be constructed and tested individually. Although this is reasonable in a small example, most realistic product lines have thousands or millions of possible products because the size of the family grows exponentially with the number of variation points. For instance, an unconstrained SPL with 10 variation points, each having 3 variants, describes 3^{10} or 59,049 distinct products.

Interaction faults in software usually depend on a small number of interacting features [13]; in CIT we sample to cover these interactions. Thereby we save testing effort: in the SPL with 10 variation points we only need 14 products to exercise all 2-way interactions, and in the example media player SPL we need just the 4 on the right side of Figure 1(a). A CIT sample is usually encoded as a covering array, which we define next.

2.1. Constrained Covering Arrays

A *covering array* (CA) is an array having N rows and k columns (factors). The elements in the i^{th} column are chosen from a set of v_i symbols (values). We adopt the convention that columns do not share symbols. The key property of a CA is that for any choice of t columns, all possible sets of t symbols (t-sets) appear in at least one row; the t-sets are said to be covered.

In CIT, a covering array's rows describe a sample of the configuration space; columns represent variation points, so each symbol in a row indicates the alternative chosen at the corresponding point. For example, $k = 3$ in the media player SPL, and each v is size 2. To test all pairwise interactions we would set $t = 2$ and obtain the sample of four products in Figure 1(a).

But there are also constraints such as hardware or software incompatibilities or developer/market preferences. Returning to the media player example, picking YES for *closed-captioning* implies the choice of VIDEO. The third configuration in our CIT sample (Figure 1(a)) violates this constraint—it is an invalid product.

The presence of such constraints in CIT was for a long time a major impediment to the sampling process [4, 6]. However, recent work [6] has shown that we can incorporate constraint information into the greedy sampling algorithms without increasing computational cost. This work led to the formalization of *constrained covering arrays* (CCAs) in [7].

In CCAs we define a Boolean variable for each symbol that could appear in the covering array. Given

a row, a variable is true if the corresponding symbol appears in that row, false otherwise. We then encode constraints on the rows as a propositional formula and restrict the covering array to rows that satisfy this formula. For instance, the formula VIDEO $\vee \neg$YES would capture the constraint where closed captioning implies video support. Once we've added constraints, fewer t-sets need to be covered, but the space of valid covering arrays has a more complicated structure.

2.2. Simulated Annealing

The objective in CIT is to find a CCA that minimizes N. Unfortunately, a tight bound is not known for general CAs, let alone CCAs [11], so approaches such as greedy algorithms and meta-heuristic search that approximate minimality are most prevalent. We concentrate on simulated annealing because it has produced small arrays for unconstrained problems [8, 22].

Under simulated annealing the search space comprises all $N \times k$ arrays populated with the values for each factor. The fitness function for the CIT problem is the number of uncovered t-sets in the sample. For instance, in Figure 1(b) on the left, we see an intermediate state of the algorithm when N is 4 (ignoring constraints for now). Of the twelve 2-sets that must be covered we are missing [MPEG, YES], [MPEG, VIDEO] [AUDIO, YES] and [VIDEO,NO]; the fitness is 4. A smaller fitness means that we are closer to the optimum, zero.

The algorithm starts with an initial, random array (the start state) and then makes a series of state transitions. Each transition is a random mutation of a single location in the solution. This mutation defines the search neighborhood. In Figure 1(b) a transition to a new solution is illustrated. The cost of the new solution, 3, is better.

The simulated annealing algorithm accepts a new solution if its fitness is the same or closer to optimal than the current one. When this is not the case, it uses a pre-defined temperature and cooling schedule to set the probability for making the bad move at $e^{f(S)-f(S')/T}$, where T is the current temperature, S and S' are the old and new solution, and $f(S)$ represents the fitness of solution S. As the algorithm proceeds, the temperature is cooled by a decrement value (the number of iterations between decrements is a parameter of the algorithm). As the temperature cools the probability drops and we are less likely to allow a bad move.

3. Base Algorithm

The original application of simulated annealing to covering array problems is described by Stevens in [22]. Though that algorithm requires each column to have the same number of symbols, Stardom [21] extended it to remove this restriction. In [8] we added optimizations and other features, and [6] introduced support for constraints. We used a satisfiability (SAT) solver to determine whether or not rows of the covering array satisfy the constraints at each move. Our work continues from this last version; we refer to it as the *base algorithm*.

Compared with our implementation of the AETG greedy algorithm, mAETG, the base algorithm met with limited success in [6]. For small 2-way problems it found constrained CIT samples that matched the quality of mAETG, but failed to compete at higher strengths both in size and time; the search had little information about constraints so it spent too many iterations undoing transitions to infeasible arrays. Moreover, though both programs took longer to solve constrained problems, the base algorithm's run time grew at a much faster rate.

The following subsections first explain the base algorithm's overall approach and then discuss its two primary components: an outer search and an inner search. Along the way we highlight some drawbacks that warrant remedy.

3.1. Design

Because the minimum covering array size cannot be known ahead of time, the base algorithm repeatedly calls simulated annealing, each time with a different N. Hence, there are two layers to the design. The *outer search* chooses values for N and either accepts or rejects them according to the results of an *inner search*, simulated annealing proper.

3.2. Outer Search

The outer search is shown in Figure 2. It takes an upper and lower bound on the size of the covering array and performs a binary search within this range.

There are two points of interest. First, at each size simulated annealing attempts to build an array (line 4), and the return value is the last array found, whether it is a solution or not, so its coverage must be checked (line 5). Second, the outer search is responsible for returning the smallest covering array constructed, so it must keep a copy of the best solution (line 6).

Though this approach works, binary search is a poor fit for this problem because its fundamental assumptions are violated. Our first two criticisms of the base algorithm are:

1. The binary search supposes that the inner search accurately evaluates whether an array of some size can be built. However, the inner search is stochas-

binarySearch$(t,k,v,C,lower,upper)$

```
1  let A ← ∅;
2  let N ← ⌊(lower + upper)/2⌋;
3  while upper ≥ lower do
4  |   let A' ← anneal(t,k,v,C,N);
5  |   if countNoncoverage(A') = 0 then
6  |   |   let A ← A';
7  |   |   let upper ← N − 1;
8  |   else
9  |   |   let lower ← N + 1;
10 |   end
11 |   let N ← ⌊(lower + upper)/2⌋;
12 end
13 return A;
```

Figure 2. The Base Outer Search, a Binary Search

tic and might terminate before finding a solution that does in fact exist.

2. The binary search expects the minimum size to lie within the given bounds. But the quantities *lower* and *upper* are merely estimates and may not bound the final answer, especially in constrained problems.

3.3. Inner Search

Figure 3 shows the specifics of the inner search. At each step, the code randomly chooses a location in the array and a symbol to put there (lines 4–6). If the replacement causes the row to violate constraints, it discards that move and tries again (line 9), following an unsophisticated policy called the death penalty [3]. Consequently, the search never enters an infeasible region. But if constraints are satisfied, it computes the change in coverage (line 10). When the alteration increases or maintains coverage, the algorithm applies it and continues (lines 11–13), but it only accepts bad moves with a probability that depends on the quality of the move and the current temperature (lines 14–16). The iterations continue until a stabilization criterion is met (line 3); that is, the algorithm has found a solution or seems not to be making further progress. Then the array is returned (line 21).

Unlike a typical implementation of simulated annealing, only the first start state is entirely random. Instead, because we are repeatedly invoking the inner search for nearly the same problem we use a single large array for every search and ignore the rows at index N and higher.

We point out four criticisms of the inner search and add them to our list:

3. Constraints decrease the connectivity of the state space, so the average path to a solution is longer in constrained problems. Thus the search needs many more iterations for the same probability of success.
4. In the extreme case, constraints disconnect the search space, rendering the problem unsolvable from some start states.
5. The initial state generator keeps no record of the symbols it has tried while creating random rows. Therefore, it may waste time making choices already known to violate constraints.
6. Although progress is saved from one run of simulated annealing to the next, the choice of rows to use is arbitrary. Difficult-to-obtain rows may be lost while redundant rows are preserved.

anneal(t,k,v,C,N)

```
1  let A ← initialState(t,k,v,C,N);
2  let temperature ← initialTemperature;
3  until stabilized(countNoncoverage(A)) do
4  |   choose row from 1...N;
5  |   choose column from 1...k;
6  |   choose symbol from v_column;
7  |   let A' ← A;
8  |   let A'_{row,column} ← symbol;
9  |   if SAT(C, A'_{row,1...k}) then
10 |   |   let δ ← countCoverage(A') − countCoverage(A);
11 |   |   if δ ≥ 0 then
12 |   |   |   let A ← A';
13 |   |   else
14 |   |   |   with probability e^{δ/temperature} do
15 |   |   |   |   let A ← A';
16 |   |   |   end
17 |   |   end
18 |   |   let temperature ← cool(temperature);
19 |   end
20 end
21 return A;
```

Figure 3. The Base Inner Search, Element-wise Simulated Annealing

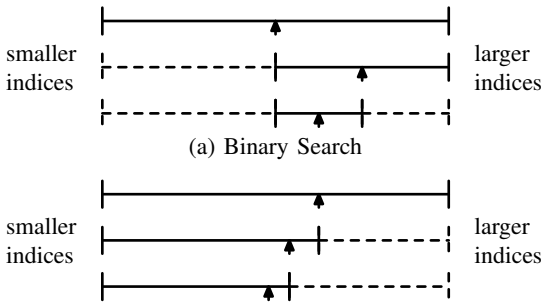
4. Modifications

For each shortcoming we detail a solution. Our ideas are organized into two categories: the two most influential changes appear as major modifications; the rest are listed under minor modifications.

4.1. Major Modifications

We discovered that our most significant alterations stemmed from Criticisms 1 and 3. We introduce one-sided narrowing, a meta-heuristic, to accommodate inaccuracy from the inner search and t-set replacement, a state space restructuring, to mitigate constraints' impact on state space traversal.

One-Sided Narrowing. In the previous section we pointed out that the binary search assumes simulated



(a) Binary Search

(b) One-Sided Narrowing

Figure 4. One-Sided Narrowing Chooses Partitions so that Larger Sizes are Eliminated

```
outermostSearch(t, k, v, C, lower, upper)
 1  let A ← ∅;
 2  while upper ≥ lower do
 3      let A′ ← binarySearch(t, k, v, C, lower, upper);
 4      if A′ = ∅ then
 5          break;
 6      else
 7          let A ← A′;
 8          let upper ← rows(A′) − 1;
 9      end
10  end
11  return A;
```

Figure 5. The New Outer Search, One-Sided Narrowing

annealing can decide whether a covering array is constructable (Criticism 1). This assumption does not hold because simulated annealing cannot conclusively show the nonexistence of a solution at a given size. Constraints make the problem harder, so the inner search is even more likely to return a false negative. Therefore, one-sided narrowing keeps the essence of a binary search, but abandons this faulty assumption.

The idea behind binary search is still valid: we want to restrict the range of candidate sizes as much as possible. Rather than narrowing from both sides though, the algorithm should only improve the upper bound because soundness is guaranteed. So at each step the code must either find a covering array of size $N \in [lower \ldots upper]$ and refine its search or give up.

The difference is illustrated by Figure 4. A solid range represents a subspace being considered; a dashed range has been eliminated. Arrowheads indicate the points of partition. In an ordinary binary search, part (a), the algorithm begins with a partition and then decides which half to discard. With one-sided narrowing, part (b), the program decides that the upper half will be eliminated, then it looks for a satisfactory partition.

Naïvely, the algorithm could always choose $N = upper$ as the partition, but our experience discourages a linear search. With rows being reused from one attempt to the next, larger fluctuations in size help knock the inner search out of local optima. Moreover, it is wasteful to decrease $upper$ by just one row when larger cuts might be possible.

Instead, we recognize that though the binary search cannot serve as the entire outer search, it does accomplish a single step: it returns an $N \in [lower \ldots upper]$ at which a covering array can be constructed or determines that it cannot find such an N. Therefore, one-sided narrowing uses binary search to locate each partition.

We now have a three-layer search. The new, outermost search shown in Figure 5 invokes the old outer search from Figure 2, which calls simulated annealing.

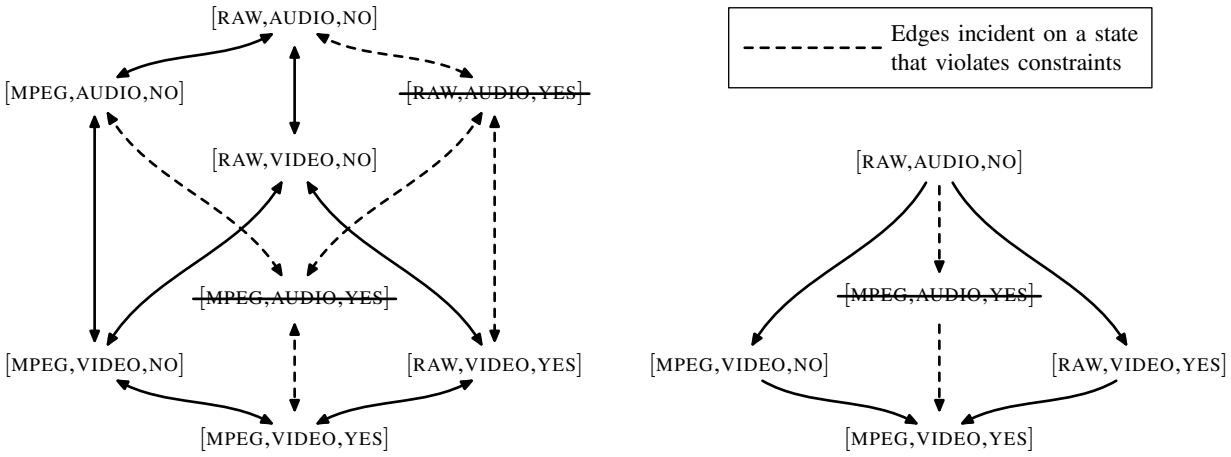
At first glance this seems to be an expensive proposition—finding the first partition means executing the entire base algorithm! Nevertheless, we expect to recover the cost by needing fewer iterations each time the inner search is called.

***t*-set Replacement.** Each iteration of simulated annealing attempts a single state transition; if we represent the state space as a graph, simulated annealing is tracing a path through the graph until it runs out of iterations or reaches a solution vertex. States that violate constraints form obstacles to search progress, leading to the problem in Criticism 3.

For example, suppose that we are testing 2-way interactions in the media player presented earlier. To make the search space small enough to illustrate, we will assume that every pair except [MPEG, VIDEO], [MPEG, YES], and [VIDEO, YES] is covered in the first rows of the covering array and that simulated annealing will only change the last row. Trivially, this row should be [MPEG, VIDEO, YES], so we will put its initial state as far away as possible, at [RAW, AUDIO, NO]. This gives the state space in Figure 6(a). Solid lines depict legal transitions; dashed lines indicate transitions that violate constraints. Note that every state also has three self-loops, not shown for the sake of readability.

Looking at just the upper right part of the graph, the path from [RAW, AUDIO, NO] through [RAW, AUDIO, YES] to [RAW, VIDEO, YES] was valid without constraints. But if simulated annealing attempts this route with constraints, it fails on the first transition and must instead detour via [RAW, VIDEO, NO].

To lessen the effect we allow simulated annealing to take short excursions through the infeasible states such as [RAW, AUDIO, YES]. Instead of checking feasibility after every transition, we have the search apply several transitions, then check. Or, equivalently, we treat this group of transitions as a single step. In the example, we combine pairs of transitions so that [RAW,



(a) Original Space

(b) Revised Space

Figure 6. The Original and Revised Search Space

VIDEO, YES] becomes directly reachable from [RAW, AUDIO, NO], much like what is shown in Figure 6(b).

We must be careful though: if we group too many transitions together adjacent states will be unrelated, and simulated annealing will degenerate to random search. To balance concerns about constraints with consideration for the search space we set the number of old moves per new move at t. There is particular advantage to choosing t. Rather than allowing all sets of t transitions, we can further inform the search by only choosing groups that substitute in a yet-to-be-covered t-set. We show this in Figure 6(b). For instance, [RAW, AUDIO, NO] becomes [RAW, VIDEO, YES] when the 2-set [VIDEO, YES] is added; the subsequent addition of [MPEG, VIDEO] or [MPEG, YES] leads to a solution.

We observe two clear advantages. First, in this new space, simulated annealing has many options when it is far from a solution, but the alternatives dwindle as it nears an answer. For example, in Figure 6(b), [RAW, AUDIO, NO] has three outgoing edges, [MPEG, VIDEO, YES] has one, and [MPEG, VIDEO, YES] has none. Being confined in a solution's vicinity, the algorithm should make a more thorough search there than in the rest of the space. Second, a t-set that by itself violates constraints will never be one that needs to be covered, so simulated annealing avoids these inherently infeasible transitions.

4.2. Minor Modifications

We developed several other changes, each motivated by a criticism from Section 3. These modifications were less effective, but we briefly list them for completeness. Criticisms 1, 2, 4, 5, and 6 are addressed by (1) dynamically bounding the inner search's iteration, (2) choosing binary search partitions more intelligently, (3) checking if the lower or upper bounds are attained, (4) replacing whole rows to escape highly constrained states, (5) adding a simple SAT history to the initial state generator, and (6) sorting the rows after each run of simulated annealing.

Along with these alterations we changed the underlying SAT solver from zChaff [15] in the base algorithm to MiniSAT [10] for comparability with the greedy implementation mAETG.

5. Evaluation

Our evaluation is organized as follows: We present two research questions and establish the scope of investigation. We then describe experiments aimed to answer each research question. Results and analysis follow. The evaluation artifacts are available for download at `http://cse.unl.edu/~bgarvin/research/ssbse2009/`.

5.1. Research Questions

Our first objective is to determine which changes are beneficial in the targeted context—constrained problems. This leads to the research question:

RQ1 (Effect of Modifications). How does each change affect performance on constrained problems?

Using the set of modifications that yield the best performance, we pose a second research question:

RQ2 (Comparison to mAETG). How does the best algorithm from RQ1 compare to a greedy approach, the modified AETG algorithm from [7]?

5.2. Scope

Conducting a full factorial experimentation to quantify the effects of each proposed change would be too costly. Instead, we group the modifications to study their effects. All of the minor modifications are grouped as a single change and we investigate the major changes—t-set replacement and one-sided narrowing—independently. In total there are four variations to the base algorithm: with minor changes (Minor), with minor changes and t-set replacement (TSR), with minor changes and one-sided narrowing (OSN), and with all of the changes (All).

Along with the minor modifications we refactored the implementation to more cleanly support our changes. When adding the minor changes, we kept the simulated annealing parameters from the base algorithm, but these settings were updated when major modifications were introduced; the specific parameter settings are available on the website given earlier.

5.3. Experiments

All of our data are obtained by executing simulated annealing once per problem unless otherwise indicated. The computations were performed on a 2.4GHz Opteron 250 running Linux. We record the final array size and total run time.

Effect of Modifications. In the first experiment we let t be two and compare all five versions on the 35 problems in [7]. Five of these problems are taken from real highly-configurable systems, and the other thirty are based on the characteristics of these five. More detail on these systems is given in [7].

Our dependent variables are the cost in run time and size of the generated sample. With just this raw data, it is not clear whether a small sample found at great expense is better than a large sample discovered quickly. But CIT is ultimately meant to reduce the total time spent selecting and testing configurations; this is the metric we should use.

This total cost is modeled as $c_{gen} + c_{config} \times N$, where c_{gen} is the cost to generate a sample and c_{config} the cost to test a configuration. Clearly, lower values of c_{gen} compensate for larger N. On the other hand, if c_{config} is expensive then expending more time to generate a smaller sample would make sense. Following convention, when comparing two CIT methods we call the point where $c_{gen} + c_{config} \times N = c'_{gen} + c_{config} \times N'$ the *break-even point*.

For this first experiment we use the sums over all the benchmarks to find break-even points. Our discussion of performance is based on these points.

Comparison to mAETG. The second experiment ran the best performing implementation 50 times over the 35 benchmarks. In this way, the averages are directly comparable with the mAETG figures reported in [7]. Our dependent variables are the array size, run time, and break-even point calculated from the mAETG and simulated annealing run times and array sizes.

5.4. Results and Analysis

Throughout the presentation of results we adopt an abbreviated notation for constrained covering arrays. A model term written as $x_1^{y_1} x_2^{y_2} \cdots x_n^{y_n}$ indicates that there are y_i columns with x_i symbols to choose from for each i. A constraint term, written in the same format, means that y_i constraints involve x_i symbols.

Effect of Modifications. The results for the first experiment are listed in Table 1. After the benchmark names and brief descriptions, the table gives the sample sizes and run times for each algorithm. The smallest values are shown in bold. Two samples for the base algorithm did not complete after 27 days (2.3 million seconds); in those cases we list N/A and append + to the sum.

It's clear that t-set replacement dramatically reduces run time. On the other hand, the effect of one-sided narrowing varies with the search space. In the old search space (columns Minor and OSN) it worsens array sizes, but combined with t-set replacement (between columns TSR and All), it always does at least as well. Put together, the All version runs more than a thousand times faster than the base algorithm and produces smaller covering arrays.

When we compute break-even points, only the version TSR ever outperforms All. So there is only one break-even point of interest: variation TSR is best when the per-configuration time is less than 39.79 seconds, and after that the All version dominates. Because few test suites for configurable systems run in under 40 seconds [16], we conclude that including both major modifications is suitable for the common case.

Comparison to mAETG. Table 2 gives the averages for simulated annealing, including both changes, and mAETG over 50 runs. For every row we choose the mAETG variant with the best break-even point compared to simulated annealing, breaking the one tie (on benchmark 22) by array size.

The contrast is sharp. Simulated annealing gives better array sizes in all but one case, averaging 11.16 fewer rows; over the set of 35 samples we only need 75% as many configurations to achieve the same CIT goals. If we had chosen different mAETG variants for each problem that produce the smallest sizes rather than the best break-even point, the average difference would

Table 1. Sizes and Times for Constrained Two-Way Problems

			Size					Run Time (s)				
Name	Model	Constraints	Base	Minor	TSR	OSN	All	Base	Minor	TSR	OSN	All
SPIN-S	$2^{13}4^5$	2^{13}	24	21	20	26	**19**	1513.8	1827.2	**1.7**	140.1	32.2
SPIN-V	$2^{42}3^24^{11}$	$2^{47}3^2$	36	33	39	**32**	36	22278.2	22206.0	**18.5**	122355.0	122.2
GCC	$2^{189}3^{10}$	$2^{37}3^3$	21	**19**	25	**19**	**19**	49856.7	50981.0	1250.4	61013.0	**850.6**
Apache	$2^{158}3^84^45^16^1$	$2^33^14^25^1$	33	**31**	36	47	34	47465.2	31638.4	**60.6**	12686.9	137.8
Bugzilla	$2^{49}3^14^2$	2^43^1	17	**16**	18	**16**	17	972.6	123.1	**4.1**	2081.4	4.4
1.	$2^{86}3^34^15^56^2$	$2^{20}3^34^1$	44	40	40	63	**38**	41055.5	9154.3	**31.6**	7047.5	74.3
2.	$2^{86}3^34^35^16^1$	$2^{19}3^3$	32	**30**	33	44	32	80020.5	6185.9	24.1	6650.4	**9.2**
3.	$2^{27}4^2$	2^93^1	**18**	**18**	**18**	**18**	**18**	284.4	3718.5	4.0	327.2	**2.8**
4.	$2^{51}3^44^25^1$	$2^{15}3^2$	21	**20**	22	23	21	16968.6	1096.8	6.2	2725.7	**5.8**
5.	$2^{155}3^74^35^56^4$	$2^{32}3^64^1$	60	48	65	81	**47**	69107.0	69003.4	**77.0**	62702.1	337.0
6.	$2^{73}4^36^1$	$2^{26}3^4$	**24**	**24**	**24**	**24**	**24**	62706.3	1777.6	**7.7**	16454.9	29.4
7.	$2^{29}3^1$	$2^{13}3^2$	**9**	**9**	**9**	**9**	**9**	121.9	2902.2	**1.5**	192.2	2.1
8.	$2^{109}3^24^25^36^3$	$2^{32}3^44^1$	49	43	45	66	**39**	243499.0	11621.1	**30.5**	19858.2	105.9
9.	$2^{57}3^14^15^16^1$	$2^{30}3^7$	N/A	30	30	**20**	**20**	N/A	10.8	**3.3**	2103.3	25.6
10.	$2^{130}3^64^55^26^4$	$2^{40}3^7$	58	44	48	78	**43**	257061.2	40577.3	**60.4**	9575.0	258.8
11.	$2^{84}3^44^25^26^4$	$2^{28}3^4$	46	43	47	43	**41**	170596.4	63153.2	**25.0**	143728.6	139.7
12.	$2^{136}3^44^35^16^3$	$2^{23}3^4$	49	41	43	62	**40**	359336.2	14544.8	**66.6**	16873.7	116.3
13.	$2^{124}3^44^15^26^2$	$2^{22}3^4$	40	**36**	**36**	54	**36**	304404.0	12641.5	**31.5**	6237.2	56.1
14.	$2^{81}3^54^36^3$	$2^{13}3^2$	37	37	41	**36**	37	81464.6	63353.5	**19.6**	159571.8	104.0
15.	$2^{50}3^44^15^26^1$	$2^{20}3^2$	N/A	**30**	**30**	**30**	**30**	N/A	6467.3	**9.6**	61320.4	20.0
16.	$2^{81}3^34^26^1$	$2^{30}3^4$	**19**	24	25	24	24	1001.9	2208.4	**15.2**	27995.8	60.9
17.	$2^{128}3^34^25^16^3$	$2^{25}3^4$	42	**39**	41	64	**39**	47396.6	36955.7	**51.7**	10406.1	69.7
18.	$2^{127}3^24^45^66^2$	$2^{23}3^44^1$	177	44	45	72	**43**	15515.0	18433.3	**61.2**	2511.9	72.7
19.	$2^{172}3^94^95^36^4$	$2^{38}3^5$	180	49	51	103	**47**	132833.5	68558.4	**118.3**	8436.6	1550.7
20.	$2^{138}3^44^55^46^7$	$2^{42}3^6$	67	56	57	105	**53**	340518.9	32677.3	**86.4**	25030.8	915.3
21.	$2^{76}3^34^25^16^3$	$2^{40}3^6$	37	36	37	53	**36**	81000.3	16736.3	**20.3**	6816.0	37.6
22.	$2^{72}3^44^16^2$	**36**	36	36	37	40	**36**	64098.0	2699.5	12.5	2882.2	**12.4**
23.	$2^{25}3^16^1$	$2^{13}3^2$	13	18	18	**12**	13	48.0	0.7	**0.5**	906.0	5.0
24.	$2^{110}3^25^36^4$	$2^{25}3^4$	51	**44**	47	71	**44**	241610.7	113832.0	**32.8**	11674.2	80.2
25.	$2^{118}3^64^25^26^6$	$2^{23}3^34^1$	59	**49**	54	80	**49**	26043.5	143208.0	**45.2**	23952.7	285.1
26.	$2^{87}3^14^35^4$	$2^{28}3^4$	**30**	**30**	33	48	31	151137.6	43067.7	**23.1**	5821.4	25.4
27.	$2^{55}3^24^25^16^2$	$2^{17}3^3$	**36**	**36**	**36**	47	**36**	5105.4	1634.9	**5.7**	2402.5	7.8
28.	$2^{167}3^{16}4^25^36^6$	$2^{31}3^6$	164	51	55	84	**50**	255115.0	115759.9	**126.3**	71001.6	787.3
29.	$2^{134}3^75^3$	$2^{19}3^3$	31	29	32	40	**27**	36434.7	17289.0	**38.7**	9752.7	283.8
30.	$2^{73}3^34^3$	$2^{20}3^2$	20	**17**	20	20	19	33245.9	32270.0	**16.1**	9378.8	36.7
Sum			1580+	1171	1257	1654	**1147**	7905417.1+	1058315.0	**2387.5**	932614.0	6696.9

only decline to 11.06, still a 25% cut. However, the overall cost for running our algorithm on these samples is more than six times that of mAETG.

In the last column we show the break-even points between mAETG and simulated annealing as the minimum time that each configuration must take before simulated annealing outperforms mAETG. Most of these times are quite small, less than 30 seconds. There are two notable exceptions: the break-even point for the GCC optimizer is just short of eight minutes, and our algorithm never does better on benchmark 22. As demonstrated by [16] though, eight minutes is not much time to test a configuration, so we conclude that the new simulated annealing is competitive in every case but one.

Further Analysis. To confirm that we have still retained the quality of the original simulated annealing algorithm on unconstrained problems we selected 29 benchmarks from a variety of publications. For each we ran version All one time and compared this with the best size reported for simulated annealing. The aggregate figures are promising; our implementation's total array size is only 0.2% higher than the best figures.

5.5. Threats to Validity

We classify our threats as affecting external, internal, or construct validity.

External Validity. The major external threat is our choice of benchmarks. We cannot guarantee that either the real CIT problems or the synthetic inputs accurately represent configurable software. Though diverse, they are all derived from open-source command-line or web-based software.

Internal Validity. There are four main threats to internal validity. First, in RQ1 we only use one run of stochastic algorithms. Our continuing experimentation suggests that this threat is minor; we see consistency as in RQ2 where the standard deviation in size is no more than three rows on any benchmark and less than a row

Table 2. Average Size and Times Over 50 Runs for Constrained Two-Way Problems

Name	Size mAETG Best [7]	Size All	Run Time (s) mAETG Best [7]	Run Time (s) All	Break-even (s/cfg.)
SPIN-S	27.0	**19.4**	**0.2**	8.6	1.11
SPIN-V	42.5	**36.8**	**11.3**	102.1	15.93
GCC	24.7	**21.1**	**204.0**	1902.0	471.67
Apache	42.6	**32.3**	**76.4**	109.1	3.17
Bugzilla	21.8	**16.2**	**1.9**	9.1	1.29
1.	53.3	**38.6**	**24.4**	179.5	10.55
2.	40.5	**31.0**	**14.7**	25.4	1.13
3.	20.8	**18.0**	**0.2**	3.4	1.14
4.	28.6	**21.0**	**3.1**	29.3	3.45
5.	63.8	**47.7**	**134.8**	655.9	32.37
6.	34.0	**24.2**	**7.2**	18.2	1.12
7.	12.5	**9.0**	**0.3**	2.1	0.51
8.	55.6	**40.5**	**45.1**	249.9	13.56
9.	26.0	**20.0**	**3.0**	29.8	4.47
10.	60.4	**44.0**	**74.3**	357.3	17.26
11.	58.3	**41.9**	**23.8**	240.7	13.23
12.	54.5	**40.4**	**68.0**	221.1	10.86
13.	48.6	**36.5**	**45.5**	60.5	1.24
14.	51.8	**36.9**	**20.2**	58.1	2.54
15.	40.4	**30.7**	**4.9**	19.3	1.48
16.	33.4	**24.1**	**9.7**	19.7	1.08
17.	53.4	**39.2**	**46.9**	335.5	20.32
18.	57.3	**42.3**	**60.1**	303.5	16.23
19.	64.7	**47.8**	**168.4**	823.6	38.77
20.	71.5	**53.2**	**121.1**	1133.3	55.31
21.	51.7	**36.3**	**14.3**	46.2	2.07
22.	**26.2**	36.0	**6.7**	12.3	N/A
23.	15.7	**12.6**	**0.6**	10.1	3.06
24.	58.3	**42.8**	**40.6**	304.5	17.03
25.	65.7	**48.3**	**61.7**	507.8	25.64
26.	42.0	**30.7**	**15.5**	71.2	4.93
27.	45.8	**36.0**	**4.6**	10.8	0.63
28.	68.5	**50.7**	**170.4**	1522.3	75.95
29.	38.4	**29.7**	**38.9**	89.3	5.79
30.	45.8	**19.3**	**10.3**	30.5	0.76
Sum	1546.1	**1155.4**	**1533.1**	9501.8	20.40

on average. Second, it may be that we bias the results by making poor parameter choices for some algorithms. For instance, perhaps we should have chosen parameters for the versions with major modifications on an individual basis, rather than as a group. Third, the refactoring that accompanies the minor changes may have affected the run time; however, this threat only affects comparisons to the base version, not the effects of the major changes. Fourth, although we have verified the results of every run, we cannot be completely confident that the implementations are correct translations from pseudocode, nor that they are bug-free.

Construct Validity. We have considered both run time and the size of the output, but we have ignored other metrics that are important in specific scenarios. For example, if we do not expect to finish the interaction testing, diversity of t-sets in the early rows is desirable.

6. Related Work

There has been a large body of work on constructing unconstrained interaction test samples [4, 8, 14, 21–23]. Two primary algorithmic techniques are greedy [4, 9, 14, 23] and meta-heuristic search. The latter includes implementations of simulated annealing, tabu search and genetic algorithms [8, 18, 21, 22].

The work of D. Cohen et al. [4] describes the problem of constrained CIT although the primary means to handle this is through a re-modeling of the CIT parameters. Czerwonka [9] provides a more general constraint handling technique and uses the t-set as the granularity of the search. However, few details of the exact constraint handling techniques are provided. Both of the above algorithms are greedy. Our own work [6, 7] provides a more general solution for greedy construction of constrained CIT samples.

Hnich et al. [12] use a SAT solver to construct CIT samples, but they do not provide a direct way to encode constraints into the problem. Recent work by [1] incorporates constraints directly into their ATGT tool which utilizes a model checker to search for a CIT sample. The only work we are aware of that incorporates constraints into a meta-heuristic CIT algorithm is our own [6] but this solution does not scale.

In this work we focus on one meta-heuristic search algorithm, simulated annealing [8], and reformulate the search to work well on both constrained and unconstrained CIT problems.

7. Conclusions

Testers need CIT tools that perform well in the presence of constraints. Earlier work focused on greedy algorithms, which tend to build larger samples than meta-heuristic searches. But the meta-heuristic search, simulated annealing, did not retain its quality and scaled poorly when finding constrained samples.

In this paper, we adapted a simulated annealing algorithm for constrained CIT. It finds the CIT sample size more efficiently and employs a coarser-grained search neighborhood. Together with some minor modifications these changes provide small sample sizes for a fraction of the original cost, less than one thousandth of the run time.

We have run experiments on four versions of the constrained simulated annealing algorithm and compared the best of these with a greedy implementation on a set of 35 realistic samples. Our experimental results show that we need on average 25% fewer configurations, but the run time for simulated annealing is longer. When we calculate the break-even point based on the

sample construction time and the time to run the test suite on each configuration, we find that if a test suite takes more than 30 seconds to run, simulated annealing typically outperforms the greedy algorithm.

In future work we will plan to add more runs to our experiments, experiment with higher strength CIT samples, and to optimize the simulated annealing parameters for both constrained and unconstrained problems.

8. Acknowledgments

We would like to thank Jiangfan Shi for the use of his mAETG tool and for supplying the CIT models for evaluation of RQ1 and RQ2. Brady Garvin is supported in part by CFDA#84.200A: Graduate Assistance in Areas of National Need (GAANN). This work is supported in part by the National Science Foundation through awards CNS-0454203, CCF-0541263, CNS-0720654, and CCF-0747009, by the National Aeronautics and Space Administration under grant number NNX08AV20A, by the Army Research Office through DURIP award W91NF-04-1-0104, and by the Air Force Office of Scientific Research through award FA9550-09-1-0129. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the position or policy of NSF, NASA, ARO or AFOSR.

References

[1] A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In *Tests and Proofs, Lecture Notes in Computer Science, 4966*, pages 66–83, 2008.

[2] P. Clements and L. Northrup. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[3] C. Coello Coello. Theoretical and numerical constraint handling techniques used with evolutionary algorithms: A survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12):1245–1287, Jan. 2002.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[5] M. B. Cohen, M. B. Dwyer, and J.Shi. Coverage and adequacy in software product line testing. In *Proceedings of the Workshop on the Role of Architecture for Testing and Analysis*, pages 53–63, July 2006.

[6] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *International Symposium on Software Testing and Analysis*, pages 129–139, July 2007.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[8] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 38–48, May 2003.

[9] J. Czerwonka. Pairwise testing in real world. In *Pacific Northwest Software Quality Conference*, pages 419–430, October 2006.

[10] N. Eén and N. Sörrenson. MiniSAT-C v1.14.1. http://minisat.se/, 2007.

[11] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Math*, 284:149 – 156, 2004.

[12] B. Hnich, S. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11:199–219, 2006.

[13] D. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[14] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG: A general strategy for t-way software testing. *Engineering of Computer-Based Systems, IEEE International Conference on the*, pages 549–556, 2007.

[15] S. Malik. zChaff. http://www.princeton.edu/~chaff/zchaff.html, 2004.

[16] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Proceedings of the 26th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2004)*, 2004.

[17] H. Muccini and A. van der Hoek. Towards testing product line architectures. In *Proceedings of the International Workshop on Test and Analysis of Component-Based Systems*, pages 111–121, 2003.

[18] K. J. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics*, 138(1-2):143–152, 2004.

[19] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[20] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering*. Springer, Berlin, 2005.

[21] J. Stardom. Metaheuristics and the search for covering and packing arrays. Master's thesis, Simon Fraser University, 2001.

[22] B. Stevens. *Transversal Covers and Packings*. PhD thesis, University of Toronto, 1998.

[23] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, 2002.

[24] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, Jan 2006.

Evolution and Search Based Metrics to Improve Defects Prediction

Segla Kpodjedo*, Filippo Ricca**,
Giuliano Antoniol* and Philippe Galinier*
{segla.kpodjedo, philippe.galinier}@polymtl.ca,
filippo.ricca@disi.unige.it, antoniol@ieee.org

* SOCCER Lab. – DGIGL, École Polytechnique de Montréal, Québec, Canada
** Unitá CINI at DISI, University of Genoa, Italy.

Abstract

Testing activity is the most widely adopted practice to ensure software quality. Testing effort should be focused on defect prone and critical resources i.e., on resources highly coupled with other entities of the software application.

In this paper, we used search based techniques to define software metrics accounting for the role a class plays in the class diagram and for its evolution over time. We applied Chidamber and Kemerer and the newly defined metrics to Rhino, a Java ECMA script interpreter, to predict version 1.6R5 defect prone classes. Preliminary results show that the new metrics favorably compare with traditional object oriented metrics.

Keywords: Defects Prediction, Software evolution, Error-Correcting Graph Matching (ECGM) algorithm, Class Rank, Evolution Cost.

1 Introduction

Most of the work in the Object-Oriented (OO) unit and integration testing focuses on the important problem of minimizing test effort while assuring the achievement of a testing goal (e.g., a coverage criteria). Indeed, there is a flourishing literature in integration testing where the goal is often to devise optimal test strategies aiming at minimizing the number of drivers and stubs (see for example [3]).

We believe that these approaches do not fully address the problem of how deeply a class should be tested leaving the manager alone in this strategic decision on where should be the testing effort concentrated. For example, let's consider a product manager and/or tester who wants to improve the quality of a large software evolved over time. Once he/she applies approaches such as [3], he/she will obtain an optimal test or retest strategy. Still he/she may need to know what are the key classes to focus on. We believe these are the defect-prone classes (i.e., classes which have the highest risk of producing failures) and classes where a slipped fault could potentially propagate extensively across the system.

What a manager needs is a set of approaches/algorithms able to pinpoint defect-prone and/or critical components. Several researchers are working in this direction and approaches based on object-oriented metrics (e.g., [5, 10, 11]) or historical data (e.g., [17, 18]) have been published. We concur with other researchers (e.g., [7]) that: (i) "the most important classes" and (ii) frequently changed classes are likely to be the most complex and that this complexity leads to defect increase. However, we also believe that the overall role of a class into the system should also be considered when identifying the most important classes.

In this paper we propose two new metrics. These metrics are calculated applying an Error-Correcting Graph Matching (ECGM) algorithm relying on a Google-inspired page ranking heuristic [13] to several versions of the target system, i.e., the system evolution history. More precisely, our algorithm computes, for each class, the *Class Rank* (CR) [19] and the *Evolution Cost* (EC) [13]. CR measures for each class its relative importance in a system, while the EC quantifies how much the class signature and its relations (i.e., aggregations, associations and generalizations) changed in a time frame. In a nutshell, the algorithm locates the "most important" and "more frequently changed" classes.

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.24

We hypothesize [15] that CR and EC can be used (alone or combined with other metrics) to predict defect-prone classes. To test our hypothesis, we conducted a case study using a Java application — the Rhino ECMA script interpreter[1]. More precisely, we ran our meta-heuristic algorithm several times against 10 subsequent versions of Rhino reverse engineered class diagrams up to version 1.6R5. Then, we computed the EC and CR based metrics for each class present in the version 1.6R5. Finally, (i) we built linear regression models to gather empirical evidence of a relationship between our metrics and defects and (ii) we built error proneness prediction models.

The main contribution of this paper can be summarized as follows: (i) we propose a family of search based evolution metrics; (ii) we provide evidence of a relation between the proposed metrics and the number of defects in Rhino version 1.6R5; (iii) we report results obtained with logistic regression models and classification – regression trees;

This paper is organized as follows. Section 2 summarizes our ECGM algorithm, already presented in [13, 14], and presents the metrics used for defect prediction, i.e., CR and EC. Section 3 describes our Rhino case study and presents the obtained experimental results. Section 4 presents related work while Section 5 concludes and outlines future work.

2 ECGM metrics

EC and CR are calculated by applying an Error-Correcting Graph Matching (ECGM) algorithm. In the following subsections we summarize: (i) our ECGM algorithm to make the paper self contained and (ii) how it can be applied to class diagrams to study OO software evolution. Finally, we will define and illustrate our metrics.

2.1 Our ECGM algorithm

In [14], we proposed and applied for the first time a Google-inspired ECGM algorithm [13] to OO software evolution. An ECGM [21] is an elegant framework for studying software evolution as it makes no assumption on problem structure. It can be easily tailored to different classes of problems by simply fitting ECGM cost weights. The ECGM problem is NP-hard [21] and optimal algorithms require prohibitive computation times even for medium sized graphs.

Our ECGM algorithm is a meta-heuristic algorithm that takes as input two graphs g_1, g_2 and produces as output "an optimal or near optimal mapping"[2] between their nodes and edges — with the constraint that a node of g_1 can be matched to at most one node g_2 — and a list of edit operations (the distortions) that transform g_1 into g_2. The parameters that must be fixed in the ECGM algorithm are the weights of each allowed edit operation, i.e. the costs of each distortion.

The edit operations that the ECGM algorithm can use to transform g_1 into g_2, are: *node/edge deletions*, *node/edge insertions* and *node/edge matching errors*. A *node matching error* refers to the dissimilarity between the matched nodes (e.g., the label is different) while an *edge matching error* refers to distortions between the matched edges. Two types of *edge matching errors* have to be considered: replacing a missing edge by an existing edge (*structural error*) or replacing one edge by another (*label error*).

As a result, there are seven possible edit operations and each one is assigned a given cost depending on the problem at hand. In summary, each ECGM cost function can then be parametrized by seven cost values corresponding to the seven edit operations:

- node matching, deletion and insertion: c_{nm}, c_{nd}, c_{ni};
- edge deletion and insertion applied to edges of deleted and added nodes: respectively c_{ed} and c_{ei}; and
- edge matching error: Given two nodes n_1, m_1 of the first graph and two nodes n_2, m_2 from the second, with n_1 matched to n_2 and m_1 matched to m_2, we have an edge matching error if the edge (n_1, m_1) (resp. (m_1, n_1)) is different from the edge (n_2, m_2) (resp. (m_2, n_2)). We have to pay c_{ems} for an edge structural error (one of the edges is missing) or c_{eml} for an edge label error (the edges have different labels).

Often, the cost of adding or deleting a node (or an edge) can be considered identical and thus there is no need to specify two different values c_{nd}, c_{ni} (or c_{ed} and c_{ei}); thus with $c_{no} = c_{nd} = c_{ni}$ and $c_{eo} = c_{ed} = c_{ei}$, five real positive values are sufficient to define a cost function: (c_{nm}, c_{no}, c_{eo}, c_{ems}, c_{eml}).

During execution, our ECGM algorithm chooses "the best move" (i.e., create or undo a match between two nodes from the two different graphs) in the space of all the possible moves aiming at minimizing the ECGM cost function. The choice is done by a Tabu algorithm guided by local node features (i.e., number of incoming/outgoing edges) and global information on

[1] http://www.mozilla.org/rhino/

[2] The algorithm tries to minimize an ECGM cost function.

the nodes. This latter heuristic is implemented via a Google inspired PageRank algorithm [19].

Further details on the ECGM algorithm can be found in [13].

2.2 Modeling class diagram evolution with an ECGM algorithm

To apply ECGM algorithms to study software evolution, we envisage the following steps. First, software artifacts, class diagrams in our case, are represented as graphs. Second, parameters of the ECGM algorithm (the costs of the edit operations) are fixed following an empirical procedure. Finally, when the graphs are available and parameters are fixed, we can build a mapping between the graphs.

Class diagrams can be thought of as labeled graphs with nodes being the classes and edges representing the relations between classes. In this work, labels on edges specify the type of the edge (i.e., association, aggregation or inheritance) while node labels specify the class name plus attributes and methods signatures of the class. In detail, a node label of a class C is a triple (l, SetAs, SetMs): where l is the class name and SetAs, SetMs represent, respectively, the textual representations of attributes and methods signatures. The textual representation of an attribute is computed collapsing attribute type, visibility and name in a string while the textual representation of a method signature is obtained collapsing method name, type of its parameters, and its return type.

To use our ECGM algorithm with this graph representation of a class diagram, we need a way to measure the similarity between two nodes: the internal node similarity. Given two nodes representing two classes and their features represented as sets of strings — C1(l1, SetAs1, SetMs1) and C2(l2, SetAs2, SetMs2) — we compute their similarity as follows.

Similarity between class names is measured using the Levenshtein distance[3], while similarity in the attributes and signatures is computed using the Jaccard index[4] between SetAs1—SetAs2 and SetMs1—SetMs2. The three similarities are then combined in the following formula for having the similarity between two nodes: $intSim(C_1, C_2) = l_w \times (1 - \frac{Levenshtein(l_1,l_2)}{max(length(l_1),length(l_2))}) + m_w \times Jaccard(SetAs1, SetAs2) + a_w \times Jaccard(SetMs1, SetMs2)$

where l_w, m_w and a_w are respectively weights for labels, methods and attributes such as $l_w + m_w + a_w = 1$. As a consequence, $intSim(C_1, C_2)$ is a real between $[0, 1]$ with 1 being the maximum similarity.

When internal similarity of classes is taken into account, we need to add l_w, m_w and a_w to the weights previously presented. Our final cost function is then represented by (c_{nm}, c_{no}, c_{eo}, c_{ems}, c_{eml}, l_w, m_w, a_w).

In [14], these parameters were experimentally tuned using a "trial and error" procedure with a small case study (LaTazza application). Finally, the following values: (c_{nm}, c_{no}, c_{eo}, c_{ems}, c_{eml}, l_w, m_w, a_w)= (50, 25, 10, 30, 25, 0.7, 0.25, 0.05) were considered adapt for studying software evolution. It means, for example, that the cost of a node matching error (c_{nm}) is set 2 times the cost of adding/removing a node (c_{no}) and that the cost function give more importance to the name of the class than to the signatures of methods and attributes.

Further details on how modeling class diagram evolution with an ECGM algorithm can be found in [14].

2.3 The Evolution Cost (EC)

When our ECGM algorithm is applied to two subsequent versions V_1 and V_2 of a software, an EC is assigned at each class of V_2. Roughly speaking, this value expresses the amount of change a class goes through from version V_1 to V_2.

The EC of the class X from V_1 to V_2 (represented as $EC(X_{1->2})$) is the sum of internal and structural changes.

$$EC(X_{1->2}) = c_{int}(X_{1->2}) + c_{struct}(X_{1->2})$$

Internal changes are the changes occurring for: the name of the class, the set of attributes and the class signature. The $c_{int}(X_{1->2})$ cost is computed by multiplying the maximum fixed cost (c_{nm}), experimentally assigned to 50 (see [14]), with the internal dissimilarity of the class X in the two different versions (X_1 and X_2).

$$c_{int}(X_{1->2}) = c_{nm} \times (1 - intSim(X_1, X_2))$$

Instead, structural changes are the changes affecting the class relations with its peers. Three cases are possible. A modified relation with a connected class (e.g, an association in V_1 becomes an aggregation in V_2), a relation missing in V_2 but existing in V_1 (e.g., the class X in version V_2 no longer uses another class), a

[3]the Levenshtein distance between two strings is given by the minimum number of edit operations (insertion, deletion or substitution of a single character) needed to transform one string into the other.

[4]The Jaccard index is a statistic providing a measure of similarity defined as the size of the intersection divided by the size of the union of the sets.

new relation with another class (e.g., a relation missing in V_1 but existing in V_2). Depending on the number and type of edit operations used by the ECGM algorithm to transform the class X from V_1 to V_2 we will have a different c_{struct} cost. The $c_{struct}(X_{1->2})$ cost is computed by the ECGM algorithm using the following formula[5]:

$$c_{struct}(X_{1->2}) = n1 \times c_{eo} + n2 \times c_{ems} + n3 \times c_{eml}$$

An EC equal to zero means that the class is not changed — from a version to another — in its fundamental parts (name, attributes, methods signature, relations with peers), while a high EC means that the class has changed a lot.

2.4 The Class Rank (CR)

Our ECGM algorithm relies on a Tabu Search algorithm guided by global information on the nodes from the PageRank algorithm and local node features such as the number of incoming/outgoing edges. PageRank [19], one of the main components behind the first versions of Google, basically measures the relative importance of each element of a hyperlinked set and assigns it a numerical weighting. In essence, the more references (incoming arcs) a vertex gets from other elements (preferably important), the more importance it deserves. Note that if a node is the only reference of a very important node, it might be valued more important than another node getting some references from low ranked nodes.

More precisely, let u be a node. Then let F_u be the set of nodes u points to and B_u be the set of nodes that point to u. Let $N_u = |F_u|$ be the number of links from u and let k be a factor used for normalization — to force the range of $PageRank(u)$ in $[0, 1]$. PageRank[6] (a slightly simplified version) is defined as:

$$PageRank(u) = k \times \sum_{v \in Bu} \frac{PageRank(v)}{N_v}$$

The CR of each class is its PageRank score in the corresponding graph generated by the class diagram. Relations are not differentiated, meaning that we don't use the type of relations to weight a relation from one class to the other.

3 Case Study

The description of the study follows the Goal-Question-Metrics paradigm [1]. The *goal* of this empirical study is to investigate the applicability of our ECGM based software metrics to identify error prone classes. The *quality focus* is achieving a prediction accuracy better than constant and random classifiers. The *perspective* is both of researchers and of programmers who often use metrics to identify error prone classes deserving more testing effort. The *context* of this study is an open-source system: the Rhino JavaScript/ECMAScript interpreter.

3.1 Object

We selected Rhino as software application for our case study since: (i) several versions are available, (ii) it was previously used in other case studies [7] and, more important, (iii) authors of [7] made available the mapping defects-classes for the version 1.6R5. Rhino is a JavaScript/ECMAScript interpreter and compiler developed as a part of the Mozilla project. Version 1.6R5 consists of 32,134 source lines of Java code (excluding comments and blank lines), 138 types (classes, interfaces, and enums), 1,870 methods, and 1,339 fields.

3.2 Research Questions

This work aims at answering the following general research question. At a very high level of abstraction, we would like to investigate if ECGM based metrics can improve prediction accuracy in identifying defect prone classes with respect to more traditional software metrics such as the Chidamber and Kemerer suite [6].

We believe that developers and managers are interested to predict if a given class contains a high or low number of defects. For most applications, it will suffice a classification of classes into categories such as no-defects, very low number of defects, a few defects, high defect number, and extremely defect prone. Therefore, this case study was designed to examine the following specific research questions, refining the general research question presented above:

- **RQ1 – metrics relevance:** do ECGM based software metrics contribute to explain the number of defects discovered in Rhino 1.6R5?
- **RQ2 – defect prone identification accuracy:** on Rhino 1.6R5, does a binary classifier where ECGM based software metrics are used as independent variables to predict error prone classes perform better than a random classifier, a constant classifier and a classifier built with Chidamber and Kemerer metrics suite?
- **RQ3 – defect prone classification accuracy:** when ECGM based software metrics are used to

[5] n1=number of edge deletions/insertions, n2=number of edge structural errors and n3=number of edge label errors.

[6] further details on it can be found in [19].

categorize Rhino 1.6R5 classes into buckets such as defect free, with few defects, highly defect prone and so on, does the accuracy improve with respect to a classifier built solely with traditional OO metrics?

3.3 Analysis Method

RQ1 aims at providing evidence that a correlation between newly proposed software metrics and the defect proneness property actually exists; in other words, that they help to explain the dependent variable *number of discovered defects*. Linear regression models (regression models for short) were used as sanity check: if new metrics are important to explain class defect proneness then they should be included in a linear model and increase R^2 (**RQ1**). R^2 measures how well a regression approximates the real data points; R^2=100% indicates that the regression perfectly fits the data.

Logistic regression and logistic classifiers built on top of logistic regression models were used to decide if a class is defect free or defect prone and thus to answer **RQ2**. Logistic regression classifiers including newly proposed metrics should improve accuracy over OO metrics (**RQ2**); logistic regression has been used for the same purpose elsewhere (e.g., [10]).

Finally, to answer **RQ3** we applied ClAssification and Regression Trees (CART) as they have been used by other researchers for similar purposes (e.g., [12]). Moreover, they are a useful paradigm to obtain models easily interpretable.

Models with a small number of independent variables were preferred, for ease of use and interpretation, and because the number of data points is still not very large (i.e., 107 data points - Rhino classes).

Linear regression models were build via backward elimination and assessed via basic diagnostics [20] (e.g., `F-test, t-test`, statistical significance, distribution of residuals, QQ-plots). At standard significance level (i.e., 5% and 10%), intercepts were never significantly different from zero; thus regression models described in this paper are forced to pass through the origin.

In a logistic regression based classifier, the dependent variable is commonly a dichotomous variable and, thus, it assumes only two values $\{0, 1\}$ — in our case defect free and error prone. The multivariate logistic regression classifier is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0+C_1\cdot X_1+\ldots+C_n\cdot X_n}}{1+e^{C_0+C_1\cdot X_1+\ldots+C_n\cdot X_n}}$$

where X_i are the characteristics describing the modeled phenomenon, and $0 \leq \pi \leq 1$ is a value on the logistic regression curve. In our problem, variable X_i will be metrics quantifying structural or evolution properties. Thus, the closer $\pi(X_1, X_2, \ldots, X_n)$ is to 1, the higher is the probability that the class contains defects. To use that model as a classifier, we fixed the decision threshold at 0.5 thus if $\pi(X_1, X_2, \ldots, X_n) > 0.5$. the class is considered error prone.

CART classifiers are built by maximizing some gain or minimizing a cost function, representative of the accuracy of the classifier with respect to the a-priory classification. Often, tree nodes are split to minimize the node level of variability given the a-priori classification and thus attempting to balance node variability and prediction accuracy on the training data [22].

All computations were performed with the R[7] programming environment; in particular, we used the `R-tree` package implementing CART.

3.4 Model assessment

Two types of model assessment were performed. First models were build and validated on the entire data set; this corresponds to train set accuracy evaluation. To identify most promising logistic regression models, basic diagnostics were applied on models trained with all available data. In a similar way, CART relevant variables were selected based on a minimization of node variability and error rate calculated on all available data.

Second, available data were divided into disjoint training and test sets. A leave-one-out cross-validation procedure was used to measure model performance. Each given model was trained on $n-1$ points of the data set L (sample size $n = 107$) and accuracy tested on the withheld datum. The step was repeated for each point in L and accuracy measures averaged over n. This methodology gives an unbiased estimate of future performance on novel data, in the field usage, and it is thus indicated in the design of predictive models. Moreover, different choices of model class, or parameter tuning, or variable included or excluded become directly comparable.

3.5 Execution

We downloaded ten subsequents versions of Rhino from the Mozilla Web site[8] (the last is version 1.6R5). All artifacts (snapshots, class diagrams, graph representations) used in this study can be downloaded from the SOCCER website[9]. Out of the 138 types of Rhino

[7] http://cran.r-project.org/
[8] *https : //developer.mozilla.org/*
[9] http://web.soccerlab.polymtl.ca/SER/

(classes, interfaces, and enumerations), we focused only on the 107 Java classes to build error proneness models. Class diagrams were reverse engineered with available tools, mapped into graphs and graph evolution studied with our ECGM algorithm. Finally, ECGM based metrics and traditional OO metrics were extracted to build models. Given that the ECGM algorithm is a meta-heuristic algorithm different runs may produce slightly different results of EC. We ran it ten times obtaining the same values of EC. This is due to the fact that Rhino is a small application and that the variation between the considered versions is not so big. Results on ECGM matching stability will be presented in the forthcoming CSMR paper [16] suggesting that actually the matching variation is very limited even for two versions quite distance (i.e., five years) in time of a large system (Mozilla).

It is important to notice, that 41 classes do not contain any defect, thus a classifier answering always *this is a buggy class* will make an error of 38,32% with a 61,68% correct decisions. In a similar way, a random binary classifier, equivalent to tossing a coin, will make the correct decision, on average, 47% of cases. In other words, a system to be useful should have an error rate lower than 38,32% when classifying classes in the two categories defect free and error prone.

3.6 Metrics used

In this subsection we explain in detail the OO and ECGM metrics used to build the models for the Rhino case study.

We chose to compare our metrics with Chidamber and Kemerer OO metrics [6] because they have been previously validated in other experiments and used with success for defects prediction (e.g., [4, 10]).

We implemented a Java and C++ metrics extractor tool on top of srcML[10]. In our approach, srcML maps Java and C++ source code into XML documents. Our metrics extractor, written in Java, reads and visits XML documents and outputs OO metrics. Our tool extracts the Chidamber and Kemerer metrics suite: Response For a Class (RFC), Lack of COhesion on Methods (LCOM), Coupling Between Objects (CBO), Weighed Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC) and Line Of Code (LOC). Notice that there is an ambiguity in WMC computation; WMC was defined [6] as the sum of methods complexity, however, often (e.g, [10]) methods are assumed to have all complexity one. Our tool strictly follows [6]; it calculates WMC as the sum of cyclomatic method complexities, we therefore added to the metrics suite the number of methods implemented by the class. Finally, we also added as metric the number of attributes. LCOM is computed following [6] thus it cannot be negative [2]. Overall, our OO metrics are a super-set of [10].

[10]http://www.sdml.info/projects/srcml/

3.6.1 ECGM metrics

Given a class X in a version V_n of a software and r previous versions of the same software, we define the two main metrics: $EC(X)$: as the sum of all the EC of a class X from V_{n-r} to V_n[11]; $CR(X)$: as the PageRank score of the class X in the target version V_n.

In [15], we observed that EC and CR evolve over time. Furthermore, we also observed that metrics dynamic may cover several orders of magnitude. We believe that a high variability, for example captured by standard deviation of EC or CR, can be representative of some important changes and thus, classes with high values of EC (CR) standard deviation are more likely to be defect prone. Also classes having, on average, higher EC (or CR) values are changed more often in a time frame and for this reason, they deserve a major attention. Thus, we derived the following metrics to capture behavior over time:

- $MeanEC(X)$: as the mean of the all $EC(X_{i->i+1})$, for $i = n-r \ldots n$
- $StdEC(X)$: as the standard deviation of all $EC(X_{i->i+1})$, for $i = n-r \ldots n$
- $MeanCR(X)$: as the mean of $CR(X_i)$, for $i = n-r \ldots n$
- $StdCR(X)$: as the standard deviation of $CR(X_i)$, for $i = n-r \ldots n$
- $logM$: as the log(1+M), M being one of the above defined metrics.

We also use the logarithm to compress the metrics dynamic. Although the information captured is the same, the visual inspection may be easier once dynamic is compressed via logarithm [15].

[11]$EC(X) = \sum_{i=n-r}^{n-1} EC(X_{i->i+1})$

3.7 RQ1 - Linear Regression Models

This fundamental preliminary analysis serves to answer **RQ1** by verifying that proposed metrics correlate with the number of defects discovered into classes.

R backward elimination (i.e., the R `step` command) penalizes models with a low likelihood and having too

many parameters. This however does not provide a diagnostic about the fit of the model. Backward elimination starts with a full model including all explanatory variables and removes one variable at the time.

On our data set the backward elimination procedure gives a model containing as explanatory variables the number of attributes and methods, the class size in LOC, RFC and LCOM plus CR, EC, StdEC, logMeanCR, logEC and logStdEC. Model R^2 is 86.48%; it has a p-value lower than $1e-16$ and each variable is significant.

Once backward elimination is performed only on traditional OO metrics the best model has an R^2 of 82.66% compared with the 86.48% of the mixed model. Explanatory variable included are the number of attributes and methods, RFC and the class size. Included variables are all significant at 95% or better; the model has a p-value of $2e-16$.

Overall, we can answer to **RQ1** in affirmative way, we conclude that newly proposed metrics actually correlate with defect number and help to explain class error proneness.

3.8 RQ2 - Logistic Regression Classifiers

To answer **RQ2** we manually performed a backward elimination procedure to determine the variables to be included in a logistic regression model. Among OO metrics only RFC was retained. A similar process was performed for ECGM based metrics and also in this case only EC was found relevant. Both model have no term C_0 and as shown in Table 1 the coefficient of the explanatory variable is highly significant.

Variable	Coefficient	p-value
RFC	0.010795	0.00197
EC	0.006699	0.000103

Table 1. Logistic regression models

To evaluate predictive accuracy on unseen data we performed cross validation and compared the two models (one based on RFC and the other on EC) via the confusion matrices[12]. The goal here is to assign each class to one of the two categories: the defect free (0) or error prone category (1).

Table 2 shows the 2x2 confusion matrices for the two models. Position (1,1) of the matrix correspond to number of true negative, defect free classes predicted to be defect free while entry (1,2) reports false positive, classes with no defect but predicted as error prone. The entry (2,1) is the false negative classification, classes containing defects classified defect free. Finally, cell (2,2) contain the true positives. Clearly the RFC model has a higher error rate (39,25%) than EC model (30,84%) and it is close to the constant classifier always answering *this class contain defects* (38,22%).

a) RFC

	0	1
0	1	40
1	2	64

b) EC

	0	1
0	12	29
1	4	62

Table 2. RFC (left) and EC (right) Confusion matrices logistic regression classifiers

On the other hand, the model using the EC as explanatory variable classifies as defect free four error prone classes, this is the double of the OO based model. Still the false negative error remains low (about 5%), lower than the false positive error (71%). Strangely enough on Rhino 1.6R5, no improvement was found by combining variables into more complex logistic classifiers.

Overall, the intuition on EC usefulness is confirmed and we can answer to **RQ2** in affirmative way. As we have seen, the EC classifier has a cross validation error rate better than the constant classifier and RFC classifier. Regarding the other proposed metric (CR) we can not conclude anything.

3.9 RQ3 - Classification and Regression Trees

To answer **RQ3**, we apply CART to predict the level of defect proneness. We defined five categories as reported in Table 3. Division in categories was inspired by percentiles and the need to model the special category of classes with zero defects (41 classes). Indeed, the median of defect number is two; the upper quartile six, thus 50% of classes contain between one and six defects. We subdivided the range from one to 100 in four regions. Overall we had five levels of defect proneness: from zero defects to extremely error prone.

When fitting a CART over the entire data set, CART variable selection procedure retains as relevant OO metrics as well as ECGM metrics thus further confirming the positive answer to **RQ1**.

More in detail, among OO metrics, the number of attributes, WMC, LCOM and RFC are retained. The model also included as relevant variables EC, logMeanEC, StdEC and MeanEC. On the entire data set of 107 classes, the classification error is 29%. This clas-

[12] A confusion matrix is a visualization tool typically used in supervised learning where each column of the matrix represents the instances in a predicted class, while each row represents the instances in an actual class.

sification error is the training set error and thus an upper limit for the accuracy.

Category	From	To
0 - defects free	0	0
1 - few defects	1	5
2 - medium defect prone	6	15
3 - highly defect prone	16	20
4 - extremely defect prone	21	100

Table 3. CART mapping of number of defects into categories.

Following this first phase, also for CART we performed a cross validation to evaluate model accuracy on unseen data. Starting with the explanatory variables retained in the previous phase, we manually selected variables to obtain the most accurate model and in particular to minimize the number of faulty classes classified as error free. Table 4 shows the confusion matrices for a mixed model (left) and a model based solely on OO metrics (right). The mixed model of Table 4.a uses as explanatory variables EC, MeanCR, logMeanEC, CBO and the class size; it has an error rate of 29% when the faulty versus error free classification is considered (i.e., first row and first columns of Table 4.a but the class 0). The most interesting fact is that the model classifies only 17 faulty classes (marked in bold in Table 4.a as non faulty and 11 of these classes contains less than five defects (they are in the category 1). In other words, its false negative error is close to 25%, this is higher then the false negative rate of the logistic classifier, however, CART provides a more fine grain classification. It is worth mentioning that a simplified CART build with the sole ECGM metrics performs significantly worse than the more complex model (Table 4.a) and such a model has a false negative rate of 37% with an error rate of 34%.

When the model is built using only OO metrics the most accurate model is the one built with the number of attributes and methods, the size, WMC, CBO, RFC and LCOM. As we can see confronting the two confusion matrices (Table 4), ECGM metrics positively contribute to an accurate classification. Overall, Table 4.b), such a model attains an 45% error rate. It classifies as error free 30 classes containing at least one error (marked in bold in Table 4.b) thus almost the double of the mixed model.

	0	1	2	3	4
0	27	9	4	1	0
1	**11**	20	4	0	2
2	**4**	4	3	1	2
3	**2**	0	0	0	0
4	**0**	1	3	0	9

a) OO and ECGM metrics

	0	1	2	3	4
0	22	13	5	1	0
1	**17**	17	3	0	0
2	**8**	3	3	0	0
3	**1**	0	1	0	0
4	**4**	1	0	0	8

b) OO metrics

Table 4. Confusion matrix for the best models. OO plus ECGM metrics model (left) and OO metrics model (right)

In summary, we can answer to **RQ3** in affirmative way. The model including newly proposed metrics has a lower error rate than a model limited to OO metrics (29% vs. 34%), is more accurate, and it has a lower false negative recognition rate (25% vs. 37%).

3.10 Threads to Validity

This subsection discusses threats to validity that can affect our study. First and foremost, it is important to underline that this is a preliminary study aiming at verifying if at least for Rhino 1.6R5 EC, CR and derived quantities were related to class defect proneness and thus useful in identifying defect prone classes. Intuition as well as Rhino evidence suggest that EC and CR are useful, but we did not formally investigated EC and CR following the guidelines of measurement theory [9]. We plan to do this as part of our future work while addressing the threats to external validity.

Threats to *construct validity* concern the relationship between the theory and the observation. In this case, the threat can be mainly due to the use of incorrect defect classification or operational measures concerning the investigated phenomenon (i.e., collect metrics). In our study we used material and defects manually classified and used in other studies [7]. However, we cannot be absolutely sure that defect classification is 100% correct as sometimes there is a level of subjectivity in deciding if an issue is a defect or another maintenance activity. Metrics extraction was based on home-made tools already used in other experiments and metrics values were manually assessed for subset of classes.

Threats to *internal validity* concern any confounding factor that could influence our results. In particular, this kind of threats can be due to a possible level of subjectiveness caused by the manual construction of oracles, and to the bias that can be introduced by the manual defect classification. We attempted to avoid any bias in the building of the oracle by adopting a classification made available by other researchers [7]. Another factor influencing results is the choice of costs used by our ECGM algorithm. We selected matching costs on a different case study to maximize agreement with small software well documented class diagram evolution. However, we cannot be sure that changing ECGM costs will not affect accuracy of cross valida-

tion results or backward elimination retained variables and we plan to verify this as part of our future work.

Threats to *external validity* concern the possibility of generalizing our results. The study is limited to one system, Rhino, and defects of one release (1.6R5). Although the approach is perfectly applicable to other systems, we do not know whether the same results will be obtained with other case studies. In addition, we have built different classifiers and we cannot be sure that the relative performance will remain the same on different systems or versions. Furthermore, on different systems or versions, variable selection procedure can lead to build different models with different sets of variables.

4 Related Work

Approaches and algorithms to study the evolution of software systems are manifold, but no previous work applies ECGM algorithms to software evolution. For example, Xing and Stroulia in [23] propose an algorithm named UMLDiff for matching different versions of an application, using several class metrics. Similarly to our ECGM algorithm, it recovers the design from the code in an intermediate representation and compares it with subsequent software releases.

Several researchers have found correlations between static object-oriented metrics, such as the Chidamber and Kemerer metrics [6], and fault-proneness. For example, Khoshgoftaar et al. [11] used 16 static software metrics as predictors of software quality in a large legacy system for telecommunications (over 38,000 procedures in 171 modules). Similarly to us the results of the predictors were interesting.

Another empirical study [5] conducted on an industrial C++ system (over 133 KLOC) supports the hypothesis that classes in inheritance structures are more defect prone. It follows that Chidamber and Kemerer's DIT and NOC metrics could therefore be used to find classes that are likely to have higher defect densities.

More recently, Gyimothy et alt. [10] compare the accuracy of "an array" of different Chidamber and Kemerer metrics to predict fault-proneness classes within the open-source project Mozilla. They conclude that the CBO metric is the best metric. They also found LOC to perform this task fairly well.

The experiment [24] conducted using Eclipse as case study goes in the same direction of the others. It essentially showed that a combination of complexity metrics can predict defects, suggesting that the more complex the code is, the more defects it has. However, El Eman in [8] showed that after controlling for the confounding effect of "size", the correlation between static object-oriented metrics and fault-proneness disappeared. The reason is that many OO metrics are correlated with size, and therefore they "add nothing" to the models that predict the fault-proneness.

Other researchers used historical data [18] and code churn [17] (i.e., number of lines added or deleted between versions) to predict software defect density. We share with them the intuition that frequently changed classes in the past are the most fault-prone.

5 Conclusion

In this paper, we have proposed to adopt ECGM and PageRank to build evolution aware software metrics with the goal of identifying error prone classes.

To obtain empirical evidence of a relation between proposed evolution aware metrics and class error proneness, we performed a case study on the Rhino Java application answering to three research questions: **RQ1** on metrics relevance, **RQ2** on defect prone identification accuracy, and **RQ3** on defect prone classification accuracy. More precisely, we build defect prediction models and classifiers using traditional OO and newly proposed metrics. On manually classified Rhino version 1.6R5 defects, we obtained statistical evidence that the newly proposed metrics improve defect prediction accuracy and error proneness classification. Overall, empirical evidence allows us to positively answer to the three research questions. However, it pays to be cautious as identifying defect prone classes is a difficult task and we have empirical evidence limited to a single data point (Rhino version 1.6R5).

Our future work will be devoted to replicate this case study on different applications to improve empirical evidence of evolution aware metrics usefulness and more in general to address threats to validity.

6 Acknowledgment

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (Research Chair in Software Evolution #950-202658) and by G. Antoniol Individual Discovery Grant.

References

[1] V. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm Encyclopedia of Software Engineering.* John Wiley and Sons, 1994.

[2] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-

oriented systems. *Empirical Softw. Eng.*, 3(1):65–117, 1998.

[3] L. C. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Trans. on Software Engineering*, 29(7):594–607, 2003.

[4] L. C. Briand, W. L. Melo, and J. Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. on Software Engineering*, 28(7):706–720, 2002.

[5] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Trans. on Software Engineering*, 26(8):786–796, August 2000.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. on Software Engineering*, 20(6):476–493, June 1994.

[7] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Transaction on Software Engineering*, 34(4):497–515, 2008.

[8] K. E. Emam, S. Benlarbi, N. Goel, and S. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. on Software Engineering*, 27(7):630–650, July 2001.

[9] N. Fenton and S. Pfleeger. *Software Metrics: A Rigorous and Practical Approach (2nd Edition)*. Thomson Computer Press, Boston, 1997.

[10] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transaction on Software Engineering*, 31(10):897–910, 2005.

[11] T. Khoshgoftaar., B. Allen, K. Kalaichelvan, and N. Goel. Early quality prediction: a case study in telecommunications. *IEEE Software*, 13(1):65–71, 1996.

[12] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *MSR '06: Proceedings of the international workshop on Mining software repositories*, pages 119–125, New York, NY, USA, 2006.

[13] S. Kpodjedo, P. Galinier, and G. Antoniol. A google-inspired error correcting graph matching algorithm. Technical Report EPM-RT-2008-06, Ecole Polytechnique de Montreal, 06 2008.

[14] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol. Error correcting graph matching application to software evolution. In *Proc. of the Working Conference on Reverse Engineering*, pages 289–293, 2008.

[15] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol. Not all classes are created equal: toward a recommendation system for focusing testing. In *RSSE '08: Proc. of the International Workshop on Recommendation Systems for Software Engineering*, pages 6–10, New York, NY, USA, 2008.

[16] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol. Recovering the evolution stable part using an ecgm algorithm: is there a tunnel in mozilla. In *To appear in Proc. of IEEE European Conference on Software Maintenance and Reengineering*, 2009.

[17] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 284–292, 2005.

[18] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. on Software Engineering*, 31:340–355, 2005.

[19] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[20] J. Rawlings, S. G. Pandula, and D. A. Dickey. *Applied Regression Analysis a Research Tool*. Springer Texts in Statistics. New York: Springer-Verlag, second edition, 1998.

[21] W. Tsai and K.-S. Fu. Error-correcting isomorphism of attributed relational graphs for pattern analysis. *IEEE Trans. on Systems, Man, and Cybernetics*, 9:757768, 1979.

[22] I. Witten and E. Frank. *Data Mining Practical Machine Learning Tools and Techniques - Second Edition*. Elsevier, 2005.

[23] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005.

[24] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, May 2007.

Search-Based Prediction of Fault Count Data

Wasif Afzal*, Richard Torkar and Robert Feldt
Blekinge Institute of Technology,
S-372 25 Ronneby, Sweden
{*waf,rto,rfd*}*@bth.se*

Abstract

Symbolic regression, an application domain of genetic programming (GP), aims to find a function whose output has some desired property, like matching target values of a particular data set. While typical regression involves finding the coefficients of a pre-defined function, symbolic regression finds a general function, with coefficients, fitting the given set of data points. The concepts of symbolic regression using genetic programming can be used to evolve a model for fault count predictions. Such a model has the advantages that the evolution is not dependent on a particular structure of the model and is also independent of any assumptions, which are common in traditional time-domain parametric software reliability growth models. This research aims at applying experiments targeting fault predictions using genetic programming and comparing the results with traditional approaches to compare efficiency gains.

1. Introduction to the research problem

A software fault is a defect in an executable product that causes system failures during operations [11]. The number of faults in a software module or particular release of a software system represents the quantitative measure of software quality. A fault prediction model then uses previous software quality data in the form of metrics (including software fault data) to predict the number of software faults in a module or release of a software system [12]. The practical aspect of such models has strong implications on the quality of the software project. The information gained from such models can be an important decision making tool for the project managers to make better decisions in uncertain situations. A fault prediction model helps a software development team in prioritizing the effort to be spent on a software project. If the predictions forecasts a high number of faults in the coming release of a project, then the management has the option of investing required levels of effort to circumvent possible failures in operation. Proper allocation of resources for quality improvement might cause considerable savings for a software project. The development of large software systems is costly therefore even small gains in prediction accuracy should be appreciable [10]. Apart from the efficiency gains, architectural improvements can be made by better designing high-risk segments of the system [14].

There have been a number of software fault prediction and reliability growth modeling techniques proposed in software engineering literature [9, 5]. Despite the presence of large number of models, there is no agreement within the research community about the best model. One of the reasons for such a situation is that models exhibit different predictive accuracies across different data sets. Therefore, the quest for a consistently accurate predictor model is continuing. The result is that the prediction problem is seen as being largely unsolvable and NP-hard: *the ability to build prediction systems for software engineers remains an important but largely unsolved problem...due to the fact that the problem is NP-hard* [23] *...this problem is largely unsolvable* [5].

2. Genetic programming for predictions

The use of statistical regression analysis (e.g., linear, logarithmic and logistic) for software fault predictions may not be the best approach. This argument is supported by the fact that software engineering data come with certain characteristics that creates difficulties in making accurate software prediction models. These

*Wasif Afzal is a PhD student, advised by Richard Torkar and Robert Feldt, at the Department of Systems and Software Engineering, Blekinge Institute of Technology, Sweden. This paper is written specifically for the PhD forum.

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.17

characteristics include missing data, large number of variables, strong collinearity between the variables, heteroscedasticity[1], complex non-linear relationships, outliers and small size [10]. Therefore, it is not surprising that we possess an incomplete understanding of the phenomenon under study, so *it is very difficult to make valid assumptions about the form of the functional relationship between the variables* [4]. This reason is also highlighted by [21]. This argument strengthens earlier established results that show program metrics to be insufficient for accurate prediction of faults. Moreover, the acceptability of models has seen little success due to lack of meaningful explanation of the relationship among different variables and lack of generalisabilty of model results [10]. Additionally, these *parametric* models are often characterized by a number of assumptions [9] that are necessary for developing a mathematical model. These assumptions are often violated in real-world situations (see e.g. [24]), therefore, causing problems in the long-term applicability and validity of these models.

Under this scenario, what becomes significantly interesting is to have modeling mechanisms that can exclude the pre-suppositions about the model and is based entirely on the fault data. This is where the application of symbolic regression using genetic programming (GP) becomes feasible. The advantages of using GP for symbolic regression problems are [20]:

1. GP, being a non-parametric method, does not conceive a particular structure for the resulting function. Therefore, the evolved model truly represents the data, be it linear or non-linear.
2. The model and the associated coefficients are evolved based on the fault data collected during the initial test phase.
3. The equations are derived according to the fitness evaluation criterion of the individuals only, since GP does not make any assumptions about:
   (a) The distribution of the data.
   (b) Relationship between independent and dependent variables.
   (c) The stochastic behavior of software failure process.
   (d) The nature of software faults.

3. Related work

Studies reporting the use of GP for software fault prediction are few and recent. Costa et al. [6] presented results of two experiments exploring GP models based on time and test coverage. The authors compared the results with other traditional and non-parametric artificial neural network (ANN) models. For the first experiment, the authors used 16 data sets containing time-between-failure (TBF) data from projects related to different applications. The models were evaluated using five different measures, four of these measures represented different variants of differences between observed and estimated values. The results from the first experiment, which explored GP models based on time, showed that GP adjusts better to the reliability growth curve. Also GP and ANN models converged better than traditional reliability growth models. GP models also showed the lowest average error in 13 out of 16 data sets.

For the second experiment, which was based on test coverage data, a single data set was used. This time the Kolmogorov-Smirnov test was also used for model evaluation. The results from the second experiment showed that all metrics were always better for GP and ANN models. The authors later extended GP with boosting techniques for reliability growth modeling [19] and reported improved results. A similar study by Zhang and Chen [25] used GP to establish software reliability model based on mean time between failures (MTBF) time series. The study used a single data series and used six different criteria for evaluating the GP evolved model. The results of the study also confirmed that in comparison with the ANN model and traditional models, the model evolved by GP had higher prediction precision and better applicability.

Our research using GP extends these previous studies. We focus on using *cumulative* fault count data for modeling and investigate different ways to adapt the use of modeling in current trend of *multi-release* software development. We focus on using proven experimental design practices in our research work. We intend to increase the comparison groups and also make use of larger, real-world data sets to question the generalizability of our results.

3.1. Authors' contribution and preliminary work

In the preliminary stage of our research, we evaluated the use of GP for fault predictions in two studies ([3, 1]). In the very first study [3], we evaluated the results of using GP for modeling weekly fault count data of three industrial projects in terms of goodness of fit and predictive accuracy. The results found were statistically significant in favor of GP. We later extended the scope and included comparisons with three traditional reliability growth models [1]. In terms of evaluating

[1] A sequence of random variables with different variances.

model validity, three measures were used; two of them showed favorability of GP model, while the goodness of fit of the GP evolved model was also found to be either equivalent or better than the traditional models. Lastly, the predictions of the GP evolved model was found to be less biased than traditional models. We later on, in [2], highlighted the underlying mechanisms that allows GP to progressively search for fitter solutions.

4. Methodology

The overall methodology is discussed in terms of data requirements, GP design and statistical hypothesis testing.

4.1. Fault count data sets

Fault count data sets are required to train the GP evolved models and to evaluate their applicability using various evaluation measures. The fault count data sets resembles a time-series, with faults aggregated either on weekly or monthly basis. The week/month number can be regarded as the independent variable (being controllable) and the corresponding count of faults as the dependent variable in which the effect of the treatment is measured. The data sets needs to be split in to training and test sets. We resort to a typical mechanism, with first $\frac{2}{3}$ of data in each data set for building the model and later $\frac{1}{3}$ of the data for evaluating the model. Such a choice of split preserves the chronological time series occurrence of faults.

4.2. GP design

The representation of solutions in the search space is a symbolic expression in the form of a parse tree, which is a structure having functions and terminals. The quality of solutions is measured using an evaluation function. A natural evaluation measure for symbolic regression problems is the calculation of the difference between the obtained and expected results in all fitness cases, $\sum_{i=1}^{n} \mid e_i - e_i^{'} \mid$ where e_i is the actual fault count data, $e_i^{'}$ is the estimated value of the fault count data and n is the size of the data set used to train the GP models. Various variation operators can be used to grow or shrink a variable length parse tree. Similarly, there are various selection mechanisms that can be used to determine individuals in the next generation. The effectiveness of these operators is problem-dependent [16]. In our experiments, we have used cross-over with branch swapping by randomly selecting nodes of the two parent trees. We have also used mutation in which a random node from the parent tree is substituted with a new random tree created with the available terminals and functions. A small proportion of individuals were also copied into the next generation without any action of operators. The selection mechanism selected a random number of individuals from the population and chose the best of them; if two individuals were equally fit, the one having the less number of nodes was chosen as the best.

4.3. Statistical hypothesis testing

It is important to test results for statistical significance because it is not reliable to draw conclusions merely on observed differences in means or medians because the differences could have been caused by chance alone [17]. Prior to applying statistical testing, suitable accuracy indicators are required. However, there is no consensus with regards as to which accuracy indicator is the most suitable for the problem at hand. Commonly used indicators suffer from different limitations (for details see [7, 22]). One intuitive way out of this dilemma is to employ more than one accuracy indicator, so as to better reflect on a model's predictive performance in light of different limitations of each accuracy indicator. This way the results can be better assessed with respect to each accuracy indicator and we can better reflect on a particular model's reliability and validity. However, reporting multiple measures that are all based on a basic measure like mean relative error (MRE) would not be useful because all such measures would suffer from common disadvantage of being unstable (see [7]). In [18], measures for the following characteristics are proposed: Goodness of fit (Kolmogorov-Smirnov test), Model bias (U-plot), Model bias trend (Y-plot) and Short-term predictability (prequential likelihood). These measures, although providing a thorough evaluation of a model's predictions, lacks a suitable measure for variable-term predictability.

In [8, 15], average relative error is used as a measure of variable term predictability. To our knowledge, we are not aware of any critique of such an approach for variable-term predictability. As an example of applying multiple measures, one of our recent studies [1] used measures of prequential likelihood, Braun statistic and adjusted mean square error for evaluating model validity. Additionally we examined the distribution of residuals from each model to measure model bias. Lastly, the Kolmogorov-Smirnov test was applied for evaluating goodness of fit. More recently, analyzing distribution of residuals is proposed as an alternative measure [13, 22]. It has the convenience of applying significance tests and visualizing differences in absolute residuals of compet-

ing models using box plots.

5. Conclusions

This paper presented the synopses of the research conducted so far that evaluates the use of genetic programming for predicting fault count data. Initial studies have produced better or comparable results to traditional models (see [1]). This encourages further testing the use of GP for larger data sets and to increase comparisons with other machine learning/traditional approaches. Future work includes evaluating the use of GP for *cross-release* predictions using extensive data sets covering both commercial and open source software systems. Going further, the applicability of the approach will be assessed in an on-going project in an industrial context.

References

[1] W. Afzal and R. Torkar. A comparative evaluation of using genetic programming for predicting fault count data. In *Proceedings of the Third International Conference on Software Engineering Advances*. IEEE Computer Society, 2008.

[2] W. Afzal and R. Torkar. Suitability of Genetic Programming for Software Reliability Growth Modeling. In *The 2008 International Symposium on Computer Science and its Applications*. IEEE Computer Society, 2008.

[3] W. Afzal, R. Torkar, and R. Feldt. Prediction of fault count data using genetic programming. In *Proceedings of the 12th IEEE International Multitopic Conference*. IEEE, 2008.

[4] L. C. Briand, V. R. Basili, and W. M. Thomas. A pattern recognition approach for software engineering data analysis. *IEEE Trans. Softw. Eng.*, 18(11):931–942, 1992.

[5] V. Challagulla, F. Bastani, I.-L. Yen, and R. Paul. Empirical assessment of machine learning based software defect prediction techniques. *10th International Workshop on Object-Oriented Real-Time Dependable Systems*, 2005.

[6] E. Costa, S. Vergilio, A. Pozo, and G. Souza. Modeling software reliability growth with genetic programming. *International Symposium on Software Reliability Engineering*, 2005.

[7] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit. A simulation study of the model evaluation criterion MMRE. *IEEE Transactions on Software Engineering*, 29(11), 2003.

[8] K. Gao and T. Khoshgoftaar. A comprehensive empirical study of count models for software fault prediction. *IEEE Transactions on Reliability*, 56(2), June 2007.

[9] A. L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, SE-11(12):1411–1423, 1985.

[10] A. Gray and S. MacDonnell. A comparison of techniques for developing predictive models of software metrics. *Information and Software Technology*, 39(6):425–437, 1997.

[11] T. Khoshgoftaar, N. Seliya, and N. Sundaresh. An empirical study of predicting software faults with case-based reasoning. *Software Quality Control*, 14(2), 2006.

[12] T. M. Khoshgoftaar and N. Seliya. Tree-based software quality estimation models for fault prediction. In *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, Washington, DC, USA, 2002. IEEE Computer Society.

[13] B. Kitchenham, L. Pickard, S. MacDonell, and M. Shepperd. What accuracy statistics really measure. *IEE Proceedings Software*, 148(3), Jun 2001.

[14] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[15] Y. Malaiya, N. Karunanithi, and P. Verma. Predictability measures for software reliability models. *COMPSAC 90*.

[16] Z. Michalewicz and D. Fogel. *How to Solve It: Modern Heuristics*. Springer-Verlag, second edition, 2004.

[17] I. Myrtveit and E. Stensrud. A controlled experiment to assess the benefits of estimating with analogy and regression models. *IEEE Transactions on Software Engineering*, 25(4), July-Aug. 1999.

[18] A. Nikora and M. Lyu. An experiment in determining software reliability model applicability. *ISSRE*, 1995.

[19] E. Oliveira, A. Pozo, and S. R. Vergilio. Using boosting techniques to improve software reliability models based on genetic programming. *IEEE International Conference on Tools with Artificial Intelligence*, 2006.

[20] R. Poli, W. Langdon, N. McPhee, and J. Koza. Genetic Programming: An Introductory Tutorial and a Survey of Techniques and Applications. Technical Report CES-475, ISSN: 1744-8050, 2007.

[21] M. Reformat, W. Pedrycz, and N. Pizzi. Software quality analysis with the use of computational intelligence. *Fuzzy Systems, 2002. FUZZ-IEEE'02. Proceedings of the 2002 IEEE International Conference on*, 2:1156–1161, 2002.

[22] M. Shepperd, M. Cartwright, and G. Kadoda. On building prediction systems for software engineers. *Empirical Software Engineering*, 5(3), 2000.

[23] M. Shepperd and G. Kadoda. Comparing software prediction techniques using simulation. *IEEE Transactions on Software Engineering*, 27(11):1014–1022, 2001.

[24] A. Wood. Software reliability growth models: assumptions vs. reality. In *ISSRE '97: Proceedings of the 8th IEEE International Symposium on Software Reliability Engineering*, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[25] Y. Zhang and H. Chen. Predicting for MTBF failure data series of software reliability by genetic programming algorithm. *6th International Conference on Intelligent Systems Design and Applications (ISDA '06)*, 2006.

On Search Based Software Evolution

Andrea Arcuri
School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, UK.
email: a.arcuri@cs.bham.ac.uk

Abstract

Writing software is a difficult and expensive task. Its automation is hence very valuable. Search algorithms have been successfully used to tackle many software engineering problems. Unfortunately, for some problems the traditional techniques have been of only limited scope, and search algorithms have not been used yet. We hence propose a novel framework that is based on a co-evolution of programs and test cases to tackle these difficult problems. This framework can be used to tackle software engineering tasks such as Automatic Refinement, Fault Correction, Improving Non-functional Criteria and Reverse Engineering. While the programs evolve to accomplish one of these tasks, test cases are co-evolved at the the same time to find new faults in the evolving programs.

1 Introduction

In software engineering there are many tasks that are very expensive, like for example testing the developed software [15]. It is hence important to try to automate these tasks, because it would have a direct impact on software industries.

Re-formulating software engineer as an optimisation problem has led to promising results in the recent years [10, 9]. Many tasks have been addressed by the research community, but some are mainly unexplored. We hence want to feel this gap.

We have designed a novel framework that, with little changes, can be easily applied to automate at least these following software engineering problems:

- *Automatic Refinement*: given as input a formal specification, we want to obtain a correct implementation in an automatic way.
- *Fault Correction*: given as input a program implementation and a set of test cases in which at least one test case is failed, we want to automatically evolve the input program to make it able to pass all the given test cases.
- *Improving Non-functional Criteria*: given as input a program, we want to evolve it to optimise some of its non-functional criteria (e.g., execution time and power consumption) without changing its semantics.
- *Reverse Engineering*: given as input the assembler code or byte-code of a program, we want to automatically derive its source code.

The novel framework we propose is based on co-evolution of programs (evolved for example with Genetic Programing [17]) and test cases (evolved for example with Search Based Software Testing [14]). Programs are rewarded by how many tests they do not fail, whereas the unit tests are rewarded by how many programs they make to fail. This type of co-evolution is similar to what happens in nature between *predators* and *prey*.

Preliminary results confirm that this approach is feasible [4, 2, 5, 3]. However, given the novelty of this approach and the difficulty of these tasks, more research is still required. In particular, we need to analyse whether the proposed approach would *scale* to real-world software.

The paper is organised as follows. Section 2 briefly gives background information of Genetic Programming and Co-evolution. Section 3 describes the novel co-evolutionary framework. The addressed software engineering problems are discussed in section 4. Finally, Section 5 concludes the paper.

2 Background

2.1 Genetic Programming

Genetic Programming (GP) [17] is a paradigm to evolve programs. A genetic program can often be seen as a tree, in which each node is a function whose inputs are the children of that node. A population of programs is held at each generation, where individuals are chosen to fill the next population accordingly to a problem specific fitness function.

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.12

Crossover and mutation operators are applied to the programs to generate new offspring.

2.2 Co-evolution

In co-evolutionary algorithms, one or more populations co-evolve influencing each other. There are two types of influences: *cooperative co-evolution* in which the populations work together to accomplish the same task, and *competitive co-evolution* as predators and prey in nature. In our framework we use competitive co-evolution.

Co-evolutionary algorithms are affected by the *Red Queen* effect [16], because the fitness value of an individual depends on the interactions with other individuals. Because other individuals evolve as well, the fitness function is not static. For example, exactly the same individual can obtain different fitness values in different generations. One consequence is that it is difficult to keep trace of whether a population is actually "improving" or not [7].

One of the first applications of competitive co-evolutionary algorithms was the work of Hillis on generating sorting networks [11]. He modelled the task as an optimisation problem, in which the goal is to find a correct sorting network that does as few comparisons of the elements as possible. He used evolutionary techniques to search the space of sorting networks, where the fitness function was based on a finite set of tests (i.e., sequences of elements to sort): the more tests a network was able to correctly pass, the higher fitness value it got. For the first time, Hillis investigated the idea of co-evolving such tests with the networks. The reason for doing so was that random test cases might be too easy, and the networks can learn how to sort a particular set of elements without being able of generalising. The experiments of Hillis showed that shorter networks were found when co-evolution was used.

Ronge and Nordahl used co-evolution of genetic programs and test cases to evolve controllers for a simple "robot-like" simulated vehicle [20]. Similar work has been successively done by Ashlock et al. [6]. In such work, the test cases are instances of the environment in which the robot moves.

Note that the application area of such work was restricted (i.e., sorting networks and robot controllers), whereas we use co-evolution in a more general and complex framework.

In software engineering, a co-evolutionary algorithm has been used in Mutation Testing [1]. The goal is to find test cases that can recognise faulty *mutants* of the tested software, because such a test suite would be good for asserting the reliability of the software. Mutants (generated with a precise set of rules) co-evolve with the test cases, and they are rewarded on how many tests they pass, whereas the test cases are rewarded on how many mutants they identify as semantically different from the original program.

3 Co-evolutionary Framework

In our novel framework, programs co-evolve with test cases. Programs are rewarded by how many tests they do not fail, whereas the unit tests are rewarded by how many programs they make to fail. For example, a program can obtain a value 0 if it passes a test case and 1 otherwise. The fitness function to minimise would hence be the sum of these values given by the execution of the program on all the given test cases. For the fitness function of the test cases, it would be a maximisation problem.

Depending on which software engineering problem is addressed, there are the following components of the framework that needs to be specialised:

- Input of the framework.
- Initialisation of the genetic programs.
- Oracle used for evaluating the expected outputs of the test cases.
- Generation of new test cases.
- Other objectives for the fitness function.

These specific points will be discussed in the next section.

4 Software Engineering Problems

4.1 Automatic Refinement

Since the 1970s the goal of generating programs in an automatic way has been sought [12]. A user would just define what he expects from the program (i.e., the requirements), and it should be automatically generated by the computer without the help of any programmer.

Unfortunately, this task is much harder than expected [19]. Transformation methods are usually employed to address this problem. The requirements need to be written in a formal specification, and sequences of transformations are used to transform these high-level constructs into low-level implementations. Unfortunately, this process can rarely be automated completely, because the gap between the high-level specification and the target implementation language might be too wide.

We analysed the application of our framework to this problem [4]. The presence of a gap does not preclude the application of our framework. The specifications we used were written in first order logic. Other formal specification languages could be used (e.g., Z [21]), but we would need to implement a fitness function for them.

The input to the framework is a formal specification of the program we seek. The genetic programs are initialised

at random, and the formal specification is used as an oracle. The formal specification can also be used to guide the evolution of more challenging test cases.

4.2 Fault Correction

A lot of research has been done on software testing. However, if the presence of a fault is discovered, it is still duty of the programmers to fix the software. We hence seek to also automate this expensive task. In this research field, the very few proposed techniques (e.g., [23]) have many limitations. Only very restricted types of modifications are allowed, that because these techniques are mainly doing an exhaustive search. The types of modifications they consider do not guarantee that between any two programs there is a sequence of transformations to obtain the second program from the first. So given any faulty program, it could be impossible to fix it. On the other hand, our approach can potentially fix any code level fault. This because we are searching in the entire space of possible programs.

The input of the framework would be a faulty program and a set of test cases in which at least one is failed. The test cases need to be provided (e.g., by a software tester). A formal specification can be given as input as well although it is not essential. The genetic programs are initialised with heuristics based on the input program (e.g., they can simply be copies of it). New test cases can be evaluated against a formal specification, otherwise no new test case can be generated. In that case, the programs are executed only on a subset of the given test cases. The co-evolution will be hence used to choose which subset of test cases to use at each generation. More details can be found in our preliminary work [2, 5].

We can consider fault correction as a special case of automatic refinement, in which the faulty input program is a solution structurally close to a global optimum. The input faulty program is hence heuristically exploited to help the search for a correct refinement. Because programmers do not write code at random [8], we would expect that fault correction would be much easier than automatic refinement. We can hence speculate that it could scale to real-world software, that because software can be often fixed with only few code changes [18].

4.3 Improving Non-functional Criteria

Optimising non-functional properties of software is an important part of the implementation process. One such property is execution time, and compilers target a reduction in execution time using a variety of optimisation techniques. Compiler optimisation is not always able to produce semantically equivalent alternatives that improve execution times, even if such alternatives are known to exist. Often, this is due to the local nature of such optimisations.

The input to the framework is a program we want to optimise. That program is also used as an oracle for the test cases. The co-evolved test cases are used to test the preservation of the original semantics. The initialisation of the genetic programs is not trivial. On one hand, doing that at random will make the evolution of correct programs very difficult to achieve. On the other hand, using copies might constrain the search in a particular sub-optimal area of the search space. The non-functional criteria needs to be evaluated by executing the programs on a separated set of test cases that does not co-evolve. Because there is more than one objective to optimise, multi-objective algorithms can be used. More details can be found in our preliminary work [3].

4.4 Reverse Engineering

One application of Reverse Engineering [22] is that, given as input the assembler or byte-code of a program, we want to obtain the original source code. We can use our framework to tackle this problem.

The execution of the input assembler can be used as an oracle for the test cases. Initialising the genetic programs at random is possible, but likely it will make the search too difficult. Therefore, smart seeding strategies that exploit the assembler code should be designed.

We have not carried out yet any experiment with our framework on this problem. However, GP often generate code that is difficult to understand by humans. So the readability needs to be carefully taken in account in the search, but that is a common problem in GP that is not specific to only our application. At any rate, there are cases of obfuscating techniques that make difficult to obtain even a compilable source code (e.g.,[13]). In those cases, our technique would be useful even if the evolved code would be difficult to read.

5 Conclusions and Future Work

In this paper we have described the application of a novel co-evolutionary framework to solve some software engineering problems. Co-evolutionary algorithms are not novel, but our contribution is to show how to apply co-evolution of software and test cases in software engineering.

Each software engineering problem has its own specific properties, and care needs to be spent to adapt the framework to solve them. In other words, different types of optimisations of the framework can be done, and each of them is specific to the addressed problem. However, the conceptual co-evolutionary framework would still be the same.

In this paper we briefly discussed the application of our framework to four different software engineering problems: Automatic Refinement, Fault Correction, Improving Non-functional Criteria and Reverse Engineering. Other software engineering problems might be addressed with our framework as well.

We have already obtained preliminary results in some of these tasks, and more specific details can be found in our previous papers [4, 2, 5, 3]. These software engineering tasks are very important, and we believe that our work will help the community to setup the bases from which more research can be built on.

For future work, we want to study in more details some of these problems, like for example Fault Correction. We in fact believe there is large space left for improvements and for designing more sophisticated and tailored algorithms to improve the performance. That would be a compulsory step if we want our technique to scale to real-world software. Other important point to investigate is how to improve the co-evolution in general (e.g., reward diversity in the test cases based on code coverage criteria).

6 Acknowledgements

The author is grateful to Xin Yao, David White and Rami Bahsoon for insightful discussions. This work is supported by EPSRC grant EP/D052785/1.

References

[1] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1338–1349, 2004.

[2] A. Arcuri. On the automation of fixing software bugs. In *In the Doctoral Symposium of the IEEE International Conference on Software Engineering (ICSE)*, pages 1003–1006, 2008.

[3] A. Arcuri, D. R. White, J. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *International Conference on Simulated Evolution And Learning (SEAL)*, pages 61–70, 2008.

[4] A. Arcuri and X. Yao. Coevolving programs and unit tests from their specification. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 397–400, 2007.

[5] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 162–168, 2008.

[6] D. Ashlockand, S. Willson, and N. Leahy. Coevolution and tartarus. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1618–624, 2004.

[7] D. Cliff and G. F. Miller. Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations. In *European Conference on Artificial Life*, pages 200–218, 1995.

[8] R. A. DeMillo, R. J. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[9] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE)*, pages 342–357, 2007.

[10] M. Harman and B. F. Jones. Search-based software engineering. *Journal of Information & Software Technology*, 43(14):833–839, 2001.

[11] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1-3):228–234, 1990.

[12] M. Jazayeri. Formal specification and automatic programming. In *IEEE International Conference on Software Engineering (ICSE)*, pages 293–296, 1976.

[13] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the ACM conference on Computer and Communications Security*, pages 290–299, 2003.

[14] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[15] G. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[16] J. Paredis. Coevolving cellular automata: Be aware of the red queen. In *Proceedings of the International Conference on Genetic Algorithms (ICGA)*, pages 393–400, 1997.

[17] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[18] R. Purushothaman and D. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511– 526, 2005.

[19] C. Rich and R. C. Waters. Automatic programming: myths and prospects. *Computer*, 21(8):40–51, 1988.

[20] A. Ronge and M. G. Nordahl. Genetic programs and co-evolution. developing robust general purpose controllers using local mating in two dimensional populations. In *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation*, pages 81–90, 1996.

[21] J. M. Spivey. *The Z Notation, A Reference Manual. Second Edition*. Prentice Hall, 1992.

[22] P. Tonella, M. Torchiano, B. D. Bois, and T. Systä. Empirical studies in reverse engineering: state of the art and future trends. *Empirical Software Engineering*, 12(5):551–571, 2007.

[23] W. Weimer. Patches as better bug reports. In *International conference on Generative programming and component engineering*, pages 181–190, 2006.

Local Search-Based Refactoring as Graph Transformation

Fawad Qayum, Reiko Heckel
Department of Computer Science
University of Leicester,UK
{fq7,reiko}@mcs.le.ac.uk

Abstract

To improve both performance/scalability and traceability/understandability of search-based refactoring, in this paper, we propose a local formulation of refactoring based on graph transformation. We use graphs to represent software architectures at the class level and graph transformation to formally describe their refactoring operations. This makes it possible to use concepts and techniques from the theory of graph transformation such as unfolding and critical pair analysis to identify dependencies between refactoring steps. As a result, we are able to express the search problem as an instance of the Ant Colony Optimisation metaheuristic.

1 Introduction

Refactoring is considered one of the most suitable techniques to fight the effect of software aging by improving internal software quality characteristics (3) as measured through non-functional requirements like performance, security, maintainability, usability, etc. (11). Classifying refactorings, according to which of these attributes they address and affect, makes us capable of enhancing the quality of a software system by applying the appropriate refactorings in the right places (7). However, due to complex dependencies and conflicts between the individual refactorings it is difficult to decide manually for an optimal sequence of refactoring steps (6), and existing tools offer only limited support for their automated application. Recently, this problem has been addressed using search-based approaches (10). The idea is to consider object oriented design as a combinatorial optimisation problem, attempting to find an optimal design for a given initial model and objective function. Having presented the refactoring task in this way, a variety of search techniques can be applied to solve the optimisation problem (8). One problem with automated transformation in general, and refactoring in particular, is the traceability and understandability of the resulting design. Heavy modifications will make it more difficult to relate to previous designs and developers familiar with these may struggle to understand the new structure. For this reason we would prefer changes to be local, i.e., addressing local problems through localised changes only. Similarly, optimisation algorithms such as the Ant Colony Optimisation metaheuristic (1) can benefit from localised changes by allowing a higher degree of independence between different partial solutions constructed concurrently.

In this paper, we use graphs to represent software architectures at the class level and graph transformation to formally describe their refactoring operations. This makes it possible to use concepts and techniques from the theory of graph transformation, such as unfolding and critical pair analysis, to identify implicit dependencies between refactoring steps. The goal of this work is to use this information in order to improve on search-based approaches to refactoring by expressing the problem as an instance of the Ant Colony Optimisation metaheuristic.

The remainder of this paper is organized as follows: The next section gives background to situate some of the concepts used. Following that, the approach and an example are presented. In the end we describe conclusions and future work.

2 Background

Graphs and Graph Transformation. Graphs are often used as abstract representations of models. For example, in the UML specification (9) a collection of object graphs is defined by means of a metamodel as abstract syntax of UML models. Formally, a *graph* consists of a set of vertices V and a set of edges E such that each edge e in E has a source and a target vertex $s(e)$ and $t(e)$ in V, respectively. Advanced graph models use attributed graphs (5) whose vertices and edges are decorated with textual or numerical information, as well as inheritance between node types. Graphs can occur at two levels: the type level (representing the metamodel) and the instance level (given by all valid object graphs). This concept can be described more generally by the concept of *typed graphs*, where a fixed *type graph* TG serves as abstract representation of the

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.27

metamodel. After having defined the objects of our transformation as instances of type graphs satisfying constraints, model transformations can be specified in terms of graph transformation. A *graph transformation rule* $p : L \rightarrow R$ consists of a pair of TG-typed instance graphs L, R such that the intersection $L \cap R$ is well defined. The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions. Their intersection represents the elements that are required, but not destroyed, by the transformation.

Ant Colony Optimisation. ACO is a metaheuristic widely applicable to any combinatorial optimisation problem for which a constructive solution procedure can be envisaged. To use it, we need to map our problem into a representation allowing artificial ants to construct the solution by traversing a graph. The motivation derives from the foraging real ant's behaviour. This has been recognised as a strategy that can be used to solve complex optimisation problems (1).

3 Approach

We represent the system as a graph and model refactoring as graph transformation. This enables us to analyse the refactoring process in an intuitive (visual) and automated (formal) manner. To state refactoring as an ACO problem, we represent individual refactoring steps as nodes of a graph with edges modelling dependencies between them, such as precedence and conflict. The problem is thus expressed as the search for an optimal path representing the best sequence of refactoring steps applicable to the original system. This "dependency graph" of refactoring steps is obtained through the unfolding (2) of the graph transformation system formed by the rules and the graph representation of the original system. The optimisation depends on an evaluation of paths representing candidate solutions, which takes into account both the cost of the refactoring transformations and the quality of end result.

ACO assumes a problem representation with the following characteristics (4).

1. A finite set of solution components $C = \{c_1, c_2, ...c_{N_c}\}$, where N_c is the number of components, and a set of arcs E connecting the components in C.

2. The states of the problem defined in terms of sequence $x = \langle c_i, c_j, ...\rangle$ over components in C. X is set of all possible states x. The length of a sequence *i.e.*, the number of components in the sequence is denoted by $|x|$.

3. A finite set S of candidate solutions with subset $\bar{S}$ of feasible candidate solutions determined by constraints Ω. *i.e.*, $\bar{S} \subseteq S$.

4. A non- empty subset S^* of optimal solutions.

5. An *evaluation* $f(s)$ associated to each candidate solution. For some problems it is possible to calculate the *partial evaluation* $f_p(x)$ associated with an intermediate state x of the problem.

3.1 Problem Representation

Using the formulation above, we consider a graph defined by associating the set of proposed transformation steps with the set of vertices of the graph. We set the edges of the construction graph to represent dependencies and conflicts between transformations as derived from the unfolding construction. These conflicts include both symmetric ones, requiring mutual exclusion of two refactoring steps, and asymmetric ones, prohibiting two steps to occur in a certain order, but allowing for the reverse.

The pheromone values τ_{ij} and heuristic values η_{ij} are associated with the edges of the graph. The values are determined by partial evaluations $f_p(x)$, associated with incomplete candidate solutions x, which represent preliminary feedback on the success of the search.

3.2 Construction of a Candidate Solutions

The aim is to find an optimal sequence of transformations to be applied to the program source code. In the given problem representation, the artificial ants start from the start vertex of the construction graph with an empty solution, *i.e.* $|x| = 0$. Each ant gradually checks the available components (refactoring steps) in the set C, in order to move in the search space to construct a candidate solution. Initially, the ant will select randomly any applicable solution component from the set C. Then the partial evaluation function $f_p(x)$ associated with partial solutions will guide the selection of the corresponding edge through the pheromone values in the construction graph. The ant will choose the next possible component from the available neighbours in the graph and check its dependencies (i.e., *symmetric and asymmetric conflicts and sequential dependencies*) with the previous solution component in the sequence under construction. In this way the ant increasingly chooses and adds components to the sequence by moving on the construction graph. The evaluation function will calculate the fitness value of each feasible solution (sequence) after applying it on the source graph model. So, the optimal sequence will be selected among the feasible sequences, which leads towards the desired graph having the best fitness value.

4 Example

We will consider the example of a Local Area Network simulation presented by Tom Mens at el. for analysing refactoring steps and dependencies (6). The simplified class

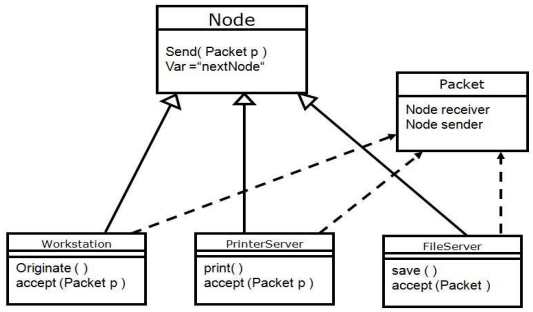


Figure 1. Simplified Class Diagram of LAN Simulation

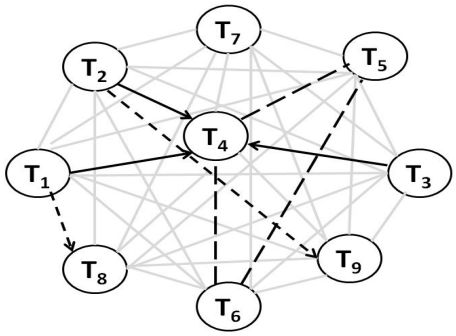
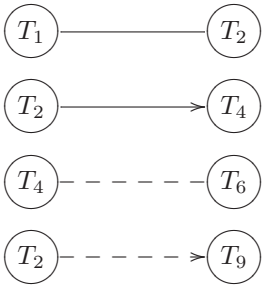


Figure 2. Construction Graph depicts set of proposed refactorings and their dependencies/conflicts between them

hierarchy for this *LAN* simulation is presented in Fig.1. We need to improve the structure of this design by applying refactoring steps from the following set, in order to get the desired model with improved metrics value.**T1, Rename Method print in class PrintServer to process; T2, Rename Method save in class FileServer to process; T3, Create Super class Server for PrintServer and FileServer; T4, Pull Up Method accept from classes PrintServer and FileServer to the super class Server; T5, Move Method accept from class PrintServer to class Packet; T6, Move Method accept from class FileServer to class Packet; T7, Encapsulate Variable receiver in class Packet; T8, Add Parameter p of type Packet to method print in class PrintServer; T9, Add Parameter p of type Packet to method save in class FileServer.**

We need to find a sequence of refactorings from the above set in order to get an improved design. Using the given formulation, we will set the construction graph by representing the set of proposed refactorings as nodes while the edges will correspond to dependencies/conflicts between these refactorings. The construction graph will aids in summarising all potential interactions among these refactorings. We can detect the dependencies/conflicts among the proposed set of refactorings with the help of critical pair analysis technique (statically) or unfolding (dynamically). They are visualise in the construction graph Fig. 2.

As we can see in this construction graph, the edges are distinguished to represent different relations between the nodes (refactorings). Their interpretation is as follows dependencies:



T_1 and T_2 can occur in any order in the sequence.

T_2 must come before T_4 in the sequence.

T_4 and T_6 cannot be part of the same sequence.

if T_2 and T_9 are part of the same sequence, then T_2 cannot occur before T_9.

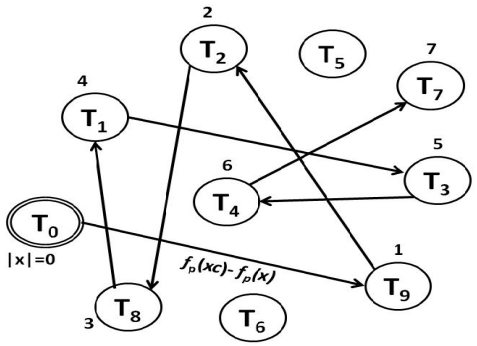


Figure 3. Abstract Diagram of Search Space

Given a problem representation as in Section 3, we will consider the proposed set of refactorings as a set C, so the search space will be defined by associating nodes of the construction graph with the elements of C. Initially, the ant will start from the start vertex with empty solution construction ($|x| = 0$) and select any applicable refactoring from the available neighbours in the search space. Then the ant will gradually look for the next possible move in the search space to construct a candidate solution and check its dependency with previous solution components in the sequence under construction. To guide the behaviour of future ants based on its preliminary success, each ant will assign the improvement in partial evaluation $f_p(xc) - f_p(x)$ to the edge traversed in the adding component c to the path *x*. Suppose we have constructed a candidate solution from the set of solution components $C = \{T_1, T_2, ...T_9\}$, *i.e.* $\mathbf{S} = T_9T_2T_8T_1T_3T_4T_7$ as mentioned in Fig. 3

Similarly we can construct $\mathbf{S}$, the set of all possible candidate solutions (sequences of nodes) until the termination condition is satisfied. After computing all feasible sequences from the search space, the optimal sequence will

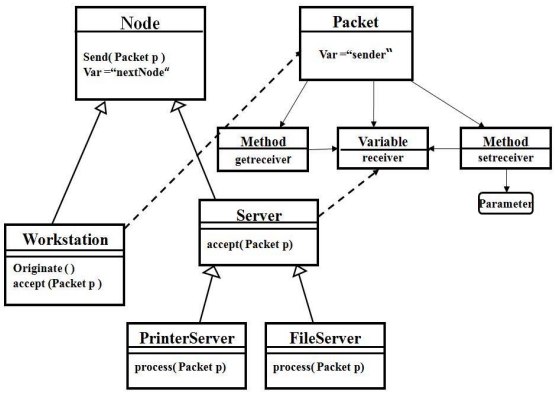


Figure 4. Final Class Diagram of LAN Simulation

be chosen based on the evaluation function, balancing costs against quality as measured by the improved metrics value.

Fig. 4 shows the improved design of our model after applying the sequence of refactorings $T_9T_2T_8T_1T_3T_4T_7$ whose construction is visualised in Fig. 3 above.

5 Conclusion and Future Work

In this paper, we are primarily interested in studying how the formulation of refactoring as graph rewriting can be exploited for optimisation. The discussion of the use of metrics and their relation with the fitness functions and pheromone values is sketchy at best and will be of interest in future research. For example, we would be interested to adopt the approach of PARETO optimality (12) allowing to combine different optimisation criteria into a single optimisation problem

The main goal is to provide automation for the process of selecting refactoring sequences based on quality metrics and formulate this as combinatorial optimisation problem to utilise locality of graph transformation with the help of the ACO metaheuristic. We will continue this work by addressing case studies and develop, implement and evaluate the approach by combining existing graph transformation and refactoring tools with suitable implementations of the optimisation algorithm. Interesting challenges include the study of the locality of metrics, i.e., measurements on graphs which can be taken in a compositional fashion by combining measurements of subgraphs as well as the incremental constructions of unfolding at the time of optimisation.

References

[1] Review of "Ant Colony Optimisation" by M.Dorigo, T. St¨utzle, MIT press, Cambridge, MA, 2004. Artif. Intell., 165(2):261– 264, 2005. Reviewer-Christian Blum.

[2] P. Baldan, A. Corradini, and U. Montanari. Unfolding and event structure semantics for graph grammars. In FoSSaCS '99: Proceedings of the Second International Conference on Foundations of Software Science and Computation Structure. pages 73–89, London, UK, 1999. Springer-Verlag.

[3] B. D. Bois and T. Mens. Describing the impact of refactoring on internal program quality. In Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications, pages 37–48. Vrije Universiteit Brussel, 2003.

[4] M. Dorigo and T. Stutzle. The Ant Colony Optimisation metaheuristic: Algorithms, applications and advances. In Handbook of Metaheuristics, pages 251–285. Kluwer Academic Publishers, 2002.

[5] M. L¨owe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. Term Graph Rewriting: Theory and Practice, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

[6] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. Software and Systems Modeling (SoSyM), pages 269–285, September 2007.

[7] T. Mens and T. Tourwe. A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2):126– 139, 2004.

[8] M. O'Keeffe and M. O. Cinn´eide. Search-based refactoring: an empirical study. Journal of software maintenance and evolution. DOI: 10.1002/smr.378 J. Softw. Maint. Evol., 20(5):345–364, 2008.

[9] OMG. Unified Modeling Language: Superstructure version 2.0. formal/2005-07-04, August 2005.

[10] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In GECCO 2006, pages 1909–1916, New York, NY, USA, 2006. ACM.

[11] R. Tiarks. Quality-driven refactoring. Technical report, University of Bremen, 2005.

[12] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in GECCO '07, New York, NY, USA: ACM Press, 2007, pp. 1106-1113.

A Study of the Multi-Objective Next Release Problem

J. J. Durillo
University of Málaga (Spain)
durillo@lcc.uma.es

Y. Zhang
King's College London (UK)
yuanyuan.zhang@kcl.ac.uk

E. Alba
University of Málaga (Spain)
eat@lcc.uma.es

A. J. Nebro
University of Málaga (Spain)
antonio@lcc.uma.es

Abstract

One of the first issues which has to be taken into account by software companies is to determine what should be included in the next release of their products, in such a way that the highest possible number of customers get satisfied while this entails a minimum cost for the company. This problem is known as the Next Release Problem (NRP). Since minimizing the total cost of including new features into a software package and maximizing the total satisfaction of customers are contradictory objectives, the problem has a multi-objective nature. In this work we study the NRP problem from the multi-objective point of view, paying attention to the quality of the obtained solutions, the number of solutions, the range of solutions covered by these fronts, and the number of optimal solutions obtained. Also, we evaluate the performance of two state-of-the-art multi-objective metaheuristics for solving NRP: NSGA-II and MOCell. The obtained results show that MOCell outperforms NSGA-II in terms of the range of solutions covered, while this latter is able of obtaining better solutions than MOCell in large instances. Furthermore, we have observed that the optimal solutions found are composed of a high percentage of low-cost requirements and, also, the requirements that produce most satisfaction on the customers.

1 Introduction

Nowadays, many software companies are concerned with the development, maintenance, and enhancement of large and complex systems. Typically, customers are interested in buying and using these systems and each of these customers has its own necessities and/or preferences. Thus, one of the first issues which has to be taken into account by software companies is to determine what should be included in the next release of their products, in such a way that the highest possible number of customers get satisfied while this entails a minimum cost for the company. This problem is known as the Next Release Problem (NRP) [1], and it is a widely known in Search Based Software Engineering [8] [9].

In engineering, a requirement is a singular documented need of what a particular product or service should be or do: it is a statement that identifies a necessary attribute, capability, characteristic, or quality of a system in order for it to have value and utility to a user. Thus, NRP consists in selecting a set among all the requirements of a software package such that the cost, in terms of money or resources, of fulfilling these requirements is minimum and the satisfaction of all the users of that system is maximum.

Since satisfying the highest number of customers and spending the lowest amount of money or resources are contradictory objectives, this problem can be treated as a multi-objective optimization problem (MOP) [17]. Contrary to single objective optimization, the solution of a MOP is not a single point, but a set of solutions known as the *Pareto optimal set*. This set is called *Pareto front* when it is plotted in the objective space. In these kinds of problems, there are two major issues to deal with. On the one hand, the solutions contained in the Pareto front should be as close as possible to the optimal Pareto front of the problem. In the context of NRP, this means to obtain solutions which produce the highest possible satisfaction of the customers as well as spending the lowest amount of money. On the other hand, the Pareto front should cover the maximum number of different situations, and provide a well distribution of solutions. In NRP, this means to provide the software engineer who takes the final decision

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.21

with a set of optimal solutions which contains a high number of different configurations.

NRP has been shown to be an instance of the *Knapsack* problem, which is $\mathcal{NP}$-hard [15]. As a consequence, it cannot be solved efficiently by using exact optimization techniques for large problem instances (e.g., an instance with 50 requirements has more than 10^{15} different configurations). In this situation, we need to make use of approximated techniques such as *metaheuristics* [6]. Although this kind of techniques does not ensure to find optimal solutions, they are able to obtain near-optimal solutions in a reasonable amount of time.

Metaheuristics are a family of approximate techniques which have received an increasing attention in the last years. Among them, evolutionary algorithms for solving MOPs are very popular in multi-objective optimization [2] [3], giving raise to a wide variety of algorithms, such as NSGA-II [4], and many others.

Despite that NRP can be considered as a MOP, as of today, most of the related works in the literature have considered it as a single-objective optimization problem [1] [7] [11]. One of the few proposals using a multi-objective formulation of the problem was proposed by Zhang et *al.* [17]. In this work, NSGA-II and other multi-objective algorithms (those using the Pareto dominance concept) were evaluated when solving NRP. This work showed that NSGA-II was able of comparable solutions with those obtained by the previous mono-objective approaches used in the literature, while in addition a set of non-dominated solutions were computed in a single run to help the decision maker at the company. However, this work did not pay special attention to the range of solutions covered by the resulting front, to the number of obtained solutions, nor to which algorithm obtained higher number of high quality solutions. The comparisons between the different evaluated algorithms were done in a visual way and no statistical analysis of the obtained results was provided.

In this work we fill the gap, and we advance in the results obtained in that last work: first, we want to prove that metaheuristics provide an intelligent guidance of the search. In order to achieve this goal, we evaluate NSGA-II and another multi-objective algorithm called MOCell [12]. The latter algorithm has outperformed the former in terms of spread as well as convergence of solutions in several studies in continuous optimization [13] [14]. We take the same instances used in [17] and we compare the obtained results by both techniques with the ones obtained by a random search (RS) as a sanity check. All the comparisons are given on the basis of two quality indicators, and the final number of obtained solutions. The chosen quality indicators are: Hypervolume [19] and Spread [3]. We apply a statistical methodology to ensure the significance of the results. Our second goal is to analyze the differences between the best solutions (those computed by the metaheuristics) and the rest of solutions.

The remainder of this work is structured as follows. The next section contains some background about multi-objective optimization. Section 3 presents the Next Release Problem formally. The algorithms used in this work are described in Section 4. The next sections is devoted to experimentation. We describe the obtained results in section 6, and we analyze them from the point of view of NRP in Section 7. Finally, Section 8 draws the main conclusions and lines of future work.

2 Multi-Objective Background

In this section we include some background on multi-objective optimization. We define the concepts of multi-objective optimization problem (MOP), Pareto optimality, Pareto dominance, Pareto optimal set, and Pareto front. In these definitions we are assuming, without loss of generality, the minimization of all the objectives. A general MOP can be formally defined as follows:

Definition 1 (MOP) *Find a vector* $\vec{x}^* = [x_1^*, x_2^*, \ldots, x_n^*]$ *which satisfies the* m *inequality constraints* $g_i(\vec{x}) \geq 0, i = 1, 2, \ldots, m$, *the* p *equality constraints* $h_i(\vec{x}) = 0, i = 1, 2, \ldots, p$, *and minimizes the vector function* $\vec{f}(\vec{x}) = [f_1(\vec{x}), f_2(\vec{x}), \ldots, f_k(\vec{x})]^T$, *where* $\vec{x} = [x_1, x_2, \ldots, x_n]^T$ *is the vector of decision variables.*

The set of all values satisfying the constraints defines the *feasible region* Ω and any point $\vec{x} \in \Omega$ is a *feasible solution*. As mentioned before, we seek for the *Pareto optima*. Its formal definition is provided below:

Definition 2 (Pareto Optimality) *A point* $\vec{x}^* \in \Omega$ *is Pareto Optimal if for every* $\vec{x} \in \Omega$ *and* $I = \{1, 2, \ldots, k\}$ *either* $\forall_{i \in I}(f_i(\vec{x}) = f_i(\vec{x}^*))$ *or there is at least one* $i \in I$ *such that* $f_i(\vec{x}) > f_i(\vec{x}^*)$.

This definition states that $\vec{x}^*$ is Pareto optimal if no other feasible vector $\vec{x}$ exists which would improve some criteria without causing a simultaneous worsening in at least one other criterion. Other important definitions associated with Pareto optimality are the following ones:

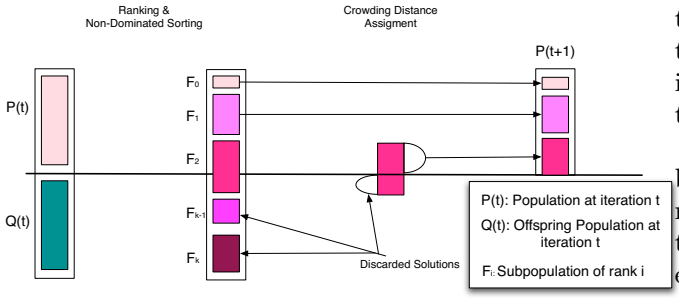


Figure 1. The NSGA-II procedure.

Definition 3 (Pareto Dominance) *A vector* $\vec{u} = (u_1, \ldots, u_k)$ *is said to dominate* $\vec{v}=(v_1, \ldots, v_k)$ *(denoted by* $\vec{u} \preccurlyeq \vec{v}$*) if and only if* $\vec{u}$ *is partially less than* $\vec{v}$*, i.e.,* $\forall i \in \{1, \ldots, k\}$, $u_i \leq v_i \ \wedge \ \exists i \in \{1, \ldots, k\} : \ u_i < v_i$.

Definition 4 (Pareto Optimal Set) *For a given MOP* $\vec{f}(\vec{x})$*, the Pareto optimal set is defined as* $\mathcal{P}^* = \{\vec{x} \in \Omega | \neg \exists \vec{x}' \in \Omega, \vec{f}(\vec{x}') \preccurlyeq \vec{f}(\vec{x})\}$.

Definition 5 (Pareto Front) *For a given MOP* $\vec{f}(\vec{x})$ *and its Pareto optimal set* $\mathcal{P}^*$*, the Pareto front is defined as* $\mathcal{PF}^* = \{\vec{f}(\vec{x}), \vec{x} \in \mathcal{P}^*\}$.

Obtaining the Pareto front of a MOP is the main goal of multi-objective optimization. In theory, a Pareto front could contain a large number of points (even infinite). In practice, a good solution will only contain a limited number of them; thus, an important goal is that they should be as close as possible to the exact Pareto front, as well as they should be uniformly spread. Otherwise, they would not be very useful to the decision maker: closeness to the Pareto front ensures that we are dealing with optimal solutions, while a uniform spread of the solutions means that we have made a good exploration of the search space and no regions are left unexplored.

3 Problem Statement

This section is aimed at describing the mathematical model of NRP.

Given a software package, there is a set, C, of m customers whose requirements have to be considered in the development of the next release of this system. Each customer has associated a value, c_i, which reflects its degree of importance as a customer for the software development company.

There is also a set, R, of n requirements to complete. In order to meet each requirement it is needed to expend a certain amount of resources, which can be transformed into an economical cost: the cost of satisfying the requirement. We denote as r_j, $(1 \leq j \leq n)$ the economical cost of achieving the requirement j.

We also assume that more than one customer can be concerned with any requirement, and that all the requirements are not equally important for all the customers. In this way, associated with each customer and each requirement, there is a value v_{ij}, which represents the importance of the requirement j for the customer i. All these values can be represented by a matrix. Associated with the set R, there is a directed acyclic gaph $G = (R, E)$, where $(r_i, r_k) \in E$ if and only if $r_i \in R$ is a prerequisite of $r_k \in R$ (i.e., it is mandatory to fulfill r_i before to r_k).

G is also transitive; then, if $(r_k, r_j) \in E$, and $(r_j, r_i) \in E$, the requirement r_k must be also fulfilled in order to afford r_i. In the special case where no requirement has any prerequisite, $E = \emptyset$, we say that the problem is basic.

The problem faced by the software company is to find a subset, R', of requirements which minimizes the cost and maximizes the total satisfaction of the customers with the finally included requirements. Thus the multi-objective Next Release Problem can be formalized as

$$minimize \ f_1 = \sum_{r_i \in R'} r_i \tag{1}$$

and,

$$maximize \ f_2 = \sum_{i=1}^{n} c_i \sum_{r_j \in R'} v_{ij}. \tag{2}$$

Since minimizing a given function f is the same as maximizing (-f), in this work we have considered the maximization of (-f1) (i.e., the economical cost for companies), and f2 (i.e, the customer satisfaction).

4 Solver Algorithms

In this section we describe the three algorithms which will be evaluated for solving NRP.

4.1 NSGA-II

NSGA-II, proposed by Deb *et al.* [4], is the reference algorithm in multi-objective optimization. It is based on a ranking procedure, consisting in extracting the non-dominated solutions from a population and assigning them a rank of 1. These solutions are removed from this population; the next group of non-dominated solutions have a rank of 2, and so on.

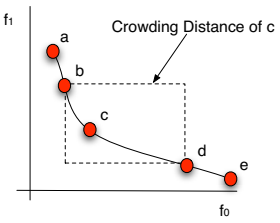


Figure 2. Measuring the crowding distance of a non-dominated point.

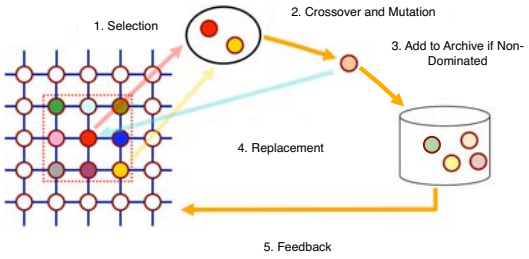


Figure 3. The MOCell procedure.

The algorithm has a current population that is used to create an auxiliary one (the offspring population); after that, both populations are combined to obtain the new current population. The procedure is as follows: the two populations are sorted according to their rank, and the best solutions are chosen to create the new population. In the case of having to select some individuals with the same rank, a density estimation based on measuring the crowding distance (see Figure 4) to the surrounding individuals belonging to the same rank is used to get the most promising solutions. Typically, both the current and the auxiliary populations have equal size. The procedure of NSGA-II is depicted in Figure 1.

4.2 MOCell

MOCell (Multi-Objective Cellular Genetic Algorithm), introduced by Nebro et *al.* [14], is a cellular genetic algorithm (cGA). In cGAs, the concept of (small) *neighborhood* is intensively used; this means that an individual may only cooperate with its nearby neighbors (see Figure 3) in the breeding loop. The overlapped small neighborhoods of cGAs help in exploring the search space because the induced slow diffusion of solutions through the population, providing a kind of exploration (diversification). Exploitation (intensification) takes place inside each neighborhood by genetic operations. MOCell also includes an external archive to store the non-dominated solutions found so far. This archive is limited in size and uses the crowding distance of NSGA-II to keep diversity in the Pareto Front. We have used here an asynchronous version of MOCell, called aMOCell4 [13].

4.3 Random Search

We also apply a random search (RS) to NRP. This is merely a 'sanity check'; all metaheuristic algorithms should be capable of comfortably outperforming random search for a well-formulated optimization problem. It consist in generating solutions randomly. The final results is the set of all the non-dominated solutions found.

5 Experimentation

This section is aimed at presenting the indicators used to measure the quality of the obtained results and the benchmark problems we have used. It also describes how the solutions of the problem have been encoded as well as the genetic operators employed, the configuration of the algorithms, and the methodology we have followed.

5.1 Quality Indicators

For assessing the quality of the results obtained by the algorithms, two different issues are normally taken into account: (i) to minimize the distance of the Pareto front generated by the proposed algorithm to the optimal Pareto front (convergence), and (ii) to maximize the spread of solutions found, so that we can have a distribution of vectors as smooth and uniform as possible (diversity). To determine these issues it is usually necessary to know the exact location of the optimal Pareto front. In the case of NRP, we are dealing with a problem whose optimal Pareto front is unknown. Thus, we have employed as Pareto optimal front the one composed of the non-dominated solutions out of all the executions carried out (i.e., the best front known for these problems until now).

A number of quality indicators have been proposed to be used for comparative studies among metaheuristics when solving MOPs. We use in this work one indicator which measures the diversity of the solution, and other one which measures both convergence and diversity.

- **Spread (Δ).** The *Spread* indicator [3] is a diversity quality indicator that measures the extent

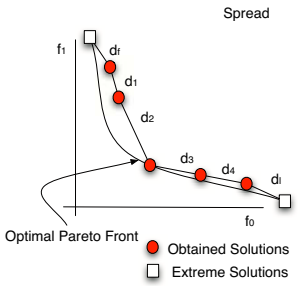


Figure 4. Distances from the extreme solutions.

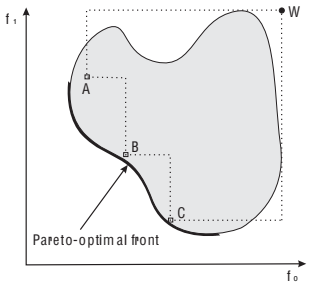


Figure 5. The hypervolume enclosed by the non-dominated solutions.

of spread by the set of computed solutions. Δ is defined as:

$$\Delta = \frac{d_f + d_l + \sum_{i=1}^{N-1} |d_i - \bar{d}|}{d_f + d_l + (N-1)\bar{d}} \ , \tag{3}$$

where d_i is the Euclidean distance between consecutive solutions, $\bar{d}$ is the mean of these distances, and d_f and d_l are the Euclidean distances to the *extreme* solutions of the optimal Pareto front in the objective space (see Figure 4 for further details). This metric takes a zero value for an ideal distribution, pointing out a perfect spread of the solutions in the Pareto front. We apply this metric after a normalization of the objective function values to the range [0..1]. Pareto fronts with a smaller Δ value are desired.

- **Hypervolume (HV).** This indicator calculates the volume (in the objective space) covered by members of a non-dominated set of solutions Q (the region enclosed into the discontinuous line in Figure 5, $Q = \{A, B, C\}$) for problems where all objectives are to be minimized [19]. Mathematically, for each solution $i \in Q$, a hypercube v_i is constructed with a reference point W and the solution i as the diagonal corners of the hypercube. The reference point can simply be found by constructing a vector of worst objective function values. Thereafter, a union of all hypercubes is found and its hypervolume (HV) is calculated:

$$HV = \text{volume}\left(\bigcup_{i=1}^{|Q|} v_i\right). \tag{4}$$

In this case, we also apply this metric after a normalization of the objective function values to the range [0..1]. A Pareto front with a higher HV than another one could be due to two things: some solutions in the former front dominate solutions in the second one, or, solutions in the first front are better distributed than in the second one. Thus, algorithms with larger values of HV are desirable.

Another important topic when solving real problems is related to the number of obtained solutions. To this point, we also consider here the number of obtained solutions by the evaluated algorithms as well as the number of them which are Pareto optimal with regard to all the computed solutions.

5.2 Test Problems

The three algorithms were applied to two test problem sets, both of them composed of three problems, for two separate empirical study cases: Empirical Study 1 (ES1) and Empirical Study 2 (ES2). In the ES1 we report results concerning the performance of the three algorithms for what might be considered 'typical' cases, with the number of customers ranging from 15 to 100 and the number of requirements ranging from 40 to 140. In ES2, we are concerned with bounding NRP. This latter benchmark is composed of three instances: an instance with 20 requirements and 100 customers (few requirements, high number of customers), an instance with 25 requirements and 100 customers, and other with only 2 customers and 200 requirements (high number of requirements, few customers). All the instances have the nomenclature c_r, where c is the number of customers, and r the maximum number of requirements. We have not considered dependences among requirements [17].

5.3 Solution Encoding and Genetic Operators

As described in Section 3, a solution to the problem consists in selecting a subset of requirements to be included in the next release of the software package. In this work, each solution is encoded as a binary string, s, of length n (the maximum number of requirements), where $s_i = 1$ means that the requirement i is included in the next release of the software.

As to the genetic operators, we have used *binary tournament* as the selection scheme. This operator works by randomly choosing two individuals from the

population and the one dominating the other is selected; if both solutions are non-dominated one of them is selected randomly. We also use *single point crossover* as crossover operator. It works by creating a new solution in which the binary string from the beginning of a parent solution to a crossover point, previously chosen, is copied from that parent while the rest is copied from other parent. Finally, the mutation operator used is *random mutation* in which some random bits of the string are flipped.

5.4 Configuration

All approaches were run for a maximum of 10,000 function evaluations. The initial population was set to 100 in NSGA-II and MOCell. In MOCell, the archive size is also limited to 100 solutions. In both algorithms the probability of the crossover operator was set to $P_c = 0.9$ and the probability of the mutation operator to $P_m = 1/n$, being n the number or requirements.

All the algorithms have been implemented using jMetal [5], a Java framework aimed at the development, experimentation, and study of metaheuristics for solving multi-objective optimization problems.

5.5 Methodology

To assess the search capabilities of the algorithms, we have performed 100 independent runs of each experiment, and we show the mean, $\bar{x}$, and standard deviation, σ_n, as measures of location (or central tendency) and statistical dispersion, respectively. Since we are dealing with stochastic algorithms in order to provide confident results, we have also included a testing phase by performing a multiple comparison of samples [10]. We have used the Wilcoxon test for that purpose. We always consider a confidence level of 95% (i.e., significance level of 5% or p-value below 0.05) in the statistical tests.

Table 1. Results of the Δ quality Indicator

Problem	NSGA-II $\bar{x}_{\sigma_n}$	MOCell $\bar{x}_{\sigma_n}$	RS $\bar{x}_{\sigma_n}$
15_40	4.93e-1$_{\pm 4.2e-2}$	3.76e-1$_{\pm 3.3e-2}$	6.91e-1$_{\pm 3.8e-2}$
50_80	5.00e-1$_{\pm 3.5e-2}$	4.10e-1$_{\pm 3.9e-2}$	7.72e-1$_{\pm 2.9e-2}$
100_140	5.51e-1$_{\pm 3.7e-2}$	4.85e-1$_{\pm 3.8e-2}$	8.08e-1$_{\pm 2.4e-2}$
100_20	7.98e-1$_{\pm 9.1e-3}$	6.16e-1$_{\pm 4.9e-3}$	6.09e-1$_{\pm 5.3e-2}$
100_25	5.79e-1$_{\pm 2.4e-2}$	5.43e-1$_{\pm 2.6e-2}$	6.45e-1$_{\pm 4.2e-2}$
2_200	6.06e-1$_{\pm 3.6e-2}$	5.60e-1$_{\pm 4.7e-2}$	8.11e-1$_{\pm 3.2e-2}$

6 Obtained Results

In this section we present the obtained results by the three evaluated algorithms. We start by describing the values obtained by the two quality indicators used. Then, we pay attention to the number of obtained solutions, and how many of these solutions are among the better solutions found so far.

Table 2. Results of the HV quality Indicator

Problem	NSGA-II $\bar{x}_{\sigma_n}$	MOCell $\bar{x}_{\sigma_n}$	RS $\bar{x}_{\sigma_n}$
15_40	6.63e-1$_{\pm 1.6e-3}$	6.63e-1$_{\pm 1.2e-3}$	5.27e-1$_{\pm 7.4e-3}$
50_80	5.88e-1$_{\pm 4.6e-3}$	5.85e-1$_{\pm 4.9e-3}$	4.35e-1$_{\pm 7.0e-3}$
100_140	5.28e-1$_{\pm 6.4e-3}$	5.20e-1$_{\pm 5.7e-3}$	3.65e-1$_{\pm 5.4e-3}$
100_20	6.13e-1$_{\pm 2.1e-4}$	6.13e-1$_{\pm 3.0e-4}$	5.72e-1$_{\pm 4.5e-3}$
100_25	6.35e-1$_{\pm 6.3e-4}$	6.35e-1$_{\pm 5.7e-4}$	5.45e-1$_{\pm 6.2e-3}$
2_200	5.26e-1$_{\pm 7.6e-3}$	5.17e-1$_{\pm 7.5e-3}$	3.02e-1$_{\pm 5.0e-3}$

The obtained values for Δ, HV, number of solutions, and number of Pareto optimal solutions among all the solutions found are shown in tables 1, 2, 4, and 5, respectively. In the case of the Δ indicator, the lower the value the better, while in the rest of indicators used, the higher the value the better.

We start by analyzing the Δ quality indicator (Table 1). If we focus on the instances belonging to the first problem set, ES1 (i.e., 15_40, 50_80, and 100_140), we see that MOCell is the algorithm which has obtained the better values of the indicator. As to the ES2 benchmark (i.e., 100_20, 100_25, and 2_200), the results observed in ES1 hold the same for the instances with more than 20 requirements. As a general conclusion from the obtained values by this indicator, MOCell has performed better than the other techniques.

In most of the instances the results obtained by the RS are far from those obtained by MOCell and NSGA-II. However, it is remarkable the obtained results by RS in the instance 100_20 (the one with the smallest number of requirements).

Proceeding as before, we analyze the results obtained for the HV quality indicator, whose values are included in Table 2. Considering the instances belonging to ES1, the obtained results by MOCell and NSGA-II has been quite similar in the instances with 40 and 80 requirements. However, in the instance with the highest number of requirements, NSGA-II has outperformed MOCell. Concerning ES2, MOCell and NSGA-

Table 3. Comparison NSGA-II vs MOCell: Δ and HV Indicators

Problem	NSGA-II vs MOCell	
	Δ Indicator	HV Indicator
15_40	MOCell	•
50_80	MOCell	NSGA-II
100_140	MOCell	NSGA-II
100_20	MOCell	•
100_25	MOCell	•
2_200	MOCell	NSGA-II

II have obtained similar results in the instances with 20 and 25 requirements, whereas in the instance with the highest number of requirements (200) NSGA-II has performed better than MOCell.

For this indicator, all the results obtained by RS has been significantly worse than the obtained by MOCell and NSGA-II.

In Table 3, we summarize the comparison between MOCell and NSGA-II, considering Δ and HV. The second and third column show the name of the algorithm which has been significantly better in each indicator, respectively; a "•" symbol means that no statistical differences have been found between them. As general conclusion for the Δ indicator, we have that MOCell has outperformed the results obtained by NSGA-II in all the instances. As to the HV indicator, both algorithms, MOCell and NSGA-II, have behaved in a similar way in the instances with the smaller number of requirements (until 40 requirements); however, as the number of requirements increases NSGA-II has obtained better figures than MOCell.

Regarding to the number of obtained solutions (Table 4), we observe that NSGA-II, and MOCell have performed similarly: only in the instance with the smallest number of requirements of the ES2 benchmark, NSGA-II has obtained more solutions than MOCell clearly. As to how many of the obtained solutions are Pareto optimal among all those obtained (Table 5), NSGA-II has outperformed MOCell in all the cases. Thus, focusing in NSGA-II, we observe that in the instances with 20, 25, and 40 requirements, it has obtained from 42 to 20 Pareto optimal solutions, respectively. In the instances with 80 and 140 requirements, it has only found less than five Pareto optimal solutions, and, in the instance with the 200 requirements and only 2 customers, it has obtained around 50 Pareto optimal solutions per execution.

Finally, we see that RS has obtained a fewer number of solutions than NSGA-II and MOCell in practically all the instances. Nevertheless, it is remarkable that RS has found some optimal solutions in the instance with the highest number of requirements.

In Table 6 we summarize the comparison between NSGA-II and MOCell according to the number of found solutions. As commented above, the algorithm which obtains a higher number of solutions and Pareto optimal solutions is NSGA-II. In this case, all the results but two are given with statical confidence.

An example of the obtained fronts in the different instances is depicted in Figure 6. In this figure "Customer Satisfaction" means the total satisfaction of the customers, and "-cost" represents the economical cost for companies. In all the cases the metaheuristic techniques have outperformed RS. In the instance with the smaller number of requirements (Figure 6.b), we see that the results of RS are closer to the results of the other techniques than in the other instances. It is also worth mentioning how MOCell is able to obtain a front which covers a high range of different solutions; however, NSGA-II have obtained solutions which dominate those obtained by the other techniques, specially in the instances with the higher number of requirements.

The advantages of using metaheuristic techniques can be easily drawn from Figure 6. Comparing the obtained results by these techniques against RS, we observe the benefits of using them for a software company. On the one hand, the cost is reduced; NSGA-II as well as MOCell have obtained solutions with less economical cost and also higher satisfaction of their customers. On the other hand, multi-objective metaheuristics provide the software engineer with a set of

Table 4. Results of the Number of Obtained Solutions

Problem	NSGA-II $\bar{x}_{\sigma_n}$	MOCell $\bar{x}_{\sigma_n}$	RS $\bar{x}_{\sigma_n}$
15_40	$1.00e+2_{\pm 0.0e+0}$	$9.99e+1_{\pm 3.4e-1}$	$3.34e+1_{\pm 3.7e+0}$
50_80	$1.00e+2_{\pm 0.0e+0}$	$1.00e+2_{\pm 1.4e-1}$	$4.08e+1_{\pm 4.5e+0}$
100_140	$1.00e+2_{\pm 0.0e+0}$	$9.99e+1_{\pm 3.4e-1}$	$4.12e+1_{\pm 5.5e+0}$
100_20	$1.00e+2_{\pm 0.0e+0}$	$7.71e+1_{\pm 1.0e+0}$	$3.78e+1_{\pm 4.2e+0}$
100_25	$1.00e+2_{\pm 0.0e+0}$	$1.00e+2_{\pm 2.2e-1}$	$3.61e+1_{\pm 3.9e+0}$
2_200	$1.00e+2_{\pm 0.0e+0}$	$9.93e+1_{\pm 1.8e+0}$	$2.89e-1_{\pm 4.2e+0}$

Table 5. Results of the Number of Solutions Contained in the Best Front Found

Problem	NSGA-II $\bar{x}_{\sigma_n}$	MOCell $\bar{x}_{\sigma_n}$	RS $\bar{x}_{\sigma_n}$
15_40	$1.98e+1_{\pm 4.4e+0}$	$1.39e+1_{\pm 4.5e+0}$	$3.40e-1_{\pm 8.1e-1}$
50_80	$2.11e+0_{\pm 2.9e+0}$	$8.00e-2_{\pm 3.1e-1}$	$0.0e+0_{\pm 0.0e+0}$
100_140	$1.19e+0_{\pm 2.5e+0}$	$2.00e-2_{\pm 1.4e-1}$	$0.0e+0_{\pm 0.0e+0}$
100_20	$4.23e+1_{\pm 1.6e+0}$	$3.03e+1_{\pm 9.6e-1}$	$9.20e-1_{\pm 1.1e+0}$
100_25	$3.77e+1_{\pm 3.7e+0}$	$3.32e+1_{\pm 4.0e+0}$	$3.00e-2_{\pm 1.7e-1}$
2_200	$4.90e+1_{\pm 2.8e+0}$	$0.0e+0_{\pm 0.0e+0}$	$1.30e+1_{\pm 5.4e-1}$

Table 6. Comparison NSGA-II vs MOCell: Number of Solutions and Pareto Optimal Solutions

	NSGA-II vs MOCell	
Problem	Solutions	Pareto Optimal Solutions
15_40	NSGA-II	NSGA-II
50_80	•	NSGA-II
100_140	NSGA-II	NSGA-II
100_20	NSGA-II	NSGA-II
100_25	•	NSGA-II
2_200	NSGA-II	NSGA-II

trade off optimal configurations instead a single one allowing him (or her) to choose the most appropriated solution in each different situation.

7. Analysis

Previous section has shown that multi-objective optimization algorithms are suitable techniques for solving NRP; in this section, we are interested in analyzing the obtained solutions from the point of view of the problem.

Let us suppose that a software engineer has to select a subset of requirements to including in the next release of a software package and he only wants to optimize the cost of fulfilling these requirements; in this situation, it is clear that he should include, in this set, those requirements which are cheaper of implementing than others. Something similar happens if he only wants to maximize the satisfaction of the users of that system: he should consider those requirements which satisfy the customer the most. But, what requirements should he consider if he wants to optimize both objectives?

We have analyzed which requirements are included in each solution. Thus, for each point in a front of solutions, we have computed the percentage of included requirements which are cheaper or equal expensive of implementing than all the requirements not included by this point. Similarly, we have also calculated the percentage of requirements included which satisfy the customers more or equal than all the requirements not included by this point. Let us call these percentages as P_1 and P_2, respectively. They can be formalized as

$$P_1 = \frac{\| r_i \in R', r_i \leq r_j, \forall r_j \in (R - R') \|}{\|R'\|} \quad (5)$$

and,

$$P_2 = \frac{\| r_i \in R', \sum_{j=1}^{n} c_j * v_{ji} \geq \sum_{j=1}^{n} c_j * v_{jk}, \forall r_k \in (R - R') \|}{\|R'\|} \quad (6)$$

.

The mean and standard deviation of P_1 and P_2 in all the solutions in all the executions carried out are included in Tables 7 and 8, respectively. Regarding to Table 7, we observe that the computed solutions by the metaheuristic techniques, NSGA-II and MOCell, are composed of a higher percentage of cheapest requirements than the solutions computed by RS. For example, in the instance 15_40 the percentage of cheapest requirements in the solutions found by NSGA-II and MOCell are 78.9% and 77.2%, respectively, while in the solutions found by RS this percentage has only been of 23.4%. The same holds with the percentage of requirements which most satisfy the customers (Table 8): this percentage is higher in the solutions found by the metaheuristic techniques than in the solutions found by RS. Thus, the better solutions are composed of a high number of low cost requirements, and also those requirements which most satisfy the customers.

Table 7. Percentage of Included Requirements with Cost Fewer of Equal that all the Requirements not Included

Problem	NSGA-II $\bar{x}_{\sigma_n}$	MOCell $\bar{x}_{\sigma_n}$	RS $\bar{x}_{\sigma_n}$
15_40	$7.89e{+}1_{\pm 1.8e+0}$	$7.72e{+}1_{\pm 2.1e+0}$	$2.34e{+}1_{\pm 1.8e+0}$
50_80	$7.22e{+}1_{\pm 3.9e+0}$	$6.55e{+}1_{\pm 5.3e+0}$	$1.18e{+}1_{\pm 1.5e+0}$
100_140	$5.48e{+}1_{\pm 6.1e+0}$	$4.45e{+}1_{\pm 6.5e+0}$	$7.85e{+}0_{\pm 7.6e-1}$
100_20	$8.18e{+}1_{\pm 6.0e-1}$	$8.07e{+}1_{\pm 5.0e-1}$	$5.35e{+}1_{\pm 4.1e+0}$
100_25	$8.15e{+}1_{\pm 1.3e+0}$	$8.10e{+}1_{\pm 1.4e+0}$	$3.40e{+}1_{\pm 2.9e+0}$
2_200	$2.04e{+}1_{\pm 4.3e+0}$	$1.87e{+}1_{\pm 3.4e+0}$	$1.12e{+}1_{\pm 3.7e-1}$

Table 8. Percentage of Included Requirements with Customer Satisfaction Bigger of Equal that all the Requirements not Included

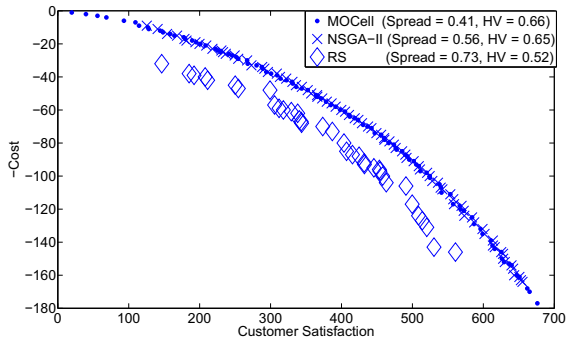
Problem	NSGA-II $\bar{x}_{\sigma_n}$	MOCell $\bar{x}_{\sigma_n}$	RS $\bar{x}_{\sigma_n}$
15_40	$4.80e{+}1_{\pm 2.6e+0}$	$4.50e{+}1_{\pm 2.6e+0}$	$1.27e{+}1_{\pm 2.0e+0}$
50_80	$1.54e{+}1_{\pm 3.1e+0}$	$1.39e{+}1_{\pm 3.1e+0}$	$4.51e{+}0_{\pm 6.5e-1}$
100_140	$3.42e{+}0_{\pm 1.4e+0}$	$3.42e{+}0_{\pm 1.9e+0}$	$1.44e{+}0_{\pm 3.0e-1}$
100_20	$4.85e{+}1_{\pm 8.4e-1}$	$4.79e{+}1_{\pm 6.1e-1}$	$2.99e{+}1_{\pm 2.8e+0}$
100_25	$3.41e{+}1_{\pm 1.9e+0}$	$3.28e{+}1_{\pm 1.7e+0}$	$1.64e{+}1_{\pm 1.9e+0}$
2_200	$1.19e{+}1_{\pm 3.4e+0}$	$1.05e{+}1_{\pm 3.3e+0}$	$2.49e{+}0_{\pm 4.3e-1}$

8 Conclusions and Future Work

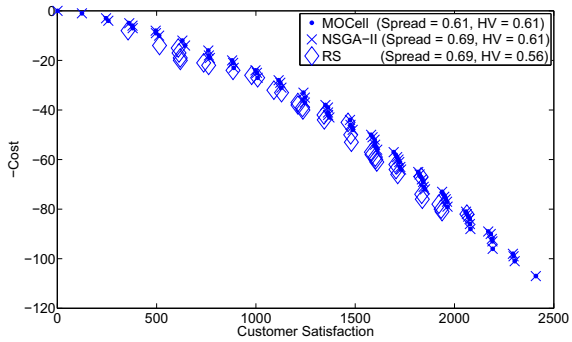
In this paper we have studied the Next Release Problem, with the intention of analyzing the performance of different multi-objective algorithms, and the solutions they have provide. We have evaluated two state-of-the-art multi-objective optimization algorithms and compared them against a random search. This comparison has been done in the basis of two quality indicators, HV and Δ, and the number of solutions obtained by those algorithms.

In the case of the Δ indicator, which measures how uniform is the distribution of computed solutions, MOCell is the algorithm which has worked out the better results in instances with more than 20 requirements. The random search algorithm has obtained the worst results in practically all the instances.

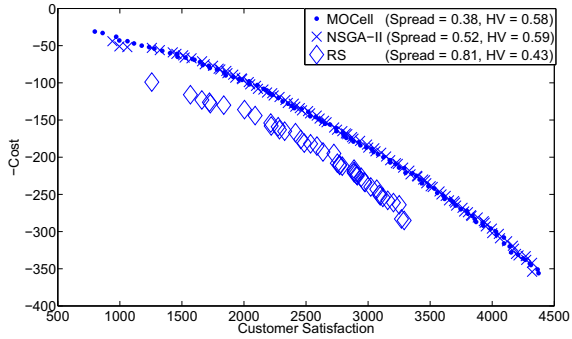
Regarding to the HV indicator, which measures the convergence and distribution of the fronts obtained by the algorithms, NSGA-II and MOCell have both shown a similar performance in instances with a small and



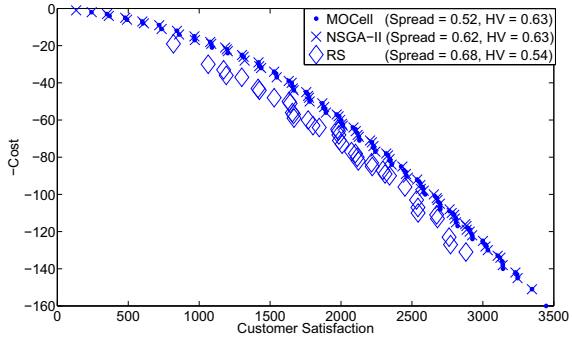
(**a**) 15 customers; 40 requirements



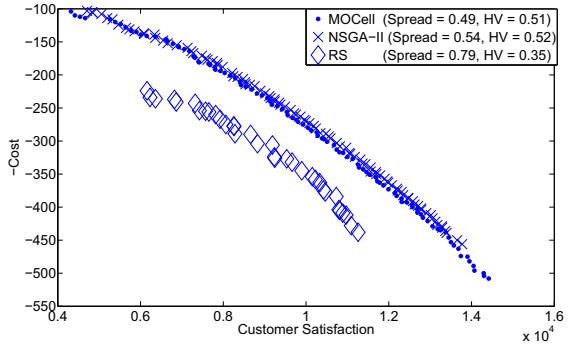
(**d**) 100 customers; 20 requirements



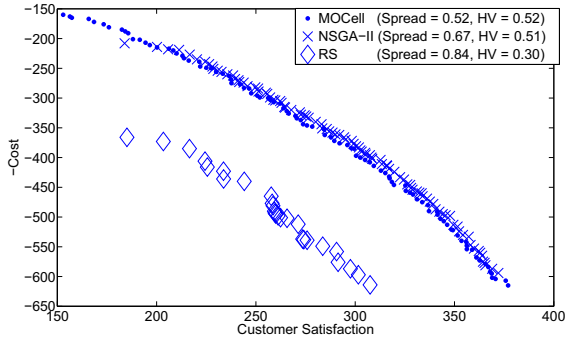
(**b**) 50 customers; 80 requirements



(**e**) 100 customers; 25 requirements



(**c**) 100 customers; 140 requirements



(**f**) 2 customers; 200 requirements

Figure 6. Examples of the Obtained Front of Solutions in each Instance.

medium number of requirements, i.e., from 20 to 40 requirements. As the number of requirements increases, NSGA-II provides better results than MOCell. The random search has obtained the worst results in all the instances.

As to the number of obtained solutions, NSGA-II is also the algorithm which has shown the best performance, and it is also the technique which computes the highest number of solutions which are contained in the best front known for each instance.

Concerning the solutions of the problem, we have observe that the best solutions are composed of a high percentage of low-cost requirements, and also, of requirements which most satisfy the customers.

The future work will verify these findings by applying search techniques to real world problems. This will provide valuable feedback to researchers and practitioners in search techniques and in software engineering communities. Other reformulations of the problem considering different sets of objectives and constraints including dependency relationship between requirements will be experimented with. This in turn may give rise

to the need for the development of more efficient solution techniques. It is also interesting to investigate how these techniques scale when the number of requirements and/or customer increases. In order to come to this goal, it is also needed to develop a procedure which allows the systematic creation of instances with the desired features; in this sense, we plan to design a problem generator for NRP instances.

Acknowledgements

J. J. Durillo, A. Nebro, and E. Alba acknowledge founds from the "Consejería de Innovación, Ciencia y Empresa", Junta de Andalucía under contract P07-TIC-03044 DIRICOM project (`http://diricom.lcc.uma.es`), and the Spanish Ministry of Science and Innovation and FEDER under contract TIN2008-06491-C04-01 (the M* project). J. J. Durillo is also supported by grant AP-2006-03349 from the Spanish Ministry of Education and Science.

References

[1] Bagnall, A., Rayward-Smith, V., and Whittley, I. The next release problem. *Information and Software Technology 43*, 14 (Dec. 2001), 883–890.

[2] Coello Coello, C. A., Van Veldhuizen, D. A., and Lamont, G. B. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic Publishers, New York, May 2002.

[3] Deb, K. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, Chichester, UK, 2001.

[4] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA–II. *IEEE TEC 6*, 2 (Apr. 2002), 182–197.

[5] Durillo, J.J., Nebro, A.J., Luna, F., Dorronsoro, B., and Alba, E. jMetal: A Java Framework for Developing Multi-Objective Optimization Metaheuristics. Tech. Rep. ITI-2006-10, Dept. de Lenguajes y Ciencias de la Computación, University of Málaga.

[6] Glover, F. W., and Kochenberger,G. A. Handbook of Metaheuristics. Kluwer, 2003.

[7] Greer, D., and Ruhe, G. Software release planning: an evolutionary and iterative approach. *Information & Software Technology 46*, 4 (2004), 243–253.

[8] Harman, M. The current state and future of Search Based Software Engineering. In FOSE 2007.

[9] Harman, M., and Jones, F. B. Search Based Software Engineering. *Information and Software Technology 43*, 14 (2001), 833-839

[10] Hochberg, Y., and Tamhane, A. C. Multiple Comparison Procedures. Wiley, 1987.

[11] Karlsson, J., Wohlin, C., and Regnell, B. An evaluation of methods for priorizing software requirements. *Information and Software Technology 39*, (1998), 939–947.

[12] Nebro, A. J., Durillo, J. J., Luna, F., Dorronsoro, B., and Alba, E.. A Cellular Genetic Algorithm for Multiobjective Optimization. Proceedings of NICSO 2006.

[13] Nebro, A. J., Durillo, J. J., Luna F, Dorronsoro, B, and Alba, E. Design issues in a multi-objective cellular genetic algorithm. EMO 2007.

[14] Nebro, A. J., Durillo, J. J., Luna, F., and Dorronsoro, B., and Alba, E. MOCell: A Cellular Genetic Algorithm for Multiobjective Optimization. International Journal of Intelligent Systems (to appear), 2008.

[15] Papadimitriou, C. H., and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity*. Dover, 1998.

[16] Szidarovsky, F., Gershon, M. E., and Dukstein, L. *Techniques for multi-objective decision making in systems management*. Elsevier, New York, 1986.

[17] Zhang, Y., Harman, M., and Mansouri, A. S. The Multi-Objective Next Release Problem. In GECCO 2007.

[18] Zitzler, E., Laumanns, M., Thiele, L. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Tech. Rep. 103, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH), Zurich, Switzerland.

[19] Zitzler, E., Thiele, L. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. IEEE TEC 3(4):257–271.

Dynamic Architectural Selection: A Genetic Algorithm Based Approach*

Dongsun Kim and Sooyong Park
Department of Computer Science and Engineering, Sogang University
1 Shinsoo-dong, Mapo-gu, Seoul, 121-742, Republic of Korea
{darkrsw,sypark}@sogang.ac.kr

Abstract

As the software industry is focusing on dealing with various requirements and environments, such as mobile and ubiquitous environments, software systems are increasingly undergoing many situational changes. These changes influence the quality of services that the software provides. Therefore, to maintain the performance of the software, it must be reconfigured. The reconfiguration is a complex problem if an application faces a large number of situations and has a number of software architectural instances. In this paper, we propose a novel approach to autonomous architectural selection in response to the current situation of various environments. This approach enables a software system to determine the best architectural instance for the current situation. To quickly find the best instance, we apply a genetic algorithm to the selection process. Further, we provide a performance evaluation to demonstrate that our approach efficiently find the best instance (or considerably good instance).

1. Introduction

User preferences, which represent the quality attributes that the user desires and their priority of quality attributes, differ for each user and change depending on each instance of usage during execution. Therefore, an application must contain different functions to suit different users' requirements; for example, if user A has more concerns about security, then the application should sacrifice usability and add more security functions to achieve a more secure execution, while user B desires more faster execution and the application must sacrifice usability and durability by removing rich user interfaces and so on.

In addition to the user requirements, the changing environments in which applications perform their tasks complicate the situation. For example, situation variance such as position, noise, light, battery level, and network bandwidth can influence mobile software applications. These factors can change the quality of applications. Therefore, the applications must modify their functionality in response to situational changes.

The goal of this study is to provide a method that autonomously selects an appropriate software architectural instance from the large set of candidates in response to situational and requirement changes at runtime. To describe this problem, we provide a motivating example that illustrates how the software architecture must be changed when situations and requirements change. Then, we precisely formulate the architectural selection problem using softgoal interdependency graphs [2]. To deal with this problem, this paper provides a novel approach to autonomous architectural selection using genetic algorithms [7]. This approach enables a software system to find an architectural instance that satisfies the current situation and user requirements within a short amount of time even if the application have a large number of candidate architectural instances. Further, this selection process attempts to find an optimal architectural instance for the situation and requirements.

The remainder of this paper is organized as follows. The next section provides a motivating example that requires autonomous architectural selection at runtime. Section 3 formulates the architectural selection problem based on softgoal interdependency graphs. In Section 4, we propose a genetic algorithm-based approach to autonomous architectural selection. Section 5 evaluates our approach in terms of its performance. Section 6 compares our approach to related work. Finally, Section 7 concludes the paper.

2. Motivating Examples

2.1. Situations, Quality Attributes and Functional Alternatives

Changing environments and user requirements affect applications. For example, mobile applications are exposed

*This research was performed for the Intelligent Robotics Development Program, one of the 21st Century Frontier R&D Programs funded by the Ministry of Knowledge Economy (MKE).

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.11

to diverse factors such as the location where an application performs its function and the time when it performs its function. These factors lead to changes in application configurations. For example, when the user of an mobile application moves from indoors to outdoors, the noise level of the environment can change (usually, it gets louder). This change may have an impact on the performance of an application (e.g., an sound alert reminding about an appointment may not be effective because of the background noise). To handle these changes, we need to know which aspect from environments and user requirements affects applications and which element in an application is affected by that aspect.

First, a situational change in an environment is an influential aspect on an application. Changes in an environment can be represented by a set of several *situation variables* that represent situational aspects in the environment. For example, in mobile environments, the RSSI (Received Signal Strength Indication) level, battery level, and brightness are representative situational aspects that can affect the quality of service of applications. They can have ordinal, nominal, and numerical values, e.g., RSSI and battery level can have ordinal values while the brightness level has integer values of $[0, 255]$.

A change in situation variables represents a change of environment. For example, if the current value of the situation variable "RSSI level" changes from 1 to 5, we can assume that the environment of an application has changed. To specify the contextual change of the environment, it can have a vector of situation variables that may affect the application's performance. For example, suppose that an application A_1 may concern RSSI level, battery level, and brightness. A vector $< (rssi), (battery), (brightness) >$ can denote the contextual status of the environment (e.g., $< 0, 1, 220 >$ represents RSSI level = 0, Battery Level = 1, and Brightness = 220).

There are two types of user requirements: functional and non-functional requirements. Functional requirements (FRs) include the system behaviors that must be performed in the software system. On the other hand, non-functional requirements (NFRs)[1] address quality issues for software systems[2]. NFRs deal with the degree of satisfaction. For example, using an authentication method in an application may satisfy security requirements *at some level*. This concept is known as "satisficing" - *sufficiently satisfactory*. This term was used by Herbert Simon in the 1950s. In this paper, we deal only with changes of NFRs because applications tend to be required to change their functions at runtime according to the user's changing requirements concerning the quality of service rather than functional aspects.

To adapt to changing situations and quality attributes, a software system can have diverse alternative functions, e.g., "RichGUI," "SimpleGUI," and "NormalDisplay" as shown in Figure 1. These alternatives represent candidate functions that the application can take in situational and quality changes. Alternatives can be grouped by a type. For example, "HighContrastDisplay" and "NormalDisplay" alternatives belong to the same type, "Display" type. In each type, only one alternative can be activated (e.g., "NormalDisplay" and "HighContrastDisplay" cannot coincide).

Each alternative has relationships with quality attributes, as shown in Figure 1. For example, using a rich GUI (Graphical User Interface) may influence the application's usability and durability because the rich GUI can provide a better user experience and consume more battery life. This influence can be quantized. The quantization can describe the relationship more specifically. For example, the high contrast display alternative can have a positive impact on readability (denoted by "$+$") but a worse impact on durability (denoted by "$--$"). These impacts can be aggregated for each quality attribute as shown in Figure 1 (on the top of each quality attributes, responsiveness, usability, durability, and readability). Assume that the plus and minus signs denote "$+1$" and "-1," respectively. These aggregated scores can be used to measure how much the user is *satisficed*.

To simply measure the degree of satisfaction for quality attributes, we can integrate the scores as shown in Figure 1. Suppose that the user has the weight values (i.e., priority) of each quality attribute (0.2, 0.5, 0.1, and 0.2 for responsiveness, usability, durability, and readability, respectively). The weighted sum of the quality attributes is 0.8. This can be interpreted as the value of selected alternatives: Rich GUI, High Contrast Display, and Videotelephony.

The value of the selected alternatives can be a criterion to evaluate the selected alternatives to the current situation and the user's requirement represented by weight values. Therefore, we can identify the best combination of alternative selection by evaluating values of all possible combinations. For example, we can calculate 18 combinations of the alternatives shown in Figure 1 (three of GUI, two of display, and three of messaging alternatives) and we can find one combination that has the maximum value. This combination will provide the best user experience.

2.2. Situational and Quality Changes

The value of alternatives can be changed as the current situation changes. Value 0.8 in Figure 1 is valid in a situation where RSSI Level = 5, Battery Level = 3, and Brightness = 120. When the situation changes, the current value of the combination is not valid and it must be re-evaluated. If the situation changes to RSSI Level = 1, Battery Level = 1, and Brightness = 50, the value of the combination [RichGUI, HighContrastDisplay, and Videotelephony] can change to other values because the impact of alternatives on

[1] In this paper, we will also use **quality attributes** to represent NFRs as described in [2].

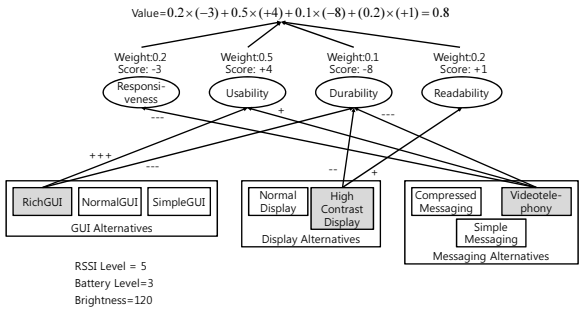


Figure 1. An example of value evaluation of selected alternatives.

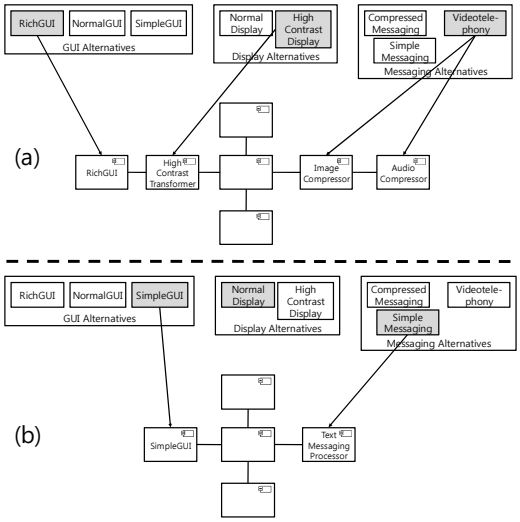


Figure 2. Examples of relationships between alternatives and an architectural configuration. (i.e., possible architectural instances)

quality attributes changes. For example, suppose that the mobile device is exposed to a low RSSI level. Then, the videotelephony function has a negative impact on responsiveness because it consumes more network resources than other alternatives. At this time, the application should select another alternative to provide a better quality of services.

Based on the changed impact values between alternatives and quality attributes, the system can re-evaluate the values of all combinations and select the best one. For every situational change, the system can re-evaluate all combinations to adapt to the current environment at runtime; however, this will be time-consuming if it involves a large number of alternatives. This time-consuming task may lead to a delay and performance degradation. Consequently, this can cause negative user experiences because the system cannot complete adaptation in time that the user can tolerate.

Similar to situational changes, the application must change its configuration when the user requirements related to quality attributes change. Although the application has the same alternatives and monitors the same situation values as those in Figure 1, the value of the selected alternatives can be changed due to the change of weight value for each quality attribute (i.e., the user may change weight values of quality attributes). For example, if the user changes weight values of quality attributes (responsiveness, usability, durability, and readability) shown in Figure 1 into $0.2, 0.2, 05$, and 0.1, respectively, the value of the selected alternatives is changed from 0.8 to -0.1.

The application must re-evaluate all combinations of alternatives to identify whether there are better alternatives that *satisfice* change user requirements. This is also time-consuming. However, it is not possible to calculate all values prior to runtime because the number of combinations of weight values and situation values is not finite (in particular, weight values are usually real numbers in $[0, 1]$). Therefore, the application should dynamically re-evaluate the values of combinations of alternatives to identify the best or better combinations in changing environments (i.e., at runtime).

2.3. Architectural Reconfiguration

After finding the best or better combination, the application must change its architectural configuration according to the selected combination. In other words, when the situation or user requirement changes, the application finds a combination of alternatives that are more appropriate for the current situation values and quality attributes, and then changes its architectural configuration according to the combination. For example, as shown in Figure 2.(a), "RichGUI" and "HighContrastDisplay" alternative can correspond to the "RichGUI" and "High Contrast Transformer" component, respectively. "Videotelephony" alternative can correspond to two components: "Image Compressor" and "Audio Compressor." If the application observes changes from the environment or user, it subsequently changes its configuration according to the selected alternatives, as shown in Figure 2.(b). These possible combinations are called *architectural instances* in this paper.

Deriving an actual software architectural configuration from a combination of architectural decision is also an important issue of software architecture research; however, this issue is not the focus of this paper and also previous studies have already dealt with this issue in terms of interface matching[10] and prescribed reconfiguration strategies[5]. In this paper, we assume that the application that applies our approach is implemented by dynamic architectures that enable the application to reconfigure its configuration.

3. Architectural Selection Problem

3.1. Quality Variables

In this section, we formulate the quality attributes using softgoal interdependency graphs (SIG) that are proposed by NFR (Non-Functional Requirements) framework[2]. A softgoal interdependency graph represents relationships between quality attributes (i.e., NFRs). A softgoal represents a quality attribute, and in an SIG, it is denoted by a cloud shape. Interdependency between two softgoals is denoted by a line connecting the two softgoals. By identifying softgoals and connecting them, an SIG represents the quality attributes of an application.

Representing quality attributes by an SIG gives several benefits to our approach. First, it helps a developer readily elicit quality attributes. The NFR framework proposed by Chung et al.[2] provides a means of identifying and analyzing quality attributes in detail. Second, this is a well-proven method for analyzing and representing quality attributes in several areas including software architectures with NFRs[9]. Third, this tree-like graph-based representation (i.e., SIG) can support the aggregation of impacts between functional alternatives and quality attributes in a bottom-up manner.

The quality attributes are formulated by a set of quality variables. Among softgoals of an SIG, only the highest quality attributes (e.g., readability, performance, durability, and usability in Figure 3) are considered to be quality variables because they will be the target of prioritization and value evaluation in our approach. A quality variable q_i can have a real number that describes how the application *satisfices* the quality attribute that the quality variable represents. A set of quality variables Q represents overall satisfaction of the user (i.e., it represents how much the user is satisficed). These quality variables should be aggregated to represent one integrated measure. A value (or utility) function U that measures the user's overall satisfaction is defined as:

$$\begin{aligned} U(Q,W) &= U(q_1, q_2, \cdots, q_n, w_1, w_2, \cdots, w_n) \\ &= q_1 \cdot w_1 + q_2 \cdot w_2 + \cdots + q_n \cdot w_n \\ &= \sum_i^n q_i \cdot w_i \end{aligned}$$

where Q is a set of quality variables, W is a set of weights, and each w_i is a weight of quality variable q_i, and n is the number of quality variables (i.e., $|Q| = n$). The sum of weights must be equal to 1 (i.e., $\sum_i^n w_i = 1$). The value of a quality variable is determined by the current situation and selected functional alternatives (to be described by the remainder of this section). The weights are defined by the user and they represent the priority of quality attributes. They can be changed by the user at runtime.

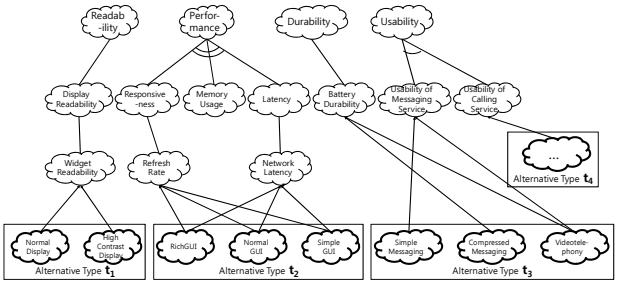


Figure 3. An example of softgoal interdependency graphs with alternative types.

3.2. Alternatives

Functional alternatives described in the previous section are denoted by operationalizing goals in our formulation. An operationalizing goal is introduced by NFR framework[2] to represent a design decision, e.g., simple, compressed, or videotelephony messaging services shown in Figure 2. An operationalizing goal in an SIG is denoted by a cloud shape with bold lines as shown in Figure 3.

An operationalizing goal can have an impact on softgoals. This relationship between an operationalizing goal and a softgoal is represented by *contribution*. An operationalizing goal contributes to one or more than one softgoals with some degrees as shown in Figure 1. In NFR framework, this relationship is quantized into some abstract notations such as "$--$ (= BREAK)" and "$+$ (= HELP)." In our approach, we do not use these notations; instead, we use situation evaluation functions (see Section 3.4) to represent the contributions of operationalizing goals.

In this approach, alternatives that have similar characteristics must be grouped by an alternative type as described in the previous section. Each type t_i can have only one value: one of the alternatives that constitutes the type t_i. In other words, alternatives that cannot coincide should be grouped as an alternative type. Each alternative should belong to only one alternative type. These alternative types are used to define architectural decision variables. An example of alternatives types is shown in Figure 3 (they are grouped by rectangles).

3.3. Architectural Decision Variable

An architectural decision variable determines part of an architectural configuration using an alternative type. In our study, one alternative type corresponds to one architectural decision variable as shown in Figure 4 (an alternative type t_i corresponds to an architectural decision variable a_i). In contrast to alternative types, architectural decision variables represent partial configurations of the ap-

plication. An alternative in an alternative type also corresponds to an architectural decision value ("NormalDisplay" and "HighContrastDisplay" are connected to $a_i = NORM$ and $a_i = HCONST$, respectively). An architectural decision value represents a partial set of components and their relationship between them as shown in Figures 2 and 4. This is formulated as follows. Suppose that a_i is an architectural decision variable. a_i can have an architectural decision value $v_i \in V$ where V is a set of architectural decision values. In this formulation, v_i must correspond to an alternative $i_j \in I$ where I is a set of alternatives. This mapping is one-to-one mapping and the number of architectural decision values (i.e., $|V|$) and the number of alternatives (i.e., $|I|$) must be equal. Also, an alternative type $t_i \in T$ must correspond to an architectural decision variable a_j and their cardinalities must be equal.

A combination of architectural decision variables comprises an architectural instance as shown in Figure 2. Let e_i be an architectural instance and it can be denoted by a vector of architecture decision variables, for example, $e_i =< a_1 = HCONST, a_2 = RichGUI, a_3 = \emptyset, \cdots, a_n = SIMPLEMSG >$. Let E be a set of possible architectural instances. We can formulate E as follows. Let $|a_i|$ be the number of architectural decision values that a_i can take. Let $|E|$ be the number of architectural instances that an application can have. Let n be the number of architectural decision variables, i.e., $|A| = n$. Then, we have the following:

$$\begin{aligned} |E| &= |a_1| \times |a_2| \times \cdots \times |a_n| \\ &= \prod_{i}^{n} |a_i| \end{aligned}$$

The number of possible architectural instances $|E|$ determines the complexity of the architectural selection problem.

3.4. Situation Variables and Functions

A situation variable s_i describes partial information of environmental changes (examples are shown in Figure 1). As described in Section 2, situation variables determine the impacts of architectural decision variables on quality attributes. To formally specify these impacts on quality attributes, we define a situation evaluation function (or situation function) as

$$\begin{aligned} f_{v_1}^{g_1}(S) &= f_{v_1}^{g_1}(s_1, s_2, \cdots, s_n) \\ f_{v_2}^{g_2}(S) &= f_{v_2}^{g_2}(s_1, s_2, \cdots, s_n) \\ &\cdots \\ f_{v_m}^{g_k}(S) &= f_{v_m}^{g_k}(s_1, s_2, \cdots, s_n) \end{aligned}$$

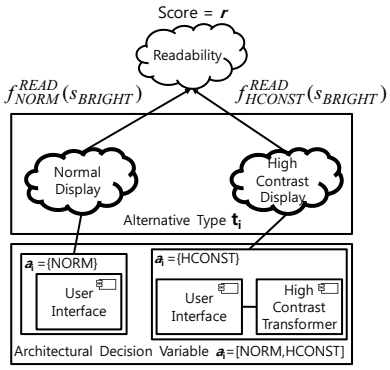


Figure 4. An example that shows interrelationships between architectural decision variables, alternative types, situation variables, and quality attributes.

where S is a set of situation variables, s_i is the *i*-th situation variable, v_j is the *j*-th architectural decision value, and g_k is the *k*-th softgoal. n is the number of situation variables and m is the number of architectural decision values. k is not the number of softgoals, but just an index of softgoals. A situation function is defined for each direct interdependency between an operationalizing softgoal and softgoal as shown in Figure 4. The number of situation functions is determined by the number of direct interdependencies between operationalizing softgoals and softgoals (i.e., the number of impacts) in the application (this is defined by an application developer who constructs the SIG of the application). Every operationalizing softgoal (which represents the related architectural decision value) must have at least one interrelationship with softgoals and the same number of situation functions; therefore, $|F| \geq |V|$ where $|F|$ and $|V|$ are the number of situation functions and the number of architectural alternative values, respectively.

3.5. Architectural Selection Problem

Based on the formulation described in the previous sections, this section describes the architectural selection problem in software systems at runtime. In this problem, quality variables are used to evaluate the user's satisfaction to the selected architectural instance, architectural decision variables are used to represent selected alternatives in this instance, and situation functions with situation variables are used to decide impacts of an architectural instance on quality attributes. An overall description of the architectural selection problem is shown in Figure 5.

The architectural selection problem is a combinatorial optimization problem[3]. In combinatorial optimization, one searches combinations in a problem space to find op-

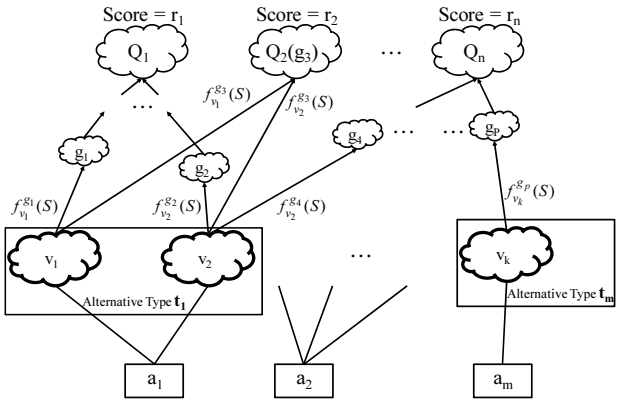


Figure 5. An overview of the architectural selection problem. ADV_i is the *i*-th quality variable, g_i is the *i*-th softgoal, v_i is the *i*-th functional alternative, a_i is the *i*-th architectural decision variable, and $f_{v_i}^{g_j}(S)$ is the situation function of alternative v_i and softgoal g_j.

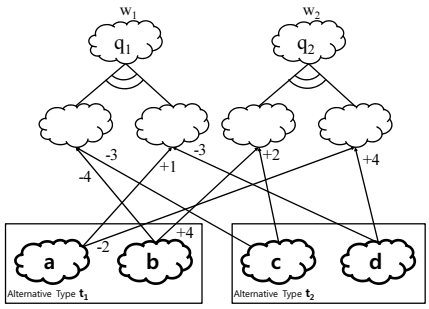


Figure 6. An example in which a greedy algorithm cannot find an optimal solution.

timal solutions according to specific evaluation criteria. In the architectural selection problem, the problem space comprises combinations of architectural decision variables (i.e., architectural instances) described in Section 3.3 and the evaluation criteria is the value (or utility) function described in Section 3.1. To calculate the result of the value function, an SIG and situation functions are required.

The goal of the architectural selection problem is to find an optimal architectural instance based on the current situation and the user's requirement represented by situation variables and quality variables (with weights), respectively. This is formulated as:

$$\begin{aligned} e^* &= \arg\max_{e_i \in E} U(Q, W) \\ &= \arg\max_{e_i \in E} U(Q_{SIG}(A), W) \\ &= \arg\max_{e_i \in E} U(Q_{SIG}(< a_1, a_2, \cdots, a_n >), W) \\ &= \arg\max_{e_i \in E} U(Q_{SIG}(e_i), W) \end{aligned}$$

where Q_{SIG} is an evaluation function based on an SIG and defined as $Q_{SIG} : A \rightarrow \hat{Q}$. $\hat{Q}$ is a set of vectors that comprise every quality variable q_i such as $< q_1, q_2, \cdots, q_m >$ where m is the number of quality variables, (i.e., $|Q| = m$). A is a vector of architectural decision variables, such as $< a_1, a_2, \cdots, a_n >$. This vector represents an architectural instance e_i. e^* is an optimal architectural instance in the current situation under the user requirements described by quality attributes.

To find e^*, value function $U(Q, W)$ must be evaluated through an SIG. In this approach, the evaluation process is conducted in a bottom-up manner from architectural decision variables to quality variables. For each instance e_i (i.e., a vector of architectural decision variables), we can directly derive a corresponding set of operationalizing softgoals (i.e., a set of alternative values). Then, we evaluate every situation function on edges, which are starting from the selected operationalizing softgoals, based on the values of the current situation variables. The results of situation functions are aggregated into related softgoals. This aggregation implies addition as shown in Figure 1.

After evaluating situation functions, the aggregated values are propagated to higher softgoals. There are three types of relationships between sub-softgoals and higher goals: direct, AND, and OR relation as shown in Figure 3. In a direct relation, the value of a subsoftgoal is directly propagated to the parent softgoal. In an AND relation, two or more than two subsoftgoals are related to a higher softgoal. An AND relation assumes the parent softgoal is satisficed if all subsoftgoals are satisficed. Therefore, every value of subsoftgoals is aggregated and the value of the parent softgoal is set to their sum if all subsoftgoals have positive values. If one subsoftgoals has a negative or zero value, the value of the parent softgoal is set to zero. Moreover, if all subsoftgoals have negative or zero values, the value of the parent softgoal is set to their sum. This rule is different from the rule provided by the NFR framework[2] (the framework only marks whether a softgoal is satisficed and unsatisficed) because it is important to measure how much the user is satisficed or unsatisficed in comparable numbers in the autonomous architectural selection problem.

In an OR relation, the parent softgoal is satisficed if, at least, one subsoftgoal is satisficed. The value of parent softgoal is determined as follows. When one more subsoftgoals have positive values, the parent softgoal takes the maximum value among the subsoftgoals; when all subsoftgoals have negative values or zero, the parent takes the minimum value among the subsoftgoals because the parent is definitely un-

satisficed; only one subsoftgoal has a positive value, the parent takes the value. Based on these propagation rules, the values of quality variables are determined.

The problem is that evaluating the value function for all possible architectural instances is time-consuming. The time complexity of this problem is determined by the number of architectural decision variables as discussed in Section 3.3. We can say that the time complexity is $O(|E|) = O(\delta^n)$ where $|E|$ is the number of architectural instances, δ represents the average number of architectural decision values in an architectural decision variable, and n is the number of architectural decision variables. This implies that an exhaustive search is not applicable for this problem (i.e., it may cause the state explosion problem). Therefore, we can apply other approaches such as greedy algorithms and dynamic programming. However, a greedy algorithm for this problem does not guarantee it will converge to an optimal solution. For example, with the SIG shown in Figure 6, the greedy algorithm cannot derive an optimal solution from the SIG. Specifically, we can determine the value of all individual operationalizing softgoals i.e., $a = -0.5, b = 0.0, c = -0.5$, and $d = 0.5$. Now, we know the best choice in each alternative type (b of t_1 and d of t_2). However, the best combination is choosing a and c. Dynamic programming can be applied to solve this problem, but it is also time-consuming (i.e., AND and OR relations in SIG do not allow substructure optimality). Therefore, it is required to provide a method to find an optimal solution for the architectural selection problem in a reasonable time span.

4. Genetic Algorithm-based Architectural Selection

4.1. Genetic Algorithm Procedure

A genetic algorithm[7] is a metaheuristic search method that approximates a solution in the solution space. It is also a well-proven method to deal with combinatorial optimization problems. In a genetic algorithm, the target problem is represented by a string of genes. This string is called a *chromosome*. Using the chromosome representation, a genetic algorithm generates an initial population of chromosomes. Then, it repeats the following procedure until a specific termination condition is met (usually a finite number of generations): (1) select parent chromosomes based on a specific crossover probability and perform crossover; (2) choose chromosomes and mutate the chromosomes based on a specific mutation probability; (3) evaluate fitness of offspring; (4) select the next generation of population from the offspring. In our approach, the above procedure will be adopted to solve the architectural selection problem. To do this, it is required to encode architectural instances into chromosomes, design the fitness function, and determine the crossover and mutation operators. The following sections will describe these issues.

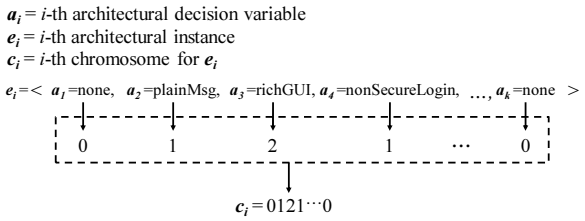


Figure 7. An example of encoding architectural decision variables into chromosomes.

4.2. Representing Architectural Instances in Genes

The main issue in applying a genetic algorithm to a certain application is encoding the problem space into a set of chromosomes. In this approach, we encode architectural instances into chromosomes because our goal is to find an optimal instance from the set of instances. We use architectural decision variables to encode instances as shown in Figure 7. The *i*-th architectural decision variable corresponds to the *i*-th digit of a chromosome. In this encoding method, the meaning of each number $0, 1, 2, \cdots, n$ is just an identifier to distinguish architectural decision values belonging to a specific architectural decision variable. Therefore, any discrete representation that can distinguish elements can be used, e.g., alphabetic representation such as $a, b, c, d, \cdots, z$.

4.3. Crossover and Mutation

Another issue of applying genetic algorithms is designing crossover and mutation operators to produce offspring. Among various crossover and mutation operators, we use two-point crossover and digit-wise probabilistic mutation.

Two-point crossover picks up two chromosomes and chooses two (arbitrary and same) positions for each chromosome. Then, it exchanges digits of the two chromosomes between two positions. We use this technique because it preserves more characteristics of parent chromosomes than other crossover techniques. Further, we assume that similar chromosomes may result in similar values to the value function. Crossover may pick up two parents in the population with crossover probability P_c. In other words, if $P_c = 0.5$, the half of the population is chosen as parents and the crossover operator produces the same number of offspring.

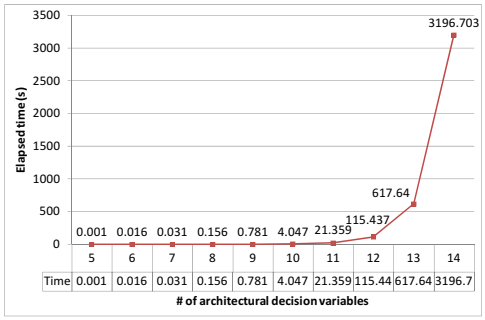


Figure 8. The performance of exhaustive search.

After performing crossover, the algorithm performs mutation. Every digit of offspring produced in the crossover step is changed to arbitrary values with mutation probability P_m. Note that if the mutation probability is too high, it cannot preserve the characteristics of parents. If the probability is too low, the algorithm may fall into local optima. Offspring produced by crossover and mutation are candidates for the next generation population.

4.4. Fitness and Selection

After performing crossover and mutation, the next step is selection. In this step, the algorithm evaluates the fitness values of all offspring and chromosomes that have better values survive. In this approach, the value (utility) function described in Section 3.1 is used as a fitness function to evaluate chromosomes and the tournament selection strategy[8] is used as a selection method. The tournament selection strategy selects the best ranking chromosomes from the new population produced by the previous steps.

The size of population determines the efficiency and effectiveness of genetic algorithms. If the size is too small, it does not allow exploring of the search space effectively, while too large a population may impair the efficiency. Practically, our approach samples at least $\delta \cdot l$ number of chromosomes where δ is the average number of alternative values for each architectural decision variable and l is the length of a chromosome.

By using the procedure described in the above sections, our approach can find the best (or reasonably good) solution from the search space when situation and requirement changes. The next section describes the result of performance evaluation.

5. Evaluation

This section provides the result of performance evaluation of our approach. First, the performance of exhaustive search has been measured to compare with the performance of our approach. Then, we have measured the performance of our approach based on the result of the previous experiment.

Every experiment was performed on SCH-V740 which is a cellular phone produced by Samsung. We implemented our approach based on Java. We designed an arbitrary SIG and gave a set of weight values to quality variables. Then, we produced architectural decision variables that have five architectural decision values on the average. For each architectural decision value, we give an arbitrary number of situation functions (i.e., impacts on softgoals). Those functions evaluate impacts on softgoals. With this setting, we conducted the following experiments.

5.1. Baseline

As a baseline, we have conducted an experiment that measures the performance of exhaustive search. In this experiment, we measured not only the elapsed time to exhaustively search every combination of architectural decision variables but also the best chromosome that will be used to evaluate the performance of our approach. As stated in Section 3.5, the architectural selection problem is a combinatorial optimization problem and has time complexity $O(\delta^n)$. The average number of architectural decision values in each architectural decision variable δ is five, therefore, the size of the search space is increasing as δ^n where n is the number of architectural decision variables. As shown in Figure 8, the device can search the problem space for an optimal architectural configuration in one second when $n < 10$. However, since $n = 10$, the elapsed time to search the problem space is exponentially increasing. The exhaustive search technique is not appropriate for the dynamic architectural selection problem because it is not acceptable for users to wait for the end of search for every moment when the current situation or the set of weights have been changed.

5.2. Performance of Our Approach

Based on the result of the previous experiment, we conducted three performance tests. The first test measures the elapsed time required to find an (near) optimal solution from the search space. It is difficult to anticipate the time required to find an optimal solution because genetic algorithms are randomized algorithms. However, we can consider a near optimal solution that is close to the best solution (such as the Las Vegas algorithm). In this test, we assume that the algorithm terminates when the difference of the elitist chromosome in the population and the best combination found in the previous experiment is smaller than 5% of the best combination (i.e., `if` $Fit(best) - Fit(elitist) < 0.05 \cdot Fit(best)$, `then` terminate the algorithm where $Fit(a)$

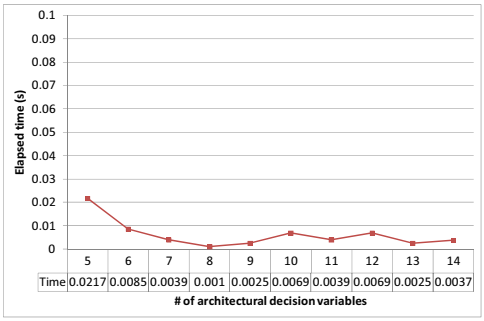


Figure 9. Elapsed time to obtain an (near) optimal solution for each number of architectural decision variables.

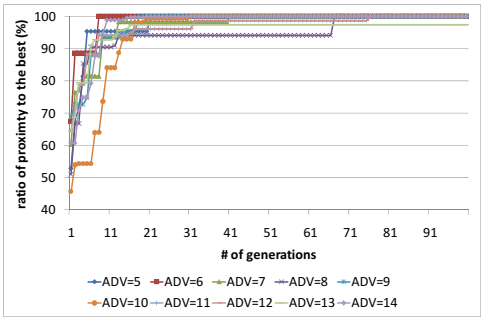


Figure 10. Ratio of proximity to the best (ADV = the number of architectural decision variables).

evaluates the value of chromosome a). As shown in Figure 9, the elapsed time to obtain an (near) optimal solution by our approach is very short compared to the time of exhaustive search (for each number of architectural decision variables, we ran ten tests and the elapsed time shown in the figure is the average value of ten tests). The elapsed time of the genetic algorithm does not increase as the number of architectural decision variables increases because it is basically a randomized algorithm as previously stated.

The next test is conducted to verify how fast the algorithm approaches the best solution. Like the Monte Carlo simulation, we fixed the number of generations and we recorded the value of the elitist chromosome for each generation. The result is shown in Figure 10. Note that y-axis represents the ratio of proximity to the best solution and the fixed number of generation is 100. For every number of architectural decision variable, the elitist chromosome quickly approaches to the best solution. In most cases, the elitist is the same before 40 generations. The approaching speed may vary for each run; however, it cannot influence the result that it finally approaches to the best solution. Further, the elapsed time for 100 generations is less than $0.01s$.

The third test shows the result of a larger number of architectural decision variables. For a larger number of architectural decision variables, it takes a very long time to find the best solution by exhaustive search. However, we can approximate that the elitist is the best or very close to the best by comparing with the elitist of previous generations. In other words, if the elitist has not been changed in a very long time, it may be the best with the high probability. In this test, the required number of generations is set to $10 \cdot l \cdot \delta$. Obviously, a large number of generations requires more time to perform the algorithm as shown in Figure 11; however, it is still much smaller than the time required to perform the exhaustive search. In practice, any time more than five or six seconds is sufficiently long for users to feel bored. Therefore, in this device, the application that uses our approach must control the number of architectural decision variable to be not over 40.

5.3. Analysis of Performance Evaluation

We have conducted a baseline experiment and three tests. The first test, which measured required time and generations to reach the best solution measured by the baseline experiment, indicates that the application can find the best combination of architectural decision variables and serve the new and best architectural instance to the user in response to every situation and requirement changes. However, practically, this type of execution cannot be applicable to real applications because it is impossible for the application to know the best solution for every situation and requirement changes. Therefore, we can fix the number of generations for each change. The second test evaluated this type of execution and shows the algorithm can find an optimal or near-optimal chromosome very fast. A fixed number of generations can be effective for a relatively small number of architectural decision variables; however, it may not be effective for a large number of them. The last test showed the result of the termination condition that finishes the algorithm when the elitist has not been changed for a specified number of generations. This type of execution can be applicable to practical systems because the number of generations proportionally varies as the number of architectural decision variables increases. In addition, even for a large number of architectural decision variables, the algorithm shows good performance compared to the exhaustive search.

6. Related Work

Floch *et al.* [4] proposed a utility-based adaptation scheme. This approach assumes that an adaptable application operates on the adaptation middleware that they have

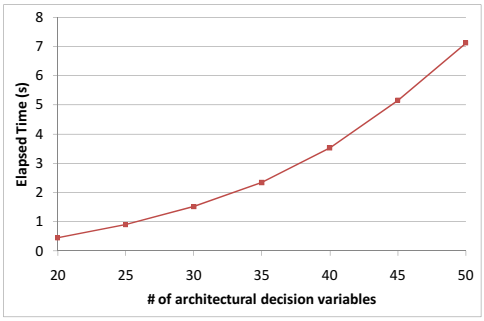


Figure 11. Required time to perform the tests to determine the elitist is the best (or very close to the best).

previously proposed[6], and the middleware monitors the current user context and system context. Based on the monitored context data, the middleware dynamically changes the application's configuration. Possible configurations are component types and their implementations. In this approach, planning is performed by mapping component implementations and properties or utility functions. When a user or system context changes, the middleware evaluates the change and compares it to the utility functions and properties of the current configuration. This adaptive planning effectively reflects contextual changes; however, they does not deal with the priority issue in which the user changes his or her preference about quality attributes.

Capra *et al*. [1] described the conflict problem in mobile applications. To deal with conflicts, they suggested a microeconomic mechanism that performs an (virtual) auction. In this mechanism, the mobile application is aware of the resource status of the device and user profiles. The application resolves intra and interprofile conflict problems using a game theoretic mechanism. Although this approach does not deal with architectural selection or adaptation issues, the idea, which autonomously selects functions of mobile applications at runtime, is related to our approach. The difference is that our approach focuses on the optimal architectural selection problem while their approach focuses on the conflict resolution.

7. Conclusions

As the software market extends its territory to mobile and ubiquitous environments, software systems are exposed to various situations and requirements. This requires the mobile application to change its architectural configuration in response to situation and requirement changes. This issue can be modeled by the architectural selection problem in which a mobile application searches its possible architectural instance and selects an optimal one to the current situation and user requirements.

In this paper, we illustrated a motivating example that requires architectural selection and formulated the architectural selection problem using softgoal interdependency graphs. Then, we proposed a novel approach to the architectural selection problem based on genetic algorithms. This approach enables a software system to autonomously search possible architectural instances (the search space) to find an optimal instance to the current situation and requirements within a short time. The evaluation of this approach showed our approach can accelerate the architectural selection of an application even if the application must search a considerable number of instances[2].

References

[1] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-aware reflective middleware system for mobile applications. *IEEE Trans. Software Eng.*, 29(10):929–945, 2003.

[2] L. Chung, B. A. Nixon, , E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 1999.

[3] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. Wiley, 1997.

[4] J. Floch, S. O. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[5] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[6] S. Hallsteinsen, E. Stav, and J. Floch. Self-adaptation for everyday systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 69–74, New York, NY, USA, 2004. ACM Press.

[7] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[8] B. L. Miller, B. L. Miller, D. E. Goldberg, and D. E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[9] N. Subramanian and L. Chung. An nfr-based framework for aligning software architectures with system architectures. In *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2*, pages 764–770, 2006.

[10] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: a combined approach to self-management. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 1–8, New York, NY, USA, 2008. ACM.

[2]Due to space limitation, we provide a supplemental material that provides discussion and future work of our approach. Visit this link: http://seapp.sogang.ac.kr:8080/discussion.pdf.

On the Use of Discretized Source Code Metrics for Author Identification

Maxim Shevertalov, Jay Kothari, Edward Stehle, and Spiros Mancoridis
Department of Computer Science
College of Engineering
Drexel University
3141 Chestnut Street, Philadelphia, PA 19104, USA
{max, jayk, evs23, spiros}@drexel.edu

Abstract

Intellectual property infringement and plagiarism litigation involving source code would be more easily resolved using code authorship identification tools. Previous efforts in this area have demonstrated the potential of determining the authorship of a disputed piece of source code automatically. This was achieved by using source code metrics to build a database of developer profiles, thus characterizing a population of developers. These profiles were then used to determine the likelihood that the unidentified source code was authored by a given developer.

In this paper we evaluate the effect of discretizing source code metrics for use in building developer profiles. It is well known that machine learning techniques perform better when using categorical variables as opposed to continuous ones. We present a genetic algorithm to discretize metrics to improve source code to author classification. We evaluate the approach with a case study involving 20 open source developers and over 750,000 lines of Java source code.

1 Introduction

Stylometry, the application of the study of linguistic style, is used to analyze the differences in the literary techniques of authors. Researchers have identified over 1,000 characteristics, or style markers, such as word length, to analyze literary works [3]. Linguistics investigators have used stylometry to distinguish the authorship of prose by capturing, examining, and comparing style markers [9]. Programming languages allow developers to express constructs and ideas in many ways. Differences in the way developers express their ideas can be captured in their programming styles, which in turn can be used for author identification.

In the context of programming languages, it has been shown that capturing style in source code can help in determining authorship. Previous work by Kothari *et al.* [13] examined source code as a text document and identified certain software developer peculiarities that persisted across different projects. Those styles were used to determine the authorship of disputed source code.

Using text-based style markers such as line length and 4-character sequence distributions, they created developer profiles. These profiles were used to determine the authorship of source code with a success rate of over of 80% on a data set consisting of 24 open source projects.

Author identification is useful in several real world applications, including criminal prosecution, corporate litigation, and plagiarism detection [14].

In this paper we focus on improving classification through the discretization of metrics used to construct developer profiles. Discretization is the process of partitioning a continuous space into discrete intervals. For example, some developers may use verbose language to comment their source code. Instead of quantifying the size of their comments based on the number of characters, words, or lines of text, one can create three categories, “short”, “medium”, and “long”. It is well known that a good discretization of data can improve the process of classification, not only in accuracy but in efficiency [7, 4].

The major challenge in discretization is how to determine the optimal number and intervals of the categories. Considering the previous example of describing the verbosity of comments in source code, it is a nontrivial task to determine how many categories of sizes there are, and how to define the intervals marking those categories. Finding how many intervals a continuous space can be divided into and what those intervals are, defines the optimization problem of discretization.

In previous work [13, 14] to build developer profiles, the authors discretized metrics with small intervals. How-

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.18

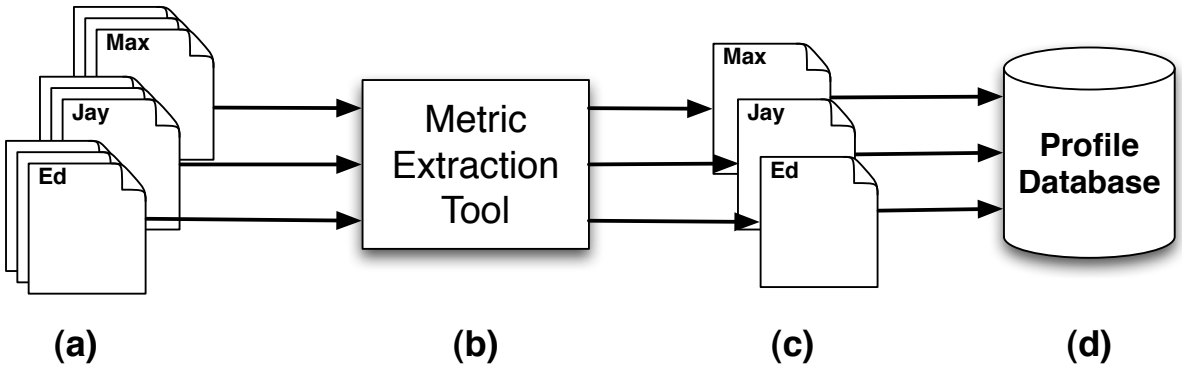


Figure 1: Process of computing developer profiles; (a) Source code samples of known developers, (b) Metric extraction tool, (c) Metric evaluations per developer, (d) Database of profiles.

ever, Shevertalov *et al.* [23] demonstrated that discretizing or "binning" large histograms into wider intervals when classifying packet streams proved to be more effective. Our work applies Shevertalov's technique to solve the discretization problem as it applies to textual metrics in source code classification.

The rest of this paper will first describe related work in extracting style from source code and discretization for use in machine learning in Section 2. Section 3 describes the process of metric extraction. The genetic algorithm developed to discretize those collected source code metrics is presented in Section 4. Section 5 demonstrates the effectiveness of our technique on a case study involving 20 open source developers and a code base of over 750,000 lines of code. We conclude with our analysis of the results as well as our plans for future work in Section 6.

2 Related Work

A prominent example in the use of style markers to verify authorship is the Bacon/Shakespeare case [10]. It had been argued that Bacon authored works that have been credited to Shakespeare. This claim has since been disproved by forensic linguists using style markers. They debunked the claim that Bacon had written Shakespeare's works by using word length frequencies and the most-frequent-word-occurrence statistics.

Oman and Cook [17] performed preliminary investigations into style and its relationship with authorship. Spafford and Weeber discussed concepts such as structure and formatting analysis [24]. Sallis compared software authorship analysis to traditional text authorship analysis [20, 21]. Gray, Sallis, and MacDonell have published multiple articles on performing author identification using metrics [8] and case-based reasoning [22]. They have also investigated author discrimination [16] via case-based reasoning and statistical methods.

More recently, Ding and Samadzadeh used statistical analysis to create a fingerprint for Java author identification [6]. Their technique makes use of several dozen metrics (*e.g.*, mean-function-name length), which they have statistically correlated to identify developers. Their metrics extraction technique is similar to our own. However, where they use mostly scalar metrics derived from the source code, such as means, our metrics are formulated as histogram distributions. Our conjecture is that this can capture more detail about a developer's style, while still providing good identification capability.

An issue in authorship identification, as well as other data mining problems, is the question of data selection. One can extract a multitude of metrics from source code. However, it is difficult to choose metrics that are useful. This problem is made more difficult by the fact that a different set of metrics may have better performance when considering different groups of authors and therefore needs to be recalculated for each sample set. Kothari *et al.* [13] used entropy filtering to select optimal metrics. Lange *et al.* [14] used a genetic algorithm to achieve similar results.

Neither approaches to identify source code authorship used discretized metrics, which we will show improve the quality and performance of authorship identification techniques. Dougherty *et al.* [7] described the need for machine learning algorithms to have a discretized search space, and the effectiveness of various algorithms when provided categorical variables as opposed to continuous ones. For example, they demonstrate that the C4.5 induction algorithm [19] provides significantly better results when used with discretized input data.

Dougherty *et al.* [7] also described range and frequency-based discretization techniques. These are simple and efficient unsupervised techniques that can produce good results under the proper circumstances. We discuss these ap-

proaches in greater detail in Section 5.

3 Metric Extraction

To determine the authorship of source code, we first extract developer profiles for a population of known developers. Figure 1 depicts the tool-chain for developing a database of profiles. The profiles describe inherent characteristics found in the source code of developers.

We begin by obtaining several samples of code from a population of developers. We associate the samples to their respective developers and process them through one or more metric extraction tools. For each developer we obtain a list of metrics and their values. We store the metrics as developer profiles in a database for use in determining authors of unclassified source code.

We store developers' profiles as histogram distributions of the extracted metrics. For example, Figure 2 shows the histogram of the line lengths for a particular developer. The developer's sample source code exhibits line lengths varying from zero characters to 120 characters, as can be see on the horizontal axis. The vertical axis indicates the normalized frequency of each line length.

Previous work [13, 14] concentrated on creating a large space of metrics to use in classification. Our goal in this work was to evaluate the effect of discretization using a genetic algorithm. Therefore, we restricted our analysis to the four metrics that seemed to produce the best results as identified by previous work. Those metrics are:

leading-spaces measure the number of whitespace used at the beginning of each line. The x-axis value represents the number of the given whitespace characters at the beginning of each line.

leading-tab measure the number of tab characters used at the beginning of each line. The x-axis value represents the number of the given tab characters at the beginning of each line.

line-len measures the length of each line of source code. The x-axis value represents the length of a given line.

line-words measures how densely the developer packs code constructs on a single line of text. The x-axis value represents the number of words on a line.

4 Genetic Algorithm

Previous work by Kyoung-jae Kim and Ingoo Han used a genetic algorithm (GA) to perform discretization as a part of an artificial neural network (ANN) system to predict the stock price index [12]. They determined that an ANN had problems with large, noisy, and continuous data sets. Therefore, they implemented a GA to mitigate those limitations. They used the ANN as a part of the evaluation function to evolve good discretization policies.

Guided by this approach, we coupled the GA used to discretize the data with our problem of classification.

This section describes the elements of our genetic algorithm. Section 4.1 describes how we encode the discretization of a histogram representation of style metrics. Section 4.2 presents the evaluation function used in the GA process, and Section 4.3 discusses the GA's breeding function.

4.1 Encoding

Candidate discretizations produced by the GA described in this section are encoded using a binary encoding scheme. A binary encoding scheme was chosen because it is generic and commonly used [25]. This encoding enables us to leverage previously established techniques.

To encode a discretization we need to identify break points, namely those values that begin and end each discrete interval. In our scheme, a 1 represents the location of a break point (Figure 3). For example, if the original histogram has the following five categories; 2,4,5,7, and 8, an encoding of `01001` corresponds to three buckets: $x < 4$, $4 \leq x < 8$, and $8 \leq x$.

4.2 Evaluation Function

Once we have an encoding, a GA requires an evaluation function to assess the fitness of a particular discretization. Four parameters must be computed to calculate the fitness:

- number of misclassifications
- number of bins
- distance from each classified entity to the correct class
- distance from each classified entity to the incorrect class

To train the GA, the set of training source files is divided into two separate sets, l_1 and l_2. l_1 is used as the learning set during and l_2 as the corresponding test set. The set containing all of the source files of l_1 and l_2 is herein referred to as the training set.

The evaluation function, illustrated in Figure 4, accepts a chromosome representing a discretization as input and

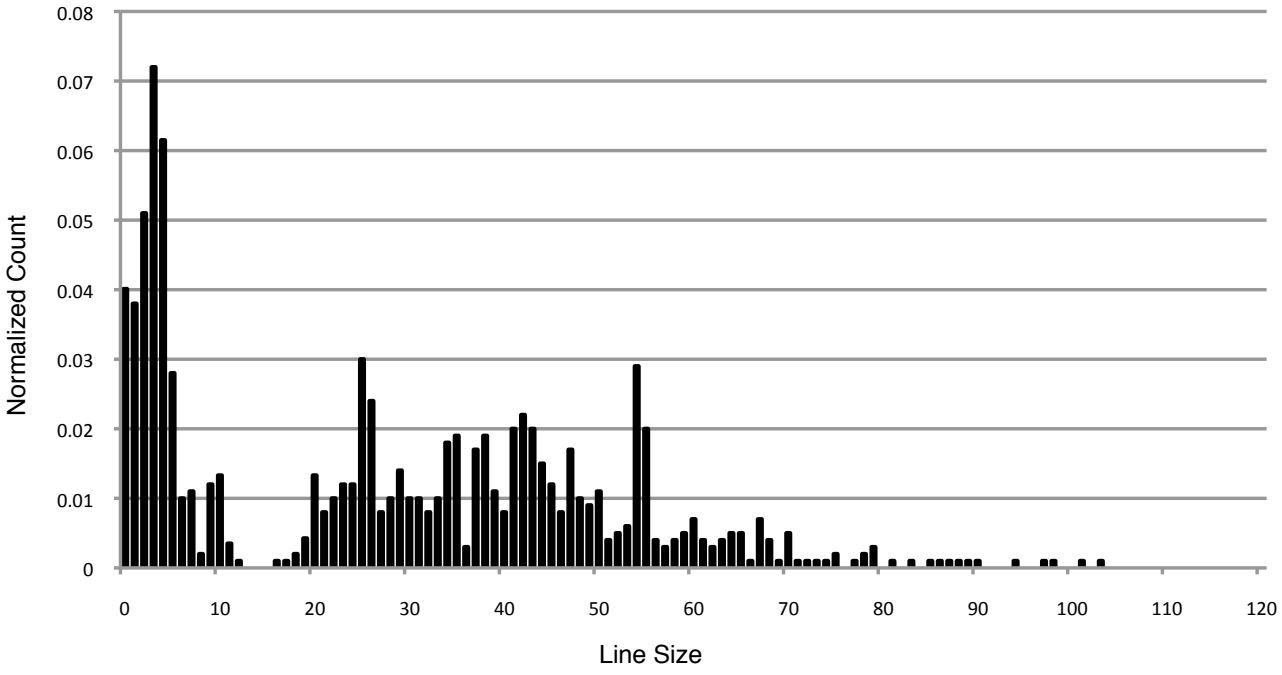


Figure 2: Histogram of line sizes; Horizontal axis indicates the line size values, and vertical axis indicates the normalized frequency, or count, of that line size value for the particular developer.

converts the undiscretized histograms into discretized histograms. The evaluation function then performs a nearest neighbor classification of the data. The learning set is used as expert knowledge by the classifier.

To classify samples using the nearest neighbor classifier, a representative histogram for each class must be identified. One way to define a representative histogram is by computing the average among a set of histograms. However, when classifying styles, we found that including all of the learning histograms in the representative set produced better results.

To determine the class of a sample from the test set, the Euclidean distance is calculated between it and each of the representative histograms in the learning set. The test sample is labeled as belonging to the same class as the representative histogram it is closest to.

The performance of the nearest neighbor classifier is improved by either reducing the number of histograms in the learning set or, by reducing the number of categories in each histogram. Once classification is complete, a chromosome's value is computed via::

$$E(x) = \frac{1}{100a + b + c - d}$$

In the previous equation, a is the number of misclassifications, b is the number of bins in the discretized histograms, c is the average Euclidean distance between histograms in the testing set and the closest histogram of the same class in the learning set, and d is the average Euclidean distance between histograms in the testing set and the closest histogram of a different class in the learning set. Since classification accuracy is the most important criterion, the number of misclassifications is weighted by an arbitrarily large factor in the equation; in this case, 100.

While keeping the number of misclassifications low is the top priority, we found that there were many different solutions with the same number of misclassifications. Therefore, other parameters were needed to further differentiate between candidate solutions.

The number of categories, the b parameter, was chosen to weigh the GA toward solutions with a smaller number of discrete intervals. The c and d parameters allow for small improvements. As the GA improves, the histograms in the testing set become more similar to histograms that belong to the same class and more dissimilar to histograms that belong to other classes. In early experiments the c and d parameters were ignored, thus leading to a quick convergence. By including these parameters, the GA converges slower, thus providing better results.

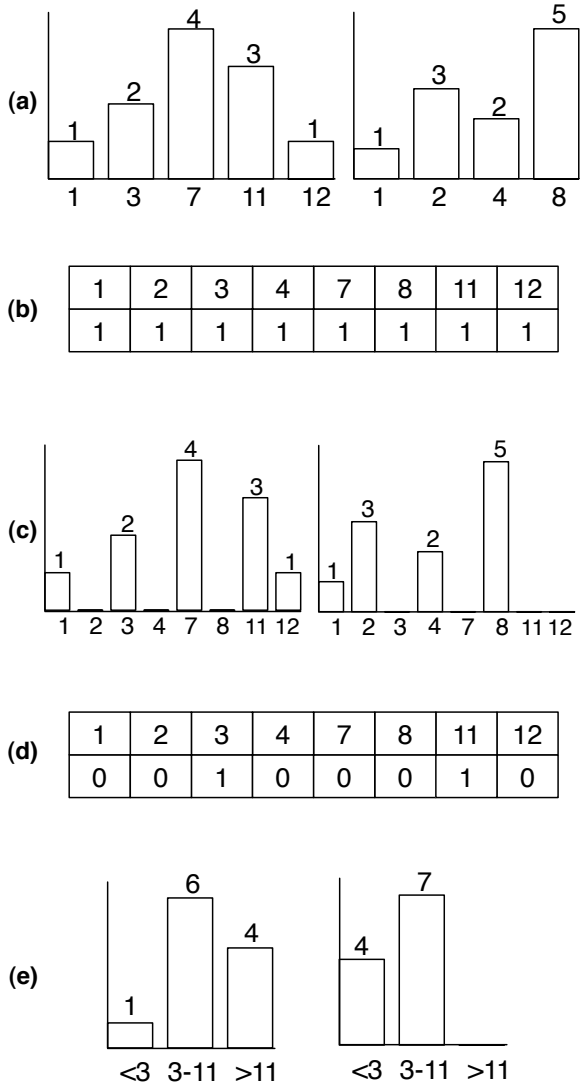


Figure 3: Process demonstrating how the encoding can be used to combine multiple histograms. **(a)** presents the original histogram. **(b)** illustrates the encoding where it is composed of all 1's and thus every bucket is its own bin. **(c)** demonstrates the results of the string encoding described in **(b)**. **(d)** presents another sample encoding such that the result is three buckets: 0-2, 3-7, and 8-12. **(e)** demonstrates the results of the discretization described in **(d)**.

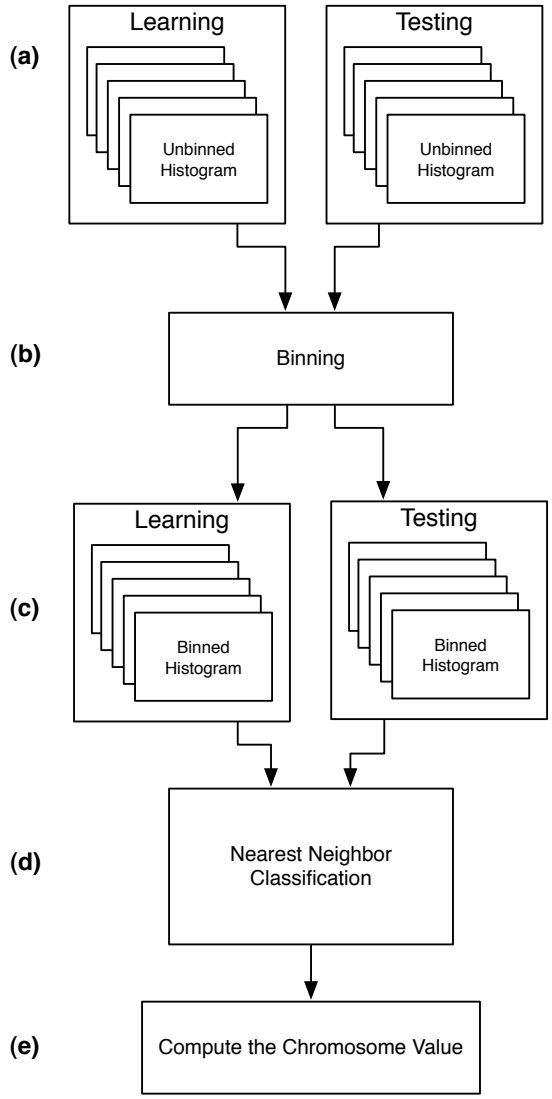


Figure 4: Process used to compute the parameters of the evaluation function. **(a)** the training set is divided into two sets, learning and testing. **(b)** all histograms are discretized based on the chromosome being evaluated. **(c)** a discretized learning and testing sets of histograms are produced. **(d)** the testing set is classified using the nearest neighbor classifier. **(e)** the chromosome value is computed.

4.3 Breeding Function

After a population of chromosomes is evaluated, candidates are chosen to breed a new population. They are selected using the roulette selection algorithm, where the probability that a chromosome is chosen is weighted by its relative fitness.

Once parents are chosen into the breeding pool, they are bred two at a time, at random, with replacement. The parents produce two children with a probability that is a configurable parameter in the implementation; it was set to 0.9 in in the case study. If they do not mate, the parents themselves are added to the next generation. In addition, the algorithm ensures that the next generation does not contain identical chromosomes. Using this restriction the GA is prevented from converging prematurely.

A mutation operation is applied to the newly generated set of chromosomes. It is applied with a probability p_m, corresponding to the current performance of the algorithm. When a mutation is applied, it changes a bit in the encoding from 1 to 0 with a probability p_c, and from 0 to 1 with a probability $1 - p_c$. In this implementation, p_c was set to be greater than 0.5 because we want the algorithm to spend more time exploring solutions with fewer categories (*i.e.*, the algorithm is biased towards fewer discrete intervals). The encodings, represented as bit strings, can be very long containing mostly 0s. If the mutation function simply flips a random bit, it is more likely to change a 0 to a 1 and thus increase the number of discrete intervals. We found that by having greater control over which was modified, a 0 or a 1, we are able to emulate two different discretization operations, splitting a category, in the case of a 0 changing to a 1; and merging two categories, in the case of changing a 1 to a 0. This abstraction allows the algorithm to converge more quickly.

The probability of mutation is set using a technique similar to simulated annealing [2]. As a generation improves, meaning that the average fitness of the population improves. The probability of mutation is decreased by a relatively large amount. If a generation does not improve, the probability of mutation is increased by a small amount, thus introducing more randomness to the search. In our case we used 0.001 as this probability. By allowing the algorithms to dynamically adjust the rate of mutation, it strikes a balance between exploring a local area of the search space, when mutation is low, and exploring the entire search space, when mutation is high. In our experiments this prevented the algorithm from converging too early by diversifying the pool of potential solutions.

5 Case Study

To demonstrate the effectiveness of discretization via a GA, we applied our technique to a sample of open-source developers' source code. The sample consisted of 60 projects by 20 developers (3 projects from each developer). Additionally we restricted the number of metrics to the following four, so that we could compare our discretized results with the undiscretized results of previous work:

- leading-spaces
- leading-tabs
- line-length
- line-words

This case study uses source code from 20 developers found on the Sourceforge website [11]. We found the 20 most active developers who had at least three projects for which they were the sole developer. In total, the 60 projects constituted over 750,000 lines of code.

Two projects were used for training, and the third for assessing the evolved solution. The genetic algorithm requires two sets to complete its work. The first is used to extract metrics and the second to assess candidate discretizations.

The first hurdle we approached when attempting this case study was that the performance of the GA evaluation function was poor in terms of completion time, taking as long as 6 hours to evaluate a single population. To mitigate this problem, the algorithm was modified to use a smaller subset of source files to perform the evaluation. Five files from each developer were chosen at random during each evaluation step. This further evens out the playing field as the number of files was inconsistent across developers. They varied anywhere from 5-6 files to as much as 300 per project. By making this adjustment, the GA would not be biased toward any single developer and treat each one as having the same weight.

In addition to comparing the GA-based discretization with non-discretized data, we wanted to evaluate it against two other discretization methods, range based and frequency based discretization [7]. Both of these are unsupervised methods that are relatively simple to implement and are not computationally expensive.

Range based discretization creates N buckets of identical size, where N is specified by the user. The algorithm first determines the domain size of the undiscretized data, either through user input or by deriving it from the learning data set. It then divides the domain by N to determine the size of each interval. For example, assume a size distribution with the smallest observable size being 60, the largest being 1000, and a chosen N of 10. The new histogram will have 10 categories each of size $(1000 - 60)/N = 94$. Thus category 0 in the new histogram would contain the sum of categories 0 to 93 from the original histogram.

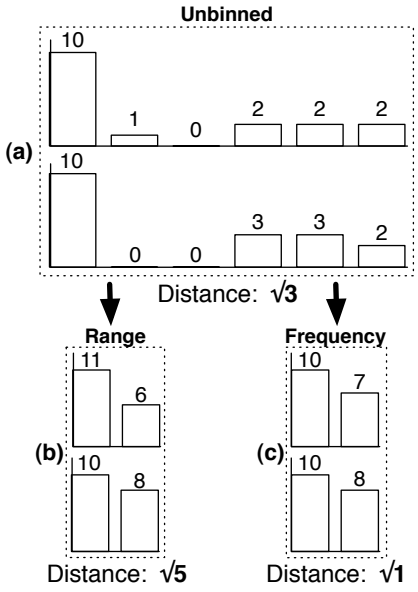


Figure 5: Range-based discretization improves results in this example. Diagram **(a)** shows two sample histograms. The discretization increases the distance between the two histograms. Range discretization **(b)** achieves the goal, and frequency discretization **(c)** provides results that are worse than using no discretization.

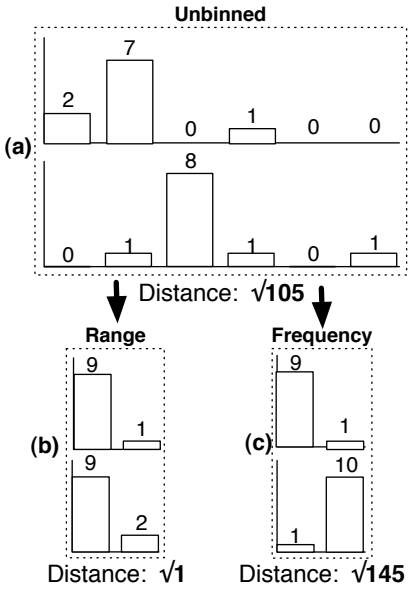


Figure 6: Frequency-based discretization improves results in this example. Diagram **(a)** shows two sample histograms. Frequency discretization **(c)** increases the distance between the two histograms whereas range based discretization **(b)** provides results that are worse than using no discretization.

Frequency-based discretization works in a manner similar to range-based discretization. It also requires a user specified parameter N to separate the domain into N categories. However, unlike the range algorithm, frequency-based discretization attempts to ensure that if the learning sample is combined into a single histogram, each interval will contain roughly the same quantity. To put it another way, frequency-based discretization attempts to split the domain such that, when the new histograms are combined, the variance between each interval is minimized.

Both range and frequency discretization algorithms work well in certain cases. For example, range-based discretization produces better results when the data is mostly uniform with a few spikes. By enabling the user to combine a number of small categories into a single larger one, small differences in the distribution can be accommodated for and given greater weight. Figure 5 demonstrates a situation where the range-based discretization approach is more effective than frequency discretization. Because the histograms are compared using Euclidean distance, the figures present the distance value without evaluating the square root.

Frequency-based discretization is useful when there are few spikes, signifying unique classes, found in the data such that the elements around those spikes are a result of noise or error in the sensors collecting the data. Figure 6 demonstrates the situation where the frequency-based discretization approach is more effective. Similarly to Figure 5, we used the Euclidean distances without evaluating the square root in order to compute the distance between two histograms.

Upon completion, the GA produces a result that is used to create a database of discretized developer profiles. This result is evaluated by classifying a third set of files. Figure 7 describes the process of using the database and a classification tool to determine the authorship of the code. Given the unidentified source code, it is discretized in the same manner as the code used to create the database of developer profiles. Once the testing code is discretized, a classification tool uses the provided database of developer profiles to classify the unknown code. This tool ranks the likelihood of each developer in the population being the author of the unidentified source code by computing the similarity of their profile and the corresponding metrics of the source code in question.

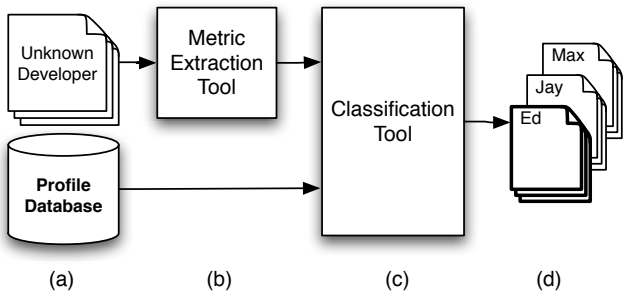


Figure 7: Process of classifying authorship of source code; (a) Input of source code with unidentified author and database of developer profiles, (b) Metric extraction tool, (c) Classification tool, (d) Classified author.

When verifying our results we used the Weka classification tool [26]. It is equipped with several different classifiers and is easy to use and integrate into a test suite. We used the IB1 [1] nearest neighbor classifier implemented in Weka to evaluate the previously described discretization methods. IB1 was used because it is most similar to the classification algorithm used by the GA when evaluating potential solutions.

Table 1 presents the results of classifying unidentified source code to a developer profile using the discretization methods described earlier. Individual files, as well as entire projects were classified for each discretization method. A project was classified to the developer profile that the majority of the files in the project were classified to. For example, consider a project containing 50 source files, and a population of 21 developers. Assume that 10 of the 50 files were attributed to a single developer, and the remaining 40 files were classified to the 20 other developers, where each developer had 2 files attributed to him. The project would be classified as the work of the developer who had 10 files attributed to him.

In the case of file-based classification, the GA outperformed all of the other approaches of discretization, as well as no discretization, but only marginally. When comparing it to range-based discretization, it successfully classified nearly twice as many files. Frequency-based discretization successfully classified only 1% fewer files than the GA with 53.3%. Using no discretization, 46.1% of files were successfully classified.

When classifying projects, the GA was once again more successful than the other approaches, with a greater margin than the classification of files. Using the GA, 75% of the projects were correctly classified. Frequency-based discretization performed only slightly worse with 70% successful classifications. Range-based discretization and no discretization were both able to correctly classify 60 and 65% of projects, respectively.

Algorithm	Files	Projects
None	46.1%	65.0%
Frequency	53.3%	70.0%
Range	30.3%	60.0%
GA	54.3%	75.0%

Table 1: The final results of classifying discretized data. The first column lists the discretization algorithm. The second column is the percentage of individual files classified correctly. The third column is the percentage of projects classified correctly. A project was attributed to an author based on the number of classification of the individual files comprising that project. The author of the project was the author with the most files attributed to him. For example, if 10 out of 50 files of a project were classified to a single developer, and the remaining files were classified evenly among 19 other developers, then the project would be classified as developed by the author with 10 files attributed to him.

This result is comparable to that of Shevertalov *et al.* [23]. Applying a GA to discretize traffic samples of networked applications found that, while the GA outperformed other methods, one of the two alternate methods provided trivially worse results. They found range-based discretization a clear second, while in our effort it was the frequency-based discretization that came in second to the GA. This demonstrates that unlike range-based and frequency-based discretization, which perform well only under specific situations, the GA approach consistently performs well due to automatic optimizations.

One of the surprises of this experiment was how well the undiscretized tests performed. Its performance was close to matching that of discretized data. Further examination of the results indicated that undiscretized results were more fragile. In other words, slight variations in the data would cause great variations in results. The distance between files classified correctly was smaller in the undiscretized data versus the discretized approaches. A file that was classified as written by a particular author, using discretized metrics, would be chosen with greater confidence.

In addition to providing fragile results, the undiscretized data set was far larger and therefore took nearly ten times longer to classify than its discretized counterpart. This was a result of the fact that the undiscretized data set resulted in 2044 intervals, whereas the data discretized by the GA resulted in 163 discrete intervals.

6 Conclusions and Future Work

This paper presents an approach to discretize metrics using a GA for the purpose of source code authorship identification. The approach builds on previous work [13, 14] of extracting source code metrics to build developer profiles. Since neither work employs discretized metrics, which are known to improve the quality and performance of machine learned classification algorithms, we augment these approaches by partitioning the space of extracted metrics into discrete intervals.

Similarly to the work of Lange *et al.* [14] and Kothari *et al.* [13], we build a database of developer profiles based on metrics extracted from source code samples. Our approach differs in that, we use a GA to discretize the metrics stored in this database.

To demonstrate the effects of discretizing source code metrics, we developed a case study using source code samples from a population of real-world open-source developers. The population consisted of 20 developers, each with three projects that they were the sole author of, and equated to over 750,000 lines of code.

We compared the effect of discretizing the metrics using the GA to other methods of discretizing data. In particular, we considered range-based discretization, frequency-based discretization, and no discretization. As expected, using metrics discretized by the GA resulted in a greater number of successful classifications. When classifying individual files we observe a success rate of 54.3%. When attributing an entire project to a single developer we note a success rate of 75%.

In our work, frequency-based discretization resulted in marginally fewer successful classifications than the GA. Shevertalov *et al.* [23] noted that range-based discretization came in a close second to their GA-based discretization. This indicates that, even though the GA may only provide slightly improved result, it is more robust than the other techniques of discretization due to its automatic optimizations.

In this work, we have demonstrated that the effect of using discrete intervals, particularly those generated by a GA, provides for improved quality, performance, and robustness in the classification of source code.

Our plan is to continue working on the subject of this paper. Specifically, we would like to perform the following:

Optimization for alternate classification algorithms In this work we used the naïve IB1 classifier. This algorithm performs simple nearest neighbor classifications. In the future we would like to optimize a discretization for more advanced classification algorithms. The machine learning literature suggests that using classification algorithms such as VFI [5], Bayes [15], and J48 [18] would produce better classifications.

Incorporation of additional metrics We used only four simple text based metrics to demonstrate the effects of discretization. Using a larger number of metrics, that are better able to capture the styles of developers, would clearly improve our classification results.

A larger case-study We would like to expand our population of developers to include multiple communities. For example, our current work only considers open-source developers; we would like to augment this population by including academic developers of various aptitudes, as well as corporate developers writing production code.

References

[1] D. Aha, D. Kibler, and M. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, 1991.

[2] E. K. Burke and G. Kendall, editors. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer, 2005.

[3] F. Can and J. M. Patton. Change of writing style with time. *Computers and the Humanities*, 38(1):61–82, 2004.

[4] M. Chmielewski and J. Grzymala-Busse. Global discretization of continuous attributes as preprocessing for machine learning. *International Journal of Approximate Reasoning*, 15(4):319–331, 1996.

[5] G. Demiroz and H. A. Guvenir. Classification by voting feature intervals. In *ECML '97: Proceedings of the 9th European Conference on Machine Learning*, pages 85–92, London, UK, 1997. Springer-Verlag.

[6] H. Ding and M. Samadzadeh. Extraction of Java program fingerprints for software authorship identification. *The Journal of Systems & Software*, 72(1):49–57, 2004.

[7] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In *International Conference on Machine Learning*, 1995.

[8] A. Gray, P. Sallis, and S. MacDonell. Identified: A dictionary-based system for extracting source code metrics for software forensics. *seep*, 00:252, 1998.

[9] D. I. Holmes. Authorship attribution. *Computers and the Humnities*, 28:87–106, 1994.

[10] J. Hope. *The Authorship of Shakespeare's Plays*. Cambridge University Press, Cambridge, 1994.

[11] S. Inc. Sourceforge.net: Open source software. `http://www.sourceforge.net/`.

[12] K. jae Kim and I. Han. Genetic algorithms approach to feature discretization in artificial neural networks for the prediction of stock price index. *Expert Systems with Applications*, 19(2):125–132, August 2000.

[13] J. Kothari, M. Shevertalov, E. Stehle, and S. Mancoridis. A probabilistic approach to source code authorship identification. In *Proceedings of International Conference on Information Technology: New Generations*. IEEE, 2007.

[14] R. Lange and S. Mancoridis. Using code metric histograms and genetic algorithms to perform author identification for software forensics. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, New York, NY, USA, 2007. ACM Press.

[15] P. Langley, W. Iba, and K. Thompson. An analysis of Bayesian classifiers. *Proceedings of the Tenth National Conference on Artificial Intelligence*, 228, 1992.

[16] S. Macdonell, A. Gray, G. MacLennan, and P. Sallis. Software forensics for discriminating between program authors usingcase-based reasoning, feedforward neural networks and multiplediscriminant analysis. *Neural Information Processing, 1999. Proceedings. ICONIP'99. 6th International Conference on*, 1, 1999.

[17] P. W. Oman and C. R. Cook. Programming style authorship analysis. In *CSC '89: Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 320–326, New York, NY, USA, 1989. ACM Press.

[18] J. Quinlan. C4. 5: Programsfor Machine Learning. *Morgan Kaufmann*, 1993.

[19] J. R. Quinlan. *C4.5 programs for machine learning*. Morgan Kaufmann, 1993.

[20] P. Sallis. Contemporary Computing Methods for the Authorship Characterisation Problem in Computational Linguistics. *New Zealand Journal of Computing*, 5(1):85–95, 1994.

[21] P. Sallis, A. Aakjaer, and S. MacDonell. Software forensics: old methods for a new science. In *Proceedings of International Conference on Software Engineering: Education and Practice*, page 481, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[22] P. Sallis, S. MacDonell, G. MacLennan, A. Gray, and R. Kilgour. Identified: Software authorship analysis with case-based reasoning. *Proc. Addendum Session Int. Conf. Neural Info. Processing and Intelligent Info. Systems*, pages 53–56, 1997.

[23] M. Shevertalov, E. Stehle, and S. Mancoridis. A genetic algorithm for solving the binning problem in networked applications detection. In *Proceedings of the IEEE Congress on Evolutionary Computation*, August 2007.

[24] E. Spafford and S. Weeber. Software forensics: Can we track code to its authors. Technical Report CSD-TR 92-010, Purdue University, Dept. of Computer Sciences, 1992.

[25] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, June 1994.

[26] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2 edition, 2005.

Search-Based Testing with in-the-loop Systems

Dr. Joachim Wegener, Peter M. Kruse
Berner & Mattner Systemtechnik GmbH, Berlin, Germany
{joachim.wegener, peter.kruse}@berner-mattner.com

Abstract
With the continuously growing software and system complexity in electronic control units and shortening release cycles, the need for efficient testing grows. In order to perform testing of electronic control units in practice hardware-in-the-loop test environments are used to run the system under test in a simulation environment under real-time conditions. Tests are usually implemented manually. Even though a lot of academic work has shown the potential of evolutionary testing to fully automate testing for different testing objectives, it is not used in industrial practice. In this work we develop an integration of evolutionary testing with a testing platform supporting model-in-the-loop-, software-in-the-loop- and hardware-in-the-loop-testing of embedded systems. We demonstrate the use of evolutionary testing for functional testing in an industrial setting by applying the developed solution to the testing of an antilock-braking-system electronic control unit.

Keywords: Testing infrastructure, Evolutionary Testing, Hardware-in-the-loop-testing, Antilock-braking-system, Functional Testing

1 Motivation

With the continuously growing software and system complexity in electronic control units and shortening release cycles, the need for efficient testing grows. Test-automation is a must to reduce expensive human efforts. Therefore, a lot of testing tools are on the market specialised in test automation. The main emphasis of the tools is the automation of test execution, monitoring, and test documentation. Seldom, support for test case design is offered.

System testing of electronic control units is usually black-box-testing. Test cases are manually derived from the specification. Various works propose the use of evolutionary testing for the automation of test case design. Most common is the automation of structural test case design, but even for the automation of structural testing evolutionary testing [1] is not widely used in industrial practice due to the missing tool support. In practice, even more important is functional testing validating the system under test against its specification. Only few works have tried to apply evolutionary testing for functional testing, e.g. [2], [15]. Functional testing using search-based techniques is not widely common, because efforts for the development of the fitness functions and for setting up appropriate test environments is high.

Buehler and Wegener [2] showed that driver assistance systems are test objects which could highly benefit from evolutionary testing. However, the tests performed were executed in a software-in-the-loop (SiL) test environment not taking the real electronic control unit into account.

In this work we develop a testing solution which supports search-based functional testing for model-in-the-loop (MiL) testing, software-in-the-loop (SiL) testing and hardware-in-the-loop (HiL) testing in a uniform way. The testing framework developed has been evaluated for testing the functioning of a serial production antilock-braking-system (ABS).

2 Test Automation Framework for Evolutionary Functional Testing

Electronic control units are used in nearly all industrial areas to control complex systems like airplanes, cars, trains, engines etc. Usually, testing of such systems contains unit testing, integration testing and system testing. When model-based development is in place also model testing is performed. Common test platforms for the testing are model-in-the-loop testing (testing the model or parts of the model in a simulation environment), software-in-the-loop testing (testing the resulting software in a simulation environment) and hardware-in-the-loop testing (testing the software integrated on the electronic control unit in a real-time simulation environment). Sometimes also processor-in-the-loop testing is performed (testing the resulting software cross compiled on an evaluation board with the target processor in a simulation environment).

In order to provide a testing framework that supports search-based testing for different testing phases and on different testing platforms we integrate different hardware and software components (Fig. 1).

modularHiL

modularHiL [3] is a modular, universal hardware-in-the-loop test system developed by Berner & Mattner. Hardware-in-the-loop testing systems are necessary to test electronic control units in a simulation environment in real-time. In order to test electronic control units their interfaces are connected to the hardware-in-the-loop test system providing the corresponding input signals and reading the output signals. Usually, a simulation environment is

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.15

necessary to run the tests and to simulate the real application environment of the electronic control unit.
As an individual module, with flexible signal conditioning and the use of open industry standards, modularHiL could be used for testing a single embedded control unit. Using an optical networking technology it is possible to link several modularHiL systems together to form a powerful test environment for integration testing. During integration testing the interplay of several electronic control units is tested. Because it uses the same modules for component and integration test systems, modularHiL offers customers cost benefits and high flexibility.

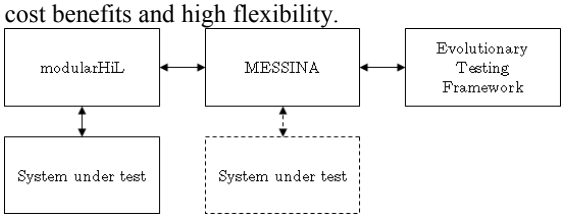


Figure 1: Components of the Test Automation Framework for Search-Based Testing (for testing in a MiL/SiL scenario no hardware-in-the-loop test system is needed, tests are directly executed by MESSINA)

MESSINA
MESSINA [4] is a testing infrastructure allowing the implementation of hardware and software independent test sequences in different notations such as UML, Java, or TPT [5]. Using two abstraction layers it offers execution on different platforms. The first layer is the signal pool containing all system signals provided by the connected devices or software devices. The signal pool allows easy read- and write-access to all the signals used by the system under test or the simulation environment. The second layer is formed by abstract in-the-loop systems, which can be MiL, SiL and HiL systems.
Three stages of testing are necessary and common. Standards like ISO 26262 [12] require even more test stages. The model has to be tested to ensure a good specification quality. If code generation is used the model represents the implementation as well and will be tested thoroughly. Software testing is performed to verify that the software implementation fulfils the specification. If code generation is performed, software testing ensures the correct transformation performed by the code generator from model to software. Integration testing of hardware and software is always performed to verify if the integrated system works correctly.
For MiL and SiL testing MESSINA supports software devices like MATLAB/Simulink models, ASCET models and AUTOSAR software components (Fig. 1 system under test with dotted lines). Multiple models and software components can be run in parallel to perform virtual system integration.
For HiL testing MESSINA is directly connected with modularHiL. It is possible to download tests implemented with MESSINA to the modularHiL where the tests are executed in real-time (Fig. 1 system under test).
As a result tests implemented in MESSINA can be used seamlessly in the model test (MiL), software test (SiL) and hardware test (HiL). Therefore, MESSINA could be used for thorough model-based electronic control unit (ECU) testing from specification to HiL testing. Since tests are defined hardware independent MESSINA ensures a high portability of the tests between different test environments. The only difference between HiL and SiL from MESSINA point of view is the usage of a different environment model (e.g. for MiL and SiL using a software ABS component, for HiL using a real hardware ABS component). Therefore test cases can be used without any further adaption for MiL, SiL, and HiL testing.
Recurrent test steps can be defined as templates in a test library and be reused for defining test sequences. Test variants can easily be created by parameterized generic test cases.

Evolutionary Testing Framework
The EvoTest [6] project implements an extensible and open Automated Evolutionary Testing Architecture and Framework that provides general components and interfaces to facilitate the automatic generation, execution, monitoring and evaluation of test cases using evolutionary computation. The evolutionary testing framework creates an evolutionary algorithm suitable for the system under test using GUIDE [8, 9]. For this, a problem description has to be provided to the framework, which is analysed by GUIDE to generate an evolutionary algorithm best suited to optimize the testing problem automatically. During the optimisation process the evolutionary algorithm generated provides the individuals representing test data for the system under test and expects the fitness values for each individual back. On basis of the fitness values the next generation of individuals is generated. Details of the evolutionary testing framework are described in [7].

Integration of Test Automation Framework
The integration of the EvoTest evolutionary testing framework with MESSINA has been implemented for the automatic generation of test cases. For this, MESSINA has been extended by a PlugIn for configuring and running the evolutionary test.
The PlugIn implements the communication with the evolutionary testing framework, controls the test execution for the generated tests and returns the fitness values calculated for the individuals on basis of the test execution results back to the evolutionary testing framework.
The signals of the system under test held in the signal pool of MESSINA containing type information, value range information etc. are passed to the evolutionary testing framework as individual descriptions. The evolutionary testing framework then creates a specific evolutionary algorithm exactly fitting to this

description using GUIDE [8, 9]. The use of mixed data types is supported by MESSINA and the evolutionary testing framework. This allows in many cases a one-to-one transformation of individuals into test data. The individuals created by the evolutionary testing are executed with the system under test by MESSINA. Generic test cases are parameterized with the individual data.

For MiL and SiL tests MESSINA calls the system under test directly with the parameterized test cases, for HiL testing MESSINA downloads the tests to modularHiL allowing real-time execution of the tests on the electronic control unit.

For the fitness function calculation the behaviour of the system under test has to be analyzed for each test case executed. MESSINA allows recording the behaviour of the system under test for the generated test data sets through monitoring interfaces. The fitness function calculation is implemented in the generic test case, and the fitness function values passed back to the evolutionary testing framework.

The MESSINA run-time system allows a comfortable execution and remote debugging of test cases directly on the target system.

With the implemented solution search based testing could be used for thorough model-based ECU testing from specification to HiL testing. Since tests are defined hardware independent MESSINA ensures a high portability of the tests between the different test platforms. The only difference between HiL, MiL and SiL testing is the usage of different environment models.

4 Evaluations

To evaluate the implemented test automation framework an evolutionary functional test for an anti-lock braking system has been realized. The anti-lock braking system used for the tests is already in serial production.

Anti-lock braking system (ABS)

An anti-lock braking system is a system which prevents the wheels of a vehicle from locking while braking, in order to allow the driver to maintain steering control under heavy braking and, in most situations, to shorten braking distances. ABS is very effective at braking in adverse weather conditions like ice, snow or rain. When ABS equipped brakes are depressed hard - like in an emergency braking situation - the ABS system pumps the brakes several times per second. Sensors measure the speed at which the wheels are turning. If the speed decreases rapidly, the electronic control system reports blocking danger. The pressure of the brake hydraulics is reduced immediately and then raised to just under the blocking threshold. This process can be repeated several times per second. The goal of the anti-locking control system is to maintain the slip of the wheels at a level which guarantees highest braking power and highest steer ability of the vehicle.

Evolutionary Testing of the ABS

For testing the anti-lock braking system a hardware-in-the-loop test environment has been set up using the test automation framework described including modularHiL. In addition, a proper simulation environment for the ABS system has to be created to simulate the system environment of the ABS system in such a manner that the anti-lock braking system detects no difference to its use in a vehicle. In case the anti-lock braking system detects improper behaviour of the system environment it automatically changes to a failure mode making realistic testing impossible. Therefore, a complex simulation environment was implemented integrating commercial brake models, vehicle dynamics models and wheel speed sensor models from Tesis [10]. Fig. 2 shows the components of the simulation environment implemented.

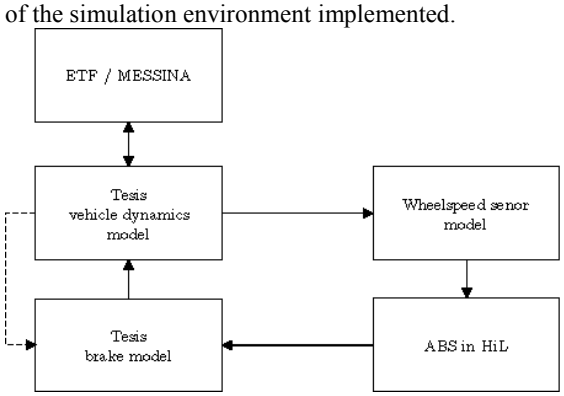


Figure 2: For testing the ABS system complex environment simulations are needed to simulate the wheelspeed sensor inputs for the ABS, the behaviour of the brakes and the vehicle dynamics

Test Results

In our first experiments the main focus lies on the evaluation of the developed test environment. The goal is to show its functioning for the fully automatic functional testing of complex electronic control units. Future experiments will deal with a thorough testing of the ABS functionality itself. Therefore, we use a simplified fitness function for the testing of the ABS system. The fitness of an individual – representing a braking manoeuvre – is calculated on the basis of the resulting braking distance of the vehicle. The fitness function maps the braking distance directly to the fitness value. Longer breaking distances result in higher fitness values indicating weaker system performance. In the test we maximize the fitness value. The search is configured using the GUIDE default parameters presented in [12]. The population size was set to 20 individuals due to the long execution times for braking manoeuvres simulated in real-time and to obtain first results on the principal functioning of the testing framework quickly.

In our experiments testing the ABS system with the test automation framework maximising the braking distance using evolutionary testing worked well. In 20 repetitions of the experiment the maximum possible

braking distance of 43 meters was always found within 10-12 generations. In accordance the vehicle velocity was maximised to 25ms^{-1}, the fastest possible speed in our model. Fig. 3 shows the results achieved by one sample test run.

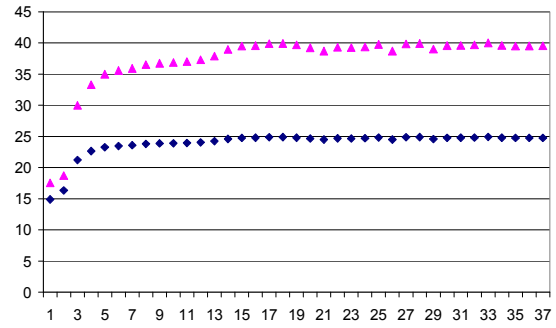


Figure 3: Fastest Vehicle Speed (♦, [$^{m}/_{s}$], y-axis) and Resulting Longest Braking Distance (▲, [m], y-axis) for a Complete Stop of the Vehicle Found in each Generation (x-axis) of the Evolutionary Testing

Real-time test execution for the individuals with the longest braking distance took up to 40 seconds on modularHiL; including speeding up the car to the maximum possible speed and afterwards performing the braking. The total execution time for all generations of one test run took up to 90 minutes. The scalability of the test execution is limited due to the real-time simulation necessary for the correct functioning of the ABS system. However, test cases resulting in the longest braking distance of the ABS system under the constraints applied for the experiment were found with high reliability fully automatically. The complexity of the search space used in the experiment was limited, so that search techniques such as hill climbing or even random testing might yield good results, too. A more realistic test of the ABS system will evaluate the slip withdrawn by the system with respect to different situations detected by the wheel speed sensors and result in search spaces of higher complexities no longer tack able with simple search techniques.

5 Conclusions and Future Work

In this work a test automation framework has been developed on basis of commercial testing products (modularHiL, MESSINA) and research prototypes (EvoTest's evolutionary testing framework) that allows full automation of functional testing on different testing platforms (MiL, SiL, HiL) by applying search based testing techniques. The provided solution supplements systematic testing by reducing the risk of non-testing situations unforeseen by the testers. The test automation framework supports the application of evolutionary testing in very common industrial settings: testing models and software in simulation environments as well as the examination of electronic control units in a hardware-in-the-loop test environment driven by a software-frontend for the definition and implementation of tests. To evaluate the test automation framework testing an ABS system formed the first case study. Test cases were generated using the evolutionary testing framework and executed in SiL and HiL test environments. The search for interesting testing scenarios was successful: driving manoeuvres with long braking distances were found fully automatically proving that functional evolutionary testing is possible and feasible.

Our next step will be a realistic test of the ABS system searching for errors in the system. For this, a fitness function representing the overall slip produced by the system under various difficult circumstances will be defined. Individuals will be transformed into realistic curve traces for the input signals of the ABS system using the signal generator developed by Windisch [11]. To cope with the higher complexity, search space reduction techniques [14] will be applied and the population size will be increased, leading to longer execution times of the test runs.

Acknowledgements

This work is supported by EU grant IST-33472 (EvoTest).

References

[1] Wegener, J.; Baresel, A.; Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology, 43(1):841-854, 2001.

[2] Buehler, O.; Wegener, J.: Evolutionary functional testing. Comput. Operations Research. 35 (10), 3144-3160, 2008.

[3] modularHiL: http://berner-mattner.com/automotive-modulhil.php

[4] MESSINA: http://berner-mattner.com/automotive-messina.php

[5] TPT: http://piketec.com/products/tpt.php?lang=en

[6] EvoTest: http://www.evotest.eu/

[7] Dimitar, M.; Dimitrov, I. M.; Spasov, I.: Evotest - Framework for customizable implementation of Evolutionary Testing. International Workshop on Software and Services, October 2008, Sofia, Bulgaria.

[8] GUIDE. http://guide.gforge.inria.fr/

[9] Da Costa, L.; Schoenauer, M.: GUIDE, a Graphical User Interface for Evolutionary Algorithms. GECCO Workshop on Open-Source Software for Applied Genetic and Evolutionary Computation (SoftGEC), 2007.

[10] Tesis veDYNA. http://www.tesis.de/en/index.php?page=544

[11] Windisch, A.: Search-Based Testing of Complex Simulink Models containing Stateflow Diagrams. Proceedings of the 1st International Workshop on Search-Based Software Testing, Lillehammer, Norway, 2008.

[12] Luis Da Costa and Marc Schoenauer: Description of Evolution Engine Parameters, http://guide.gforge.inria.fr/eeparams/EEngineParameters.html

[13] Automotive Standards Committee of the German Institute for Standardization. ISO/WD 26262: Road Vehicles – Functional Safety. Preparatory Working Draft, Technical Report, October 2005.

[14] Harman, M.; Hassoun, Y.; Lakhotia, K.; McMinn, P.; Wegener, J.: The Impact of Input Domain Reduction on Search-based Test Data Generation. Proceedings of the 6th European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 155-164, 2007.

[15] Lefticaru, R.; Ipate, F.: Automatic State-Based Test Generation Using Genetic Algorithms. Proceedings of the Ninth Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing., pp. 188-195, 2007.

A Testability Transformation Approach for State-Based Programs

AbdulSalam Kalaji, Robert M Hierons and Stephen Swift
School of Information Systems, Math and Computing
Brunel University, Uxbridge, UB8 3PH, UK
{abdulsalam.kalaji, rob.hierons, stephen.swift}@brunel.ac.uk

Abstract

Search based testing approaches are efficient in test data generation; however they are likely to perform poorly when applied to programs with state variables. The problem arises when the target function includes guards that reference some of the program state variables whose values depend on previous function calls. Thus, merely considering the target function to derive test data is not sufficient. This paper introduces a testability transformation approach based on the analysis of control and data flow dependencies to bypass the state variable problem. It achieves this by eliminating state variables from guards and/ or determining which functions to call in order to satisfy guards with state variables. A number of experiments demonstrate the value of the proposed approach.

1. Introduction

Errors in software can lead to undesired outcomes and testing is therefore a crucial stage. However, manual testing is expensive, error-prone and time consuming hence automation is very desirable.

SBT approaches such as evolutionary testing (ET) [1] have received attention due to their efficiency in deriving test data automatically, however, their applications were largely focused on structured programs where the input domain of a test target is explored to select a set of input values according to a given test criterion e.g., statement coverage. The exploration is steered by evaluation information represented by a *fitness function*. For example, Wegener et al. [2] described a fitness function (Equation 1) in the presence of nested IF statements that comprises two components: a *branch distance* [3] and the *approach level* (Equation 2) to measure how close a particular input was to executing the target branch that is missed and how many critical nodes are away from the target respectively. The critical node is a branching node at which the path control flow may divert (see Fig. 1). Since it is necessary to contrast how many conditions were achieved by a specific input, the branch distance of each IF statement is normalized to a value in the range of [0..1] (Equation 3).

$$fitness = approach\ level + norm\ (branch\_distance) \quad (1)$$

$$approach\_level = 1 - NumCriticalNodeFromTarget \quad (2)$$

$$norm\ (branch\_distance) = 1 - 1.05^{-branch\_distance} \quad (3)$$

The existence of state variables in the presence of function calls can cause problems when using SBT approaches. The main effect is that the fitness function is unable to direct the search towards the desired input values. Thus, the performance of an SBT approach is likely to degenerate to that of random search.

In the literature, some techniques, cited in [4], studied the problem of test data generation from subjects with state behavior. However, an efficient and easy test data generation approach remains a requirement. Thus, the aim of this paper is to benefit from the efficiency and flexibility introduced by testability transformation (TeTra) approaches [5] and reformulate the state variable problem as a TeTra problem. Applying TeTra to a program with state behavior was recently highlighted by Harman [4] as an open research problem.

The approach presented in this paper aims to address the problem described as: Given a target function with state variable problems, transform the test target so that an ET approach can automatically generate a set of test data that exercises this function

The primary contributions of this paper are the following: (1) It proposes a TeTra approach to bypass state variables problems. (2) It provides a method to suggest when a TeTra is likely to be required. (3) The approach can be generally applied to similar problems in a program with functions and global variables.

2. The Proposed Approach

The approach of Wegener et al. [2] described in the previous Section is efficient when a given target function is independent e.g., it is not control or data dependent on other previous functions. Nevertheless,

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.14

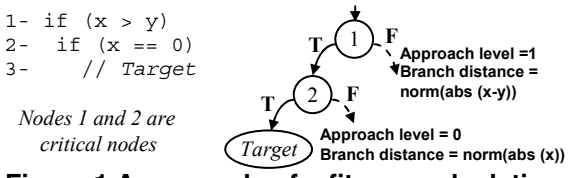


Figure.1 An example of a fitness calculation

when such a dependency exists, it is not always possible to consider only the target function.

In order to apply TeTra, we classify an assignment statement *op* to a variable *v* in a function *f* to four types: { op^{vp}, op^{vv}, op^{vc}, $op^{v\pm c}$} which denote that *v* is assigned a value that depends on a parameter, another variable, a constant and itself and a constant respectively. Also, we classify a guard *g* with a guard operator $gop \in \{<, >, \neq, =, \leq, \geq\}$ in an *f* to five types: {g^{pc}, g^{pp}, g^{pv}, g^{vv}, g^{vc}} which denote a comparison among: parameters and constant, parameters only, parameters and variables, variables only and variables and constant respectively. Based on the above classifications, we can distinguish two types of functions: affecting and affected-by functions.

Definition 1: In a given program with *n* functions, f_i is an affecting function within this program if f_i has an $op \in \{op^{vp}, op^{vc}, op^{vv}, op^{v\pm c}\}$ to *v* and there exists a guarded function f_j, where $0 \leq i < j \leq n$, f_j has a guard $g \in \{g^{vv}, g^{vc}\}$ over *v* and the statements at *op* in f_i and *g* in f_j form a definition-use (*du*) pair for *v*.

Definition 2: An f_j is an affected-by function within a program if f_j has $g \in \{g^{vv}, g^{vc}\}$ over *v* and there exists an affecting function f_i, where $0 \leq i < j \leq n$, over *v* and the statements at *op* in f_i and *g* in f_j form a *du* pair for *v*.

From Definition 2, an affected-by f_j is always data dependent on the corresponding affecting f_i. f_j is also control dependant on f_i if f_i has a guard that controls its assignment operations that affect f_j. Based on Definitions 1 and 2, we can distinguish four cases in which a target function requires a transformation.

Case 1: the problem occurs between a pair of affected-by and affecting functions (f_j, f_i) where *op* in $f_i \in \{op^{vv}, op^{vc}\}$ and f_j is control dependent on f_i.

(1) Program fragment	(2) Transformed version

`//Case study 1`
`int x,y;`
`void reset()`
`{x = 0; y = 0;}`
`void t1 (int P1,P2)`
`{if (P1==0) x = 10;`
`if (P2==0) y = 10;}`
`void target ()`
`{if (x>=10 && y>=10)`
`//Target 1 }`
	`//Case study 1- TeTra`
	`int x,y;`
	`void reset()`
	`{ x = 0; y = 0;}`
	`void target (int P1,P2)`
	`{if (P1==0){x =10;`
	`if (P2==0){y =10;`
	`if (x>=10 && y>=10)`
	`//Target 2`
	`}}}`

(3) Fitness plot of Target 1	(4) Fitness plot of Target 2

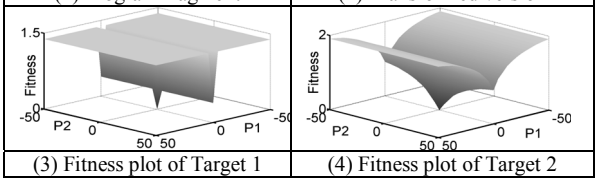


Figure.2 TeTra applied to the first case

(1) Program fragment	(2) Transformed version

`//Case study -2-`
`int x,y;`
`void reset()`
`{ x = 0; y = 0; }`
`void t1 (int P1)`
`{if (P1 != 0) x= 100;`
`else x= P1;}`
`void t2 (int P2)`
`{if (P2 != 0) y=100;`
`else y = P2;}`
`void target()`
`{if (x ==0 && y ==0)`
`//Target 1 }`
	`//Case study 2 -TeTra`
	`int x,y;`
	`void reset()`
	`{ x = 0; y = 0;}`
	`void target(int P1,P2)`
	`{`
	`if (P1 ==0 && P2 ==0)`
	`//Target 2`
	`}`

(3) Fitness plot of Target 1	(4) Fitness plot of Target 2

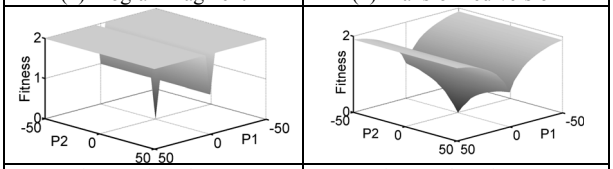


Figure.3 TeTra applied to the second case

Fig. 2 shows a case study that describes how TeTra is applied. The target function (*target*) is control dependent on task *t1* which has guards that control its assignments (op^{vc}). A sequence of calls to *reset*→*t1*→*target* does not necessarily achieve the *target* guards (g^{vc}). The fitness landscape of the original program is plotted in Fig. 2-3. Due to the flat region of this landscape, the search does not receive adequate information and relies only on chance to hit the target. Fig. 2-2 shows the transformed version of the *target* task. Since the assignments of *t1* are controlled by its input parameters, these are required on the *target* task. Also, the true branches of *t1* predicates are considered since they lead to assigning variables *x* and *y* the required values. Now, the fitness landscape of Target 2 (see Fig. 2-4) has a clear downward surface and provides adequate guidance.

Case 2: The problem occurs in a pair of affected-by and affecting functions (f_j, f_i) where *op* in $f_i \in \{op^{vp}\}$ and f_j is control dependent on f_i.

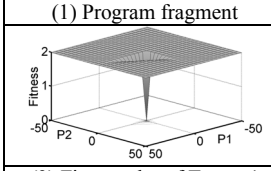
Compared to Case 1, this case has the input parameters of the affecting function referenced by the state variables that appear in the target function guards. Fig. 3 shows a case study in which a *target* task is affected by two functions *t1* and *t2* and these functions have guards that control their assignments. Furthermore, the input parameters of *t1* and *t2* are referenced by the state variables in the *target* task guards. A sequence of calls to *reset*→*t1*→*t2*→*target* does not always lead to the *target* task being exercised. Fig. 3-3 shows that the fitness landscape of the original program is flat. The transformed version of the *target* task is shown in Fig-3-2. The input parameters and the assignment enabling predicates of *t1* and *t2* are

```
//Case study 3
int x,y;
void reset()
{x = 0; y = 0;}
void t1 (int P1,P2)
{if (P1 >= 0){
  x=x+P1*2; y=y+P1;}
 else{x=100; y=100;}
 if (P2 >= 0){
  x=x+P2*3; y=y+x;}
 else{x=100; y=100;}}
void target()
{if (x==0 && y==0)
  //Target 1 }
```

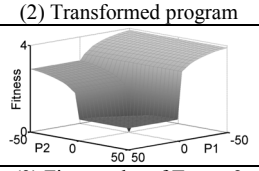
(1) Program fragment

```
//Case study 3 - TeTra
int x,y;

void reset()
{ x = 0; y = 0;}

void target (int P1,P2)
{
if (P1 >= 0){
 if (P2 >= 0){
  if (P1*2 + P2*3==0){
   if (P1*3 + P2*3==0){
       //Target 2
}}}}}
```

(2) Transformed program



(3) Fitness plot of Target 1



(3) Fitness plot of Target 2

Figure.4 TeTra applied to the third case

embedded in the transformed version. Also, the state variables of the original *target* task are replaced by the input parameters that they reference. As observed in Fig. 3-4, the fitness landscape of Target 2 provides the search with enough guidance.

Case 3: The problem exists between a pair of affected-by and affecting functions (f_j, f_i) where *op* in $f_i \in \{op^{vv}, op^{vp}, op^{vc}\}$ and f_j is control dependent on f_i.

This case can be seen as a generalisation of Case 1 and Case 2. However, the main difference is that the affecting function assignments are complicated by many types of assignments. Consequently, bypassing the state variables in this case is not a straight forward process. This problem can be transformed by applying amorphous slicing [6] for the state variables of the target function.

Fig. 4-1 presents a case study in which the *target* task has two state variables which are assigned values in *t1*. A sequence of calls to *reset*→*t1*→*target* is unlikely to solve the problem. The original fitness landscape plotted in Fig. 4-3 is almost flat and provides insufficient guidance. In Fig. 4-2, the transformation is applied to replace the state variables *x* and *y* by expressions that reference parameters. The fitness landscape of the transformed version provides the search with adequate guidance as shown in Fig. 4-4.

Case 4: The problem occurs between a pair of affected-by and affecting functions (f_j, f_i) where *op* in $f_i \in \{op^{v\pm c}\}$ and f_j is control dependent on f_i.

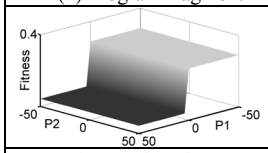
This problem is likely to exist when a state variable in the target function has the role of a counter. For such a case, it is necessary to determine which and how many calls to be made to other functions before calling the target one. Fig. 5-1 shows a case study of two functions where the *target* task is control dependent on the affecting *t1*. As shown in Fig. 5-3, the fitness

```
//Case study 4
int x,y;
void reset()
  { x = 0; y = 0;}
void t1 (int P1,P2)
{if (P1 >= 0)
   x=x+1; else x =0;
 if (P2 >= 0)
  y=y+1; else y=100;}
void target ()
{if (x>=10 && y>=10)
   //Target 1 }
```

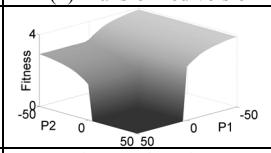
(1) Program fragment

```
//Case study 4 - TeTra
int x,y;
void reset()
  { x = 0; y = 0;}

void target (int P1,P2)
{if (P1 >= 0){
  if (P2 >= 0){
   //Target 2
for(i=1; (x<10 || y<10)
;i++)
  t1(P1,P2); }}}
```

(2) Transformed version



(3) Fitness plot of Target 1



(4) Fitness plot of Target 2

Figure.5 TeTra for finding a feasible path

landscape does not touch the zero surface and so the original scenario: *reset*→*t1*→*target* is infeasible. The transformed version shown in Fig. 5-2 tries to construct a feasible path that enables the *target* task to be triggered. Since the input parameters of the affecting *t1* decide whether the assignments are executed, these are included in the transformed version. Once suitable input parameters values are found, a loop of calls is made to the affecting *t1* assignments. The notion of implementing a loop to perform the necessary calls to an affecting function is introduced in [4]. The number of the loop cycles (number of calls) can be determined by reversing the guards of the *target* task. For this case study, this is determined as: loop while (x<10 OR y<10). Similarly, a logical connector AND is reversed to OR and guard operators: $\{<, \leq, >, \geq, =, \neq\}$ are reversed to: $\{\geq, >, \leq, <, \neq, =\}$. Once the affecting functions, the number of calls, and the suitable input parameters values are determined, a feasible path is constructed from the original code by repeatedly calling the affecting functions with the same suitable input parameters values. Fig. 5-4 shows a clear downwards fitness landscape of the transformed version for finding the suitable input parameter values to be applied to *t1*.

Table 1 lists all possible combinations among affected-by and affecting functions. The fields marked by *R* indicate the cases where we conjecture that TeTra is likely to be required. Fields marked by *F/R* indicate that the transformation is only required if the scenario is feasible and fields marked by *N* identify the cases where the transformation is not necessary.

3. Experimental Study and Conclusion

Experiments were performed on the four case studies presented in this paper by using random search

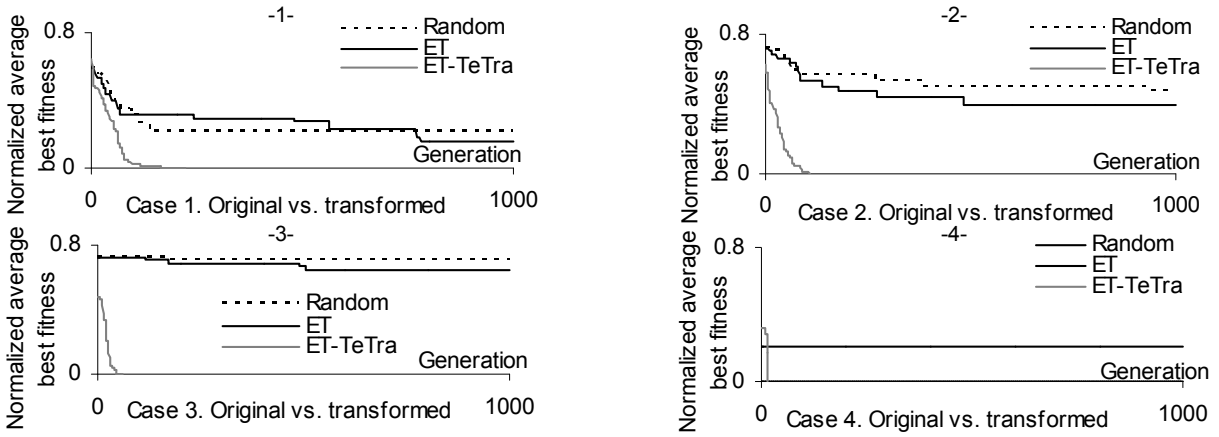


Figure 6: Results of random, ET and ET after TeTra was applied on the four case studies

and the standard ET approach described in Section 1. Although the four case studies are small in terms of code size, the complexity for a search-based algorithm is not related to the code size but it is a function of the search space size [7]. The input domain size used with the four case studies had 4×10^6 possible candidate solutions. Both of the standard ET and random approaches were implemented with the publicly available *GEATbx* [8]. The population size was 50 individuals with two variables in the range [-1000..1000]. The ET methods were: linear-ranking with 1.8 selective pressure, discrete recombination and mutate integer. The search was terminated after 1000 generations or if the objective value of zero was achieved. Finally, each search was repeated 10 times.

Fig. 6 plots the performances of random and ET approaches on each case study before applying the transformation and once again the ET performance after the transformation was applied. Each plot shows the normalized best achieved fitness yet in a specific generation for a particular search approach averaged over ten repetitions of the experiment.

Fig. 6-1, 6-2 and 6-3 plot the performances of the search approaches on Case study 1, 2 and 3 respectively. From these plots, we observe that ET and random searches exhibited relatively similar performance before applying the transformation and they failed to exercise the test target. In contrast, the ET search on the transformed version was successful and hit the target relatively quickly. Fig. 6-4 corresponds to the last case study. Since the untransformed version corresponds to an infeasible path, it was not surprising that ET and random searches both failed. However, the ET performance on the transformed version was very fast in locating the required input values. The empirical results demonstrate that the proposed TeTra approach was effective in improving and enhancing the ET technique. Further research will apply the approach to additional examples and investigate its use to derive feasible paths for the purpose of model-based testing.

Table. 1 Suggesting when TeTra is required

Guard & operator (affected-by)	Assignment (affecting)			
	(op^{pv})	(op^{vv})	(op^{vc})	($op^{v\ast c}$)
g^{pc}, g^{pp}, g^{vp} (=, <, >, ≤, ≥, ≠)	N	N	N	N
g^{vv}(=, <, >, ≤, ≥, ≠)	R	R	R	R
g^{vc}(=, <, >, ≤, ≥, ≠)	R	R	F/R	R

4. References

[1] P. McMinn, "Search-based software test data generation: a survey: Research Articles," *Software Testing, Verification & Reliability*, vol. 14, pp. 105-156, 2004.

[2] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, pp. 841-854, 2001.

[3] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," presented at Automated Software Engineering, Proceedings. 13th IEEE International Conference on, pp. 285-288, 1998.

[4] M. Harman, "Open Problems in Testability Transformation," presented at Software Testing Verification and Validation Workshop, ICSTW '08. IEEE International Conference on, pp. 196-209, 2008.

[5] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *Software Engineering, IEEE Transactions on*, vol. 30, pp. 3-16, 2004.

[6] M. Harman and S. Danicic, "Amorphous program slicing," presented at Program Comprehension, IWPC '97. Proceedings, Fifth Iternational Workshop on, 1997.

[7] P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to Search-Based test data generation," in Proceedings of the 2006 international symposium on Software testing and analysis. Portland, Maine, USA: ACM, pp. 13-24, 2006.

[8] H. Pohlheim, "GEATbx - Genetic and Evolutionary Algorithm Toolbox for Matlab," 1994-2008. http://www.geatbx.com.

Searching for Rules to find Defective Modules in Unbalanced Data Sets

D. Rodríguez
Dept. of Computer Science
University of Alcalá
Alcalá de Henares, Madrid, Spain
e-mail: daniel.rodriguezg@uah.es

J.C. Riquelme
Dept. of Computer Science
University of Seville
Seville, Spain
e-mail: riquelme@us.es

R. Ruiz, J.S. Aguilar-Ruiz
Dept. of Computer Science
Pablo de Olavide University
Seville, Spain
e-mail: {robertoruiz, aguilar}@upo.es

Abstract

The characterisation of defective modules in software engineering remains a challenge. In this work, we use data mining techniques to search for rules that indicate modules with a high probability of being defective. Using data sets from the PROMISE repository[1], we first applied feature selection (attribute selection) to work only with those attributes from the data sets capable of predicting defective modules. With the reduced data set, a genetic algorithm is used to search for rules characterising modules with a high probability of being defective. This algorithm overcomes the problem of unbalanced data sets where the number of non-defective samples in the data set highly outnumbers the defective ones.

1. Introduction

The characterization of defective software modules, which has been addressed from different perspectives, remains a challenge in the software engineering field. Recently there is an increasing interest in applying search-based techniques to relevant aspects, such as effort estimation, defect prediction and maintainability [3]. With the availability of repositories with actual data from software projects, such as the PROMISE repository [2], it is now possible to apply data mining techniques to learn from real data. In this work, we use a genetic algorithm as subgroup discovery technique to characterise defective modules from data sets contained in such repository. However, for this problem, we need to tackle two problems: (i) data sets are highly unbalanced and (ii) there are attributes (metrics) that are not relevant for predicting defective modules. The background is summarised as follows:

Feature Selection. A large number of attributes can also interfere in the data mining learning process. In theory the more attributes, the more information capable of discriminate defective modules. In practice, however, this does not hold as many attributes are redundant or irrelevant degrading the accuracy rate. The problem of feature selection received a thorough treatment in pattern recognition and data mining [7]. As stated previously, feature selection is used to identify the most relevant attributes from a data set and to remove those redundant and/or irrelevant attributes that have negative effect on the data mining learning algorithm. Feature selection is part of the data preparation phase (pre-processing) to generate a reduced data set which can be useful in different aspects:

- A reduced volume of data facilitates the application of different data mining or searching techniques to be applied. Furthermore, data mining algorithms can be executed faster with smaller data sets.
- Irrelevant and redundant attributes can generate less accurate and more complex models which are harder to understand.
- Knowing which data is redundant or irrelevant can be used to avoid data collection of those attributes in the future so that the data collection becomes more efficient and less costly.

Unbalanced Data Sets. Most data sets in defect prediction are highly unbalanced, i.e., samples of non-defective modules vastly outnumber the cases of defective modules. This is a problem that affects most data mining learning algorithms which aim at classifying correctly the maximum number of instances assuming that data are balanced. For example, a model which always selects the majority class when, for example this class represents 90% of the samples, already produces very good results so modifications to the algorithm to improve the accuracy results are generally discarded. With unbalanced data sets, data mining learning algorithms produce degenerated models that do not take into account the minority class as most data mining algorithms assume balanced data sets. When this happens, there are two alternatives, either (i) to apply algorithms that are robust to unbalanced data sets or (ii) balance the data using sampling techniques before applying the data mining algorithm .

Subgroup Discovery. Initially proposed by Klösgen [6] and Wrobel [9], [10], subgroup discovery are a set of algorithms that extract rules or patterns for subsets of the data of a previously specified the concept, for example defective modules in this work. The idea is to search for properties of

1. http://promisedata.org/

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.23

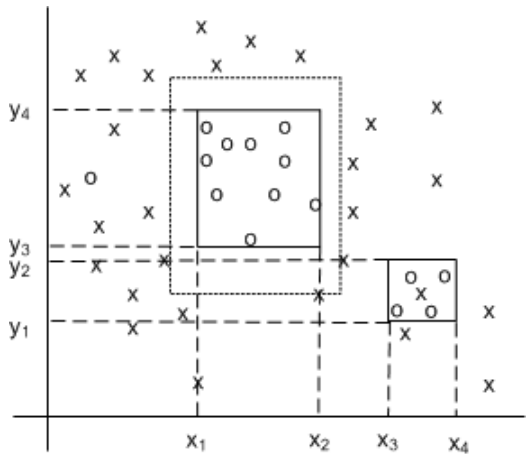


Figure 1. Examples of Rules with 2 Dimensions

subgroups with different behaviour in relation to the rest of the data. Rules for subgroup discovery have also the "*Condition* → *Class*" where the *condition* is the conjunction of a set of selected variables (pairs attribute–value) among all variables. In this work, we have used an evolutionary algorithm for subgroup discovery adapted from an algorithm to discover hierarchical rules [1]. This algorithm generates rules covering samples representing defective modules (the minority class). In this way, the algorithm is able to handle unbalanced data sets. Figure 1 shows an example of rules generated in this work with 2 dimensions. With the condition $x_1 \leq X \leq x_2$ AND $y_3 \leq Y \leq y_4$, we can consider the inner rectangle or the outer one. Also, instead of having too many rules covering perfect areas with no errors, we need a trade-off between the number of rules and number of erroneous instances included in those rules.

2. Experimental Work

In this paper, we have used the CM1, KC1, KC2, and PC1 data sets available in the PROMISE repository [2], to generate models for defect classification. These data sets were created from projects carried out at NASA and collected under their metrics[2]. Table 1 shows the number of instances (modules) for each data set together with the number of defective, non-defective and their percentage showing that all data sets are highly unbalanced varying from 7% to 20%. According to their Web site, the term module is applied to the lowest level functional unit from which metrics can b collected such as (functions, modules, or subroutines). The last attribute is the programming language used to develop those modules.

All data sets contain the same 22 attributes composed of 5 different lines of code measure, 3 McCabe metrics [8], 4 base Halstead measures [5], 8 derived Halstead measures [5], a branch-count, and the last attribute is 'problems' with 2 classes (false or true, wether the module has reported defects). Table 2 summarizes the metrics collected from the data sets.

2. `http://mdp.ivv.nasa.gov/`

Table 1. Data Sets used in this Work

	instances	*Non-def*	*Def*	*% def*	*Lang*
CM1	498	449	49	9.83	C
KC1	2,109	1,783	326	15.45	C++
KC2	522	415	107	20.49	C++
PC1	1,109	1,032	77	6.94	C

These sets of metrics (both McCabe and Halstead) have been used for quality assurance during (i) development to obtain quality measures, code reviews, etc., (ii) testing to focus and prioritize testing effort, improve efficiency, etc. and (iii) and maintenance as indicators such as comprehensibility of the modules or to detect error prone modules.

As stated previously, we are interested in rules for characterizing error prone modules. Generally, the developers or maintainers use rules of thumb or threshold values to keep modules, methods, etc within certain ranges. For example, if the cyclomatic complexity of a module is between 1 and 10, it is considered to have a very low risk; however, any value greater than 50 is considered to have an unmanageable complexity and risk. For the essential complexity (`ev(g)`), the threshold is 4, etc. Although these metrics have been used for long time, there are no clear thresholds for most of them and furthermore, they are open to interpretation. For example, although McCabe suggest a threshold of 10 for `v(g)`, NASA in-house studies of this metric concluded that a threshold of 20 is a better predictor of defective modules. Table 3 shows the number of times an attribute was selected by the feature selection algorithm as well as the range of the attribute and typical thresholds suggested in the literature.

As stated previously, the aim of this work is to search for rules that provide an indication of defective modules. To do so, we first performed a feature selection process to the data sets in order to simplify the input before applying the subgroup discovery algorithm (searching rule algorithm). In this work, we have used the Correlation-based Filter Selection [4] (CFS) as feature selection technique. The CFS is applied to the data before any other data mining algorithm and independently of them. This algorithm is in turn another searching algorithm that selects a set of attributes highly correlated with the class attribute and a low grade of redundancy between them. The CFS filter applied to the original CM1, KC1, KC2 and PC1 data sets are shown in Table 4.

Then, rules were generated using a genetic algorithm capable of handling unbalanced data sets. As a result, there is no need of applying any sampling techniques by either

Table 2. Attribute Definition Summary

	Metric	*Definition*
McCabe	loc	McCabe's Lines of code
	v(g)	Cyclomatic complexity
	ev(g)	Essential complexity
	iv(g)	Design complexity
Halstead base	uniq_Op	Unique operators, n_1
	uniq_Opnd	Unique operands, n_2
	total_Op	Total operators, N_1
	total_Opnd	Total operands N_2
Halstead derived	n	Vocabulary
	l	Program length
	v	Volume
	d	Difficulty
	i	Intelligence
	e	Effort
	b	Error Estimate
	t	Time estimator
	loCode	Count of statements
	loComment	Count of lines of comments
	loBlank	Count of blank lines
	loCodeAndComment	Code and comments
Branch	branchCount	No. branches
Class	false, true	Reported defects

Table 3. No. of Times Selected Attributes were Used by the Rules and Thresholds Suggested in the Literature

Attribute	*# of times*	Range	Threshold
loc	2	0-∞	60
iv(g)	4	1-v(g)	7
i	13	0-∞	120
loBlank	12	0-∞	10
uniq_op	3	0-∞	20
uniq_opnd	12	0-∞	20
v	1	0-∞	1,500
d	8	0-∞	30
loCode	8	0-∞	30
loComment	13	0-∞	10
branchCount	7	0-∞	19
ev(g)	3	1-v(g)	4
b	3	0-∞	0.60

Table 4. Attributes Selected for each Data Set

CM1	*KC1*	*KC2*	*PC1*
loc	v	ev(g)	v(g)
iv(g)	d	b	i
i	i	uniq_Opnd	loComment
loComment	loCode		loCodeAndComment
loBlank	loComment		loBlank
uniq_Op	loBlank		uniq_Opnd
uniq_Opnd	uniq_Opnd		
	branchCount		

increasing artificially the number of instances of the minority class artificially (in our case, number of cases with defective samples) or removing samples form the majority class. Also, the generated rules are simpler than the ones generated by other data mining techniques such as C4.5. This algorithm is a variant of another algorithm called HIDER [1]. The difference is that while HIDER generates rules for all the values of the class and those rules have to be applied hierarchically (rules have to be applied in order), the variant used in this work focuses only on one value of the class, i.e., the minority class[3].

The generated rules combine a set of attributes (metrics) to provide better estimation and explanation of defective modules. As it can be expected, most of the selected attributes were used by the rules and only in the case of the PC1 data set, the `v(g)` attribute was selected by the feature selection algorithm but not included in any of the generated rules. Table 3 shows the number of times that selected attributes were used in the condition of the rules to cover defective data sets as well as their possible range and thresholds suggested in the literature. The number of times an attributes are used reveals some interesting outcomes that need further research. For example, it seems quite natural that the intelligence of a module (i) or the number of unique operands influence the possibility of error prone modules, however it is less obvious that attributes such as $loBlank$ or $loComment$ (lines of blanks and lines of comments respectively) can be used as good predictors of defective modules.

Also, the disparity of the selected attributes in each data set may be the consequence of feature selection algorithm not being able to handle unbalanced data sets appropriately. The analysis of feature selection attributes under this condition is part of our future work. In any case, some attributes seem to be consistently more relevant than others as a predictors of defective modules.

Table 5 shows only the first rule for each data set which is the one covering the largest number of defective modules. The accuracy of each rule can be measured by the number of defective modules and non-defective modules covered by the rule space, taking into account that there is an upper limit to the number of defective modules. There is a trade-off between the number of rules, number of defective and non-defective samples covered by each rule.

In this work, in all four data sets around one third of the defective modules were covered by the rules and there is no defective samples covered by more than a single rule.

One problem observed, however, is that each rule seems to captures a small number of defective modules in proportion to the total number of them. If we consider that attributes of a data set, in a large n-dimensional space, rules cover small islands of points representing the minority class (defective

3. Implementation available at:
`http://www.ieru.org/wiki/index.php5?title=Software`

Table 5. Main Decision Rule for each Data Set

Data Set	*Rule Condition*
CM1	$37.27 \leq$ i AND
	$27 \leq$ loBlank ≤ 32 AND
	uniq_Op ≤ 32
KC1	$24{,}33 \leq$ d AND
	i ≤ 59.7 AND
	$24 \leq$ uniq_Opnd AND
	$12 \leq$ branchCount ≤ 31
KC2	$5 \leq$ ev(g) AND
	$37 \leq$ uniq_Opnd
PC1	$15 \leq$ loComment ≤ 71 AND
	$85 \leq$ uniq_Opnd

modules) which seem to be located sparsely and surrounded by non-defective modules. We considered a low value (5%) as an acceptable error when searching for rules; such value generated a large number of rules covering few instances. In software engineering, it could make more sense not to focus on the accuracy but to generate rules that are able to provide an indication of defective modules even if a rule covers as many defective modules as non-defective. The large number of rules and attributes for each rule make it hard to analyse and provide results for *human consumption*. The variance and number of selected attributes can also be affected by the fact that datasets are highly unbalanced; although our selected attributes are similar to others found in the literature for the same datasets, this needs further research.

3. Conclusions and Future Work

In this work, we have applied data mining techniques to characterize defective modules to four data sets from the PROMISE repository. To do so, we faced the problem that data sets are highly unbalanced, i.e., there are many more non-defective samples in the data sets than defective ones which conditions the range of techniques that we can apply and moreover, reduces the accuracy of the results. To solve this problem, we applied feature selection as a necessary step to reduce the data sets and then, as a subgroup discovery technique, a genetic algorithm as a subgroup discovery technique was used to generate rules for covering only defective modules.

Results showed that in general data sets are not very homogeneous in both the feature selection (attributes) selected in each data set or rules generated. The results, however, provide some points for further research. In relation to the rules we need to improve the algorithm to obtain a small number of simpler rules than the ones obtained. One approach can be to increase the percentage of error allowed for each rule. This will increase the number of instances covered per rule but also the error rate. In this way, rules will cover many more defective modules but rules will be simpler and could be used as indicators instead of almost certainty of being defective. We also need further research about the quality of the data sets. We found a large number of inconsistencies in the data sets used in this work. For example, datasets have replicated instances and contradictory instances (same values for all attributes but different class value). We believe that in general in software engineering datasets are of poor quality compared to other disciplines and this needs to be further analysed. Finally, we are only considering a binary class, it should be possible to consider more classes such as low, medium or high problematic modules.

Acknowledgements

Authors would like to thank to the anonymous reviewers for their useful comments. This research was supported by the Spanish Research Agency (TIN2007–68084–C02–00) and the Universities of Seville, Pablo de Olavide and Alcalá.

References

[1] Aguilar-Ruiz, J.S., Riquelme, J.C., Toro M., Evolutionary Learning of Hierarchical Decision Rules, *IEEE Transactions on Systems, Man and Cybernetics*, Part B, Vol 33, No. 2, pp. 324-331

[2] Boetticher G., Menzies T., Ostrand T.; PROMISE Repository of empirical software engineering data. (`http://promisedata.org/`), West Virginia University, Department of Computer Science (2007)

[3] Harman M., Jones B.F., Search-based software engineering, *Information & Software Technology*, Vol. 43, No. 14, pp. 833–839, 2001

[4] Hall M.A.; Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning. In: 17th Int. Conf. on Machine Learning, 359–366 (2000)

[5] Halstead, M., *Elements of Software Science*. Elsevier, 1977.

[6] Klösgen, W., Explora: A Multipattern and Multistrategy Discovery Assistant. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padraic Smyth, and Ramasamy Uthurusamy, editors, Advances in Knowledge Discovery and Data Mining, pages 249-271. AAAI Press, 1996.

[7] Liu H. and Yu L., Toward Integrating Feature Selection Algorithms for Classification and Clustering, IEEE Trans. on Knowledge and Data Eng., 17(3), 1–12, 2005.

[8] McCabe, T. J.; A complexity measure, *IEEE Transactions on Software Engineering* 2 (4), 308–320, 1976.

[9] Wrobel, S. An algorithm for multi-relational discovery of subgroups. In J. Komorowski & J. Zytkow (Eds.), Proc. First European Symposion on Principles of Data Mining and Knowledge Discovery (PKDD-97) (pp. 7887). Springer Verlag, 1997.

[10] Wrobel, S. (2001). Inductive logic programming for knowledge discovery in databases. In S.Deroski & N. Lavra (Eds.), Relational data mining. Springer-Verlag.

Formal model simulation: can it be guided?

Thang H. Bui and Albert Nymeyer
School of Computer Science and Engineering,
The University of New South Wales, Australia
{buih,anymeyer}@cse.unsw.edu.au

Abstract

In this work we present an approach to system development that bridges the complementary worlds of model checking and simulation. This approach, which we refer to as formal model simulation*, uses the same formal model as model checking, but instead of using a total search of the state space, we use a* guided *random-walk based search to find errors. The guide we use is a heuristic that can be derived from an abstract model of the system. We have implemented the technique in a tool called* granSPIN*, which is derived from the popular model checker* SPIN*. A series of experiments is outlined in this work using different selection strategies and different heuristics. We compare the performance of these different strategies and heuristics in* granSPIN *and* SPIN *on a buggy Peterson protocol.*

1 Introduction

Model checking is a verification technique used to guarantee the correctness of a model of a system with respect to some desired property. Model checking can guarantee the absence of certain errors in a model. Testing through simulation only partially verifies correctness. Being only partial, no amount of simulation in general will guarantee correctness. However, simulation dominates industrial development of systems. One reason for this is that no matter how large and complex a system is, it is always possible to carry out simulation, unlike model checking, where users are inevitably confronted by time/space limitations, and even intractability. These limitations do not stop software developers using a model checker as a debugger, however, as one of the major advantages of using model checking is that it not only finds errors, it can tell the user the behaviour that led to the error, often referred to as a 'witness'. This feature is invaluable for debugging of course. But using a verification tool to debug a system can be unwieldy because of the heavy demand model checkers can make on resources. This is because of the poor scalability of model checking, and means that developers may spend large amounts of time grappling with time/space problems instead of the design. The poor scalability is a consequence of the *state-explosion problem*. In fact, a model checker may not return a result: it may neither verify the model, nor find an error, and hence leave the user 'empty-handed'. For this reason, simulation is often seen as a better, more practical, and scalable alternative.

A *formal model simulator* uses the same formal model and property specification as a model checker. Only after the model simulator cannot find any property violations (in reasonable time) need the model checker be employed. In fact, in our work, we use precisely the same heuristic mechanism in both the (random-walk based) simulator and (guided) model checker to improve the efficacy of the search. The aim of the work we present here is to show this. Note that we do not describe in any detail where the heuristics that we use come from, or what constitutes a good heuristic, although we are interested in what happens when we use deliberately incorrect heuristics. There are two reasons for this. The (automated) mechanism that we use to generate a heuristic has been reported in previous work, and that work provides the starting point of this research ([8, 7]). We do discuss this below. The second reason is that we have developed heuristics by hand in this work to study particular effects.

Adding a guide to a model checker is well known [3, 8]. To understand the guided paradigm used here, it is useful to briefly recant how the heuristic that is used by the search algorithm in our conventional guided model checker is normally generated. In effect, the given formal model of a system is statically analysed to determine which variables are 'weakest' in the sense that they are 'lowest' in a data dependency analysis. These variables are then removed from the model, resulting in a model called an abstract model. We ensure the abstract model is tractable (i.e. small enough) for a conventional reachability analysis of the model's state

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.19

space. We can then compute how far every abstract state is from any (potential) abstract error states in the state space (these abstract error states correspond to the concrete goal states, defined by the temporal property). Using the above distances as cost heuristic, we then apply an A*-based guided model checker (called GOLFER) to the concrete model. GOLFER is a symbolic model checker that is derived from NUSMV[1]. Given a formal (concrete) model of a system and a temporal property, it is always possible in principle to analyse the model, build an abstract model, generate a heuristic, and to use this heuristic to search for errors in the concrete model. In practice, it is not known what method of analysis produces the best heuristic, and indeed, what constitutes a good heuristic. All heuristics have some degree of 'informedness', which is a term used to denote the level of knowledge of a heuristic. A well-informed heuristic is one that provides a 'strong' cost heuristic that drives the search. A badly-informed heuristic in one that cannot provide any real assistance, and in a conventional model checker, the result is a search algorithm that is basically breadth-first. Note that, although guided model checking can be much faster than a conventional model checker, its worst-case behaviour (which happens when the heuristic is badly informed) is the same.

Random walks also have been used before to debug models [5, 6, 4]. In essence, using a random-walk based search means abandoning verification, and instead carrying out just a partial search over the state space. Being random, no user involvement is required. In earlier work [1], we describe how we replaced the standard depth-first search algorithm in SPIN[2] by a random-walk based search algorithm. In subsequent work [2] we ported the abstract-model heuristic mechanism described above and used in GOLFER to our random-walk version of SPIN. The resulting tool we have called GRANSPIN. We report here on experiments that we have performed with GRANSPIN that investigate more deeply aspects of the guided strategy such as the method of selecting states during the search and the informedness of the heuristic. Note however that for experimental purposes we have generated the heuristics by hand in this work.

Adding guidance to the random-walk algorithm complicates the issue of selecting successor states during a search in the state space. We in fact use quite different successor selection strategies for unguided and guided random walks. If the random walk is unguided, then a successor state is selected by randomly selecting a transition, and testing whether it is executable. If it is executable, then we have a successor state, if not, then we delete this transition and randomly select another transition. We call this the *try-a-child* strategy. In contrast, if the random walk is guided, all the successor states were first computed and ordered according to their heuristic value. From this list a selection is made. We name this the *all-children* strategy. This strategy is crude, and clearly less efficient because of the additional effort required to compute every successor (child) every time.

Using these different selection strategies, and different heuristics, experiments have been carried out comparing SPIN, and guided and unguided GRANSPIN. We focus in particular on the effect of using heuristics with different degrees of 'informedness', and as a worst-case, ask the question *what happens if the heuristic is bad!*. In Section 2, we present our guided, random-walk based algorithm that underpins GRANSPIN. The experiments are discussed in Section 3. Conclusions are presented in the last section.

2 Guided Random-Walk Algorithm

Generally, a random-walk algorithm traverses the state space of a given system using random walks, starting at some initial state. If a random walk reaches an accept (goal) state, we stop the search and return that walk as a *goal path*. There are two main issues that need to be resolved when using a random-walk algorithm: when do we terminate each random walk, and when do we terminate the algorithm (i.e. how many random walks should we do)? We refer to these as the *end-walk condition* and *termination condition* resp. In Figure 1, we show our guided random-walk based model-checking algorithm. This algorithm has been derived from the algorithm in [2], and adds the try-a-child mechanism to both guided and unguided random-walk search.

```
GRANSPIN algorithm
Input: set of processes P, never-claim pnc,
       Boolean guided, Boolean trychild
Output: goal path | not found
1:  while (! terminationCond)
2:    s = init()
3:    path = ∅
4:    while (! endCond)
5:      s = ranMove(P, s, guided, trychild)
6:    s = move(pnc, s)
7:    path = path ∪ {s}
8:    return path  when (accepted(pnc, s))
9:  return not found
```

Figure 1. The guided random-walk algorithm

[1] http://nusmv.irst.itc.it

[2] http://spinroot.com

In the function *ranMove* in line 5, the value of the Boolean parameter *guided* determines whether the guided or unguided version of the algorithm is performed. There is a second Boolean parameter, *trychild*, which determines which selection strategy is used. If *trychild* is true, the algorithm uses the try-a-child strategy to select a successor state, otherwise, it uses the all-children strategy. In the **try-a-child strategy**, a transition is selected arbitrarily to execute. If the transition is unexecutable, another transition is selected. The generated state is the next successor state in the walk. We can use this strategy in either guided or unguided algorithms, which we refer to as ranSPIN$_T$ and granSPIN$_T$ resp. in our experiments. In the **all-children strategy**, all transitions are executed to generate all candidate successor states. The next successor state is selected among the candidates. We use the names ranSPIN$_A$ and granSPIN$_A$ to denote the algorithms without and with a guide.

3 Experimentation

We applied the 4 variants of the tool (i.e. 2 different selection strategies, with and without guide) to a buggy version of the well-known Peterson protocol[3]. In our experimental results, each data point in the graph (representing the execution time) is the average of 50 repetitions of the experiment, where each repetition uses a different seed in the random-number generator. All the experiments were performed on a Pentium IV 3.0GHz with 1GB RAM.

3.1 Heuristic generation

A heuristic is generated only once (at start-up time). In the experiments below we have generated heuristics by hand and have not factored their creation into the time of course. Normally heuristics are generated automatically in negligible time.

We analysed each process in the given model (consisting of many processes) and generated a 'local distance' table containing a set of 'local' states (of a given process) and their distances to the 'local' goal state (a similar method is used in [3]). These local distances form the heuristic. Whenever we require a heuristic for a state during the search, we use the distance of the corresponding local state of the single process. Furthermore, we could 'abstract' a model consisting of many processes by simply removing all but two of the processes. A breadth-first search algorithm can then be used on this cut-down model to generate a shortest goal path to an error state. The goal path is then examined (manually) and a heuristic is generated. (This process could be automated but is out of scope of this work.) We call this heuristic **h1**.

In Figure 2, we show the results for the **Peterson protocol**. The table on the top of this figure shows the number of processes **pr.**, execution time **t** (in milliseconds), number of trials before the goal path **n**, length of the goal path **cx**, and memory usage in megabytes (**m**) for each variant of granSPIN. The graph on the bottom of the figure plots the execution times. In the figure, *oom* means 'out of memory'. granSPIN uses the heuristic **h1**, ranSPIN is unguided of course. The results show that SPIN performs very poorly, and that granSPIN is faster than ranSPIN by half an order of magnitude, and produces shorter goal paths. The reason for this is that ranSPIN attempts many more random walks than granSPIN before finding the goal path, as you would expect as it is unguided. For example, for 11 processes ranSPIN$_T$ required 8.4 walks while granSPIN$_T$ required 1.9.

pr.	SPIN				ranSPIN t				ranSPIN a				granSPIN t				granSPIN a			
	t	cx		m	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m
3	0	6519		2.7	6	1.0	2631.1	2.5	10	1.1	2798.5	2.5	1	1.0	1083.4	2.4	1	1.0	1047.5	2.4
5	73020	9993		839.2	42	2.4	3319.1	2.7	72	2.4	3561.0	2.8	17	1.3	2936.8	2.5	15	1.2	2437.5	2.5
7	*oom*				96	4.7	3258.8	3.4	190	5.1	2705.2	3.5	25	1.4	3693.3	2.6	20	1.2	2998.3	2.6
9					189	7.2	4698.8	4.4	337	6.7	4565.9	4.3	45	1.8	4101.8	2.8	51	1.4	4989.4	2.7
11					222	8.4	6997.5	5.3	366	7.0	6920.2	4.9	52	1.9	4898.0	2.9	98	2.2	5873.8	3.1

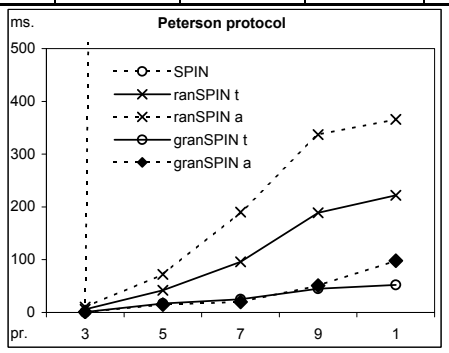


Figure 2. Comparing SPIN, ranSPIN and granSPIN

3.2 What happens if the heuristic is bad?

The experiments above that compare guided and unguided random walks do not tell us how dependent the results are on using the right heuristic. To address this issue, we re-ran the previous experiment with other heuristics. The heuristic **h1** from the previous section abstracted the concrete system into a system consisting of two processes. In this section we study also a *courser* abstraction consisting of just a single process. We named this heuristic **h0**. We also study so-called *mis-informed* heuristics, which try to guide the search *away* from the error state. We create these heuristics by reversing the supposedly 'good' heuristic that we used in the previous experiment. This is actually quite straightforward to implement. Whenever a 'good' heuristic produces a distance, or cost, h for a successor

[3] `http://anna.fi.muni.cz/benchmark`

state, we replace that cost by $maxDist - h$. This means the algorithm biases the search away from goals.

So we now have 4 heuristics: **h0** and **h1**, and their reversals **h0 rev** and **h1 rev**. Note, crucially, that **h1** is better informed than **h0** because **h1** is a finer abstraction (i.e. it throws less information away). Because **h1** is better informed, it should lead to better performance than **h0**. Conversely, however, the corresponding reversed heuristic **h1 rev** should lead to worse performance than **h0 rev** because **h1 rev** more aggressively guides the random walks in the wrong direction. Expressed another way, because **h1** is better informed than **h0**, the difference in performance between **h1** and **h1 rev** should be greater than **h0** and **h0 rev**. The results presented in Figure 3 confirm this expectation. For better readability, we have labelled the columns *unguided* (which is ranSPIN), *guided h0*, *guided h0 rev*, *guided h1* and *guided h1 rev*.

pr.	unguided				guided h0				guided h0 rev				guided h1				guided h1 rev			
	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m	t	n	cx	m
3	6	1.0	2631.1	2.5	5	1.0	2149.1	2.5	11	1.1	3845.6	2.5	1	1.0	1083.4	2.4	0	1.0	389.6	2.4
5	42	2.4	3319.1	2.7	56	2.4	3458.0	2.7	93	3.4	3981.8	2.9	17	1.3	2936.8	2.5	5	1.0	2135.8	2.5
7	96	4.7	3258.8	3.4	117	4.1	3802.5	3.3	185	6.4	3162.1	3.8	25	1.4	3693.3	2.6	20	1.1	5175.2	2.6
9	189	7.2	4698.8	4.4	249	6.6	4277.3	4.2	419	10.8	4293.2	5.4	45	1.8	4101.8	2.8	285	7.5	5266.4	4.5
11	222	8.4	6997.5	5.3	342	8.5	6937.6	5.3	459	11.5	6866.3	6.4	52	1.9	4898.0	2.9	409	10.6	6191.4	6.1

Figure 3. Comparing unguided and guided ranSPIN using 4 heuristics.

One surprise was that the performance of ranSPIN with even a 'bad' heuristic was better than SPIN. The reason is that the heuristic only acts as a guide; the random-walk search is still predominantly random. Hence, even if the heuristic is 'bad', the random-walk search is still more effective than a conventional exhaustive search.

4 Conclusions

In this work we have presented formal model simulation using a guided random-walk based model-checking algorithm. The technique has been implemented in the tool ranSPIN, which is derived from SPIN. In a series of experiments, we have compared SPIN with ranSPIN. In particular, we studied the performance of ranSPIN using two kinds of heuristic strategies, called 'try-a-child' and 'all-children'. We also tried two 'levels' of abstraction in the heuristic, one which is course-grained, the other finer-grained; and we reversed these heuristics to produce so-called mis-informed heuristics. The aim here is to study the robustness of the guided random-walk technique when compared to a conventional model checker. Overall we expect the speed-up that can be expected from using guided search to be directly related to the informedness of the heuristic. The more informed, the faster, and conversely, the more mis-informed, the slower. Experiments have confirmed these expectations.

We have carried out experiments on other (small) protocol case studies and found similar behaviour. Nevertheless, it is not known if the behaviour that we have observed will apply to industrial-scale models as well. While a static analysis can and should be used to automatically generate a heuristic, it is not clear how informed the resulting heuristic will be. Many kinds of static analysis are possible, but how does one measure informedness? This is future work.

References

[1] T. H. Bui and A. Nymeyer. ranSPIN: A random-walk based model checker. In *Int. Workshop on Advanced Computing and Applications: ACOMP'08*, pages 38–48, Mar. 2008. In print.

[2] T. H. Bui and A. Nymeyer. The spin on guided random search in verification. *ICSTW: IEEE Int. Conf. on Software Testing Verification and Validation Workshop*, 0:170–177, 2008.

[3] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF–SPIN. In *Proc. of the* 8^{th} *Int. SPIN Workshop*, volume 2057 of *LNCS*, pages 57–79. Springer, May 2001.

[4] R. Grosu and S. A. Smolka. Monte Carlo model checking. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems, TACAS'05*, volume 3440 of *LNCS*, pages 271–286. Springer, 2005.

[5] D. Owen, B. Cukic, and T. Menzies. An alternative to model checking: verification by random search of AND/OR graphs representing finite-state models. In 7^{th} *IEEE International Symposium on High-Assurance Systems Engineering, HASE'02*, pages 119–128. IEEE Computer Society, Oct. 2002.

[6] D. Owen, T. Menzies, M. Heimdahl, and J. Gao. On the advantages of approximate vs. complete verification: Bigger models, faster, less memory, usually accurate. In *Proc. of IEEE/NASA Softw. Eng. Workshop, SEW'03*, pages 75–81, 2003.

[7] K. Qian and A. Nymeyer. Abstraction-based model checking using heuristical refinement. In *Proc. of* 2^{nd} *Int. Symp. on Automated Technology for Verification and Analysis, ATVA'04*, volume 3299 of *LNCS*, pages 165–178. Springer-Verlag, 2004.

[8] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Proc. of the* 10^{th} *Int. Conf. on TACAS*, volume 2988 of *LNCS*, pages 497–511. Springer-Verlag, 2004.

How Can Optimization Models Support the Maintenance of Component-Based Software?

Vittorio Cortellessa, Pasqualina Potena
Dipartimento di Informatica
Università dell'Aquila
Via Vetoio 1, 67010, Coppito (AQ), Italy
{cortelle,potena}@di.univaq.it

Abstract

The maintenance phase of software systems is ever more increasing its incidence, in terms of effort, to the whole software lifecycle. Therefore the introduction of automated techniques that can help software maintainers to take decision on the basis of quantitative evaluation would be a suitable phenomenon. Search-based techniques offer today a very promising view on the automation of searching processes in the software engineering domain. Component-based software is a very interesting paradigm to apply such type of techniques, for example for component selection. In this paper we introduce optimization techniques to manage the problem of failures at maintenance time. In particular, we introduce two approaches that provide maintenance actions to be taken in order to overcome system failures in case of monitored and non-monitored software systems.

Keywords: Component-Based Software, Software Maintenance, Optimization model, Software Cost, Software Reliability.

1 Introduction

Due to the short time to market of software projects, combined with continuously changing requirements on software products, the maintenance of a software system is becoming one of the most complex activities of the whole lifecycle.

Software applications are often deployed before their validation and verification tasks have been completed, with the obvious consequences that software does not behave at run-time as expected from functional and non-functional viewpoints, thus leading to failures.

In this paper we show how an *unexpected* system failure can be overcome. *Unexpected* means that, on the basis of the certified reliability of the components, a failure shall not occur so early. Under the assumption that there is exactly one faulty component we propose how to maintain a system both in case it is under monitoring and in case it is not monitored.

In the first case (called *failure with awareness*, see Section 3.1) the faulty component is detected by monitoring the system. Then the number of additional test cases to be performed on it in order to bring its reliability to the expected value are estimated by using a reliability growth model proposed in [1].

In the second case (called *failure without awareness*, see Section 3.2) no specific monitoring action is devised and the faulty component originating a failure cannot be identified. Therefore in this case, after a software failure occurs, our approach searches for a different system configuration (e.g. by replacing some components) that minimizes the costs while raising the system reliability by a fair amount that (hopefully) allows in future to avoid unexpected failures.

The paper is organized as follows: in Section 2 we describe some related work and summarize the novelty of our approach, in Section 3 our maintenance solution is presented and finally in Section 4 conclusions are provided.

2 Related work

Many approaches have been introduced for supporting typical maintenance activities, such as reconfiguration (see, for example, [4, 6]).

Search-Based Software Engineering (SBSE) [7] research area is spreading its scope to different domains of the software development process. However, still few approaches have been introduced to drive decisions in the component-based software paradigm. In particular, optimization techniques have not been yet largely used for the maintenance phase, although search-based approaches have been introduced in [8] for software refactoring.

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.22

Optimization techniques might solve in this paradigm some drawbacks of the Analytic Hierarchy Process (AHP) and the Weighted Scoring Method (WSM) that are typically used in component selection approaches. In fact, "both methods come with serious drawbacks such as the combinatorial explosion of the number of pair-wise comparisons necessary when applying the AHP, the need of extensive a priori preference information such as weights for the WSM, or the highly problematic assumption of linear utility functions in both cases" [9].

The original contribution of this paper is a systematic approach to the problem of determining, at the maintenance phase, the actions that suitably overcome an unexpected failure of a software system. This problem has never been tackled in literature, at the best of our knowledge, with a model-based approach. Even though our optimization model is similar to the one in [3], we use it here for supporting decisions at maintenance phase. The results provided by the model solution induces corrective actions that the previous model cannot suggest. Another contribution is to tackle the problem in two cases with different techniques: monitored and non-monitored scenarios.

3 Searching low cost maintenance solutions

We start from a deployed component-based software that has been assembled following the architectural approach introduced in [3], whose notation is summarized here below.

Let $S =< C_{ij} >$ be a software architecture made of n components ($i = 1...n$), where C_{ij} is one of the functionally equivalent instances of the component i. C_{ij} can be either an in-house built instance ($j = 0$) or a COTS component instance ($j = 1...|J_i|$, where J_i is the set of available COTS for component i).

S has been somehow built at minimum cost while assuring (among other) a system reliability greater than the threshold R. In [3] an optimization model has been introduced (based on models that combine, respectively, cost, reliability and delivery time of components to make cost, reliability and delivery time of the whole system) to determine an optimal solution to the problem, where a solution is expressed as a selection of component instances, one for each architectural component. In addition the model solution provides the amount of testing to carry on each in-house instance to achieve the desired reliability.

Our reliability model does not take into account dependencies between components. Such assumption is common to many reliability approaches (see, for example, [5]). What is basically neglected under this assumption is the error propagation probability, which in several real domains (such as control systems) is not an issue because component errors are straightforwardly exposed as system failures.

3.1 Failure with awareness

The reliability of each component instance can be either provided from the vendor in case of COTS components or estimated with a reliability growth model [11] in case of an in-house built instance.

We remark that the case of a faulty COTS component can be easily solved through searching, by enumeration, if any other available COTS is reliable enough to satisfy the reliability constraint. The cost of such a COTS, if it exists, represents the cost of the maintenance action. We rather focus on the case of a faulty in-house built instance that is computationally more complex.

The reliability growth model that we have adopted [1] makes the failure probability θ_i of a component instance ([1]) depending on: (i) the number N_i^{suc} of successful (i.e. failure-free) tests executed, (ii) the probability p_i that the instance is faulty, and (iii) the testability $Testab_i$ [1], as follows:

$$\theta_i = \frac{Testab_i \cdot p_i(1 - Testab_i)^{N_i^{suc}}}{(1 - p_i) + p_i(1 - Testab_i)^{N_i^{suc}}} \quad (1)$$

In Figure 1 a graphical representation of such model is provided while varying the number of tests on the x-axis. Let us assume that the figure refers to the faulty component in this scenario.

The curve labelled with the $(p_i^{est}, Testab_i)$ pair of parameters represents the function that we have considered to assemble the deployed system. In particular, the point **A** is the component reliability θ_i^{est} estimated with the executed amount of tests N_i^{est}.

Upon the failure occurrence, in case of awareness, we not only are able to identify the faulty component, but also to quantify its experimented failure probability. This can be done by simply counting the number of successful invocations of this component before the failure occurred. Let us denote by θ_i^{exp} this experimentally measured value that (as shown in the figure) must be higher than θ_i^{est} by definition of unexpected failure. In other words, the experimental observation of the system at work reveals that N_i^{est} tests were not enough, as estimated, to bring the failure probability of the component to θ_i^{est}, but only to θ_i^{exp} in practice. Ultimately we were expecting to be in point **A** of Figure 1, while we have observed to be in point **B**.

This information helps to adjust the reliability estimation as follows. Given that the testability is an intrinsic characteristic of a component that is mainly due to the component complexity, we address the θ_i^{est} to θ_i^{exp} gap to a wrong estimate of p_i^{est}. By using the collected data, and without moving the testability, it is possible to obtain the refined value

[1] Failure probability and reliability of a component sum to one, thus they are considered equally powerful metrics.

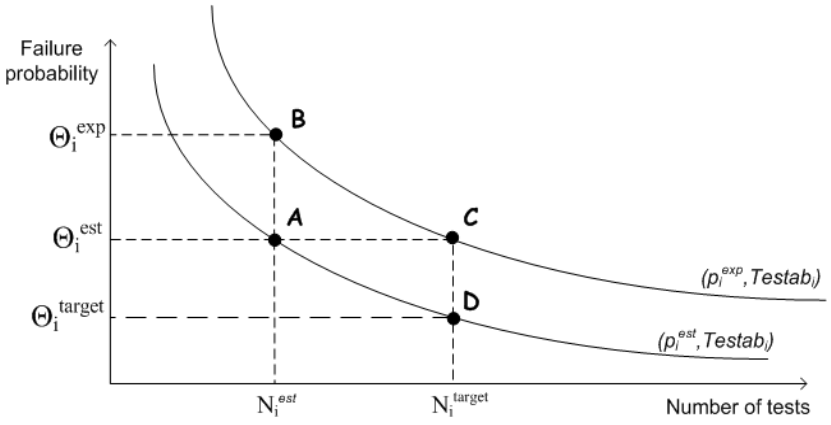


Figure 1. Reliability and number of tests.

p_i^{exp} of the probability to be faulty by using equation (1) as follows:

$$p_i^{exp} = \frac{\theta_i^{exp}}{\theta_i^{exp} + (Testab_i - \theta_i^{exp})(1 - Testab_i)^{N_i^{est}}} \quad (2)$$

Basing on these new values we are able to draw a more realistic reliability growth model, that is the curve labelled with the $(p_i^{est}, Testab_i)$ pair in Figure 1. Our goal is to use this new model to find out how many additional tests are necessary to bring the component failure probability to the desired level, that is the original θ_i^{est}. We therefore need to move, along the new model, to point **C** of Figure 1 to find out that the originally needed number of tests to achieve such failure probability is N_i^{target} ([2]).

The difference $\Delta N = N_i^{target} - N_i^{est}$ represents the amount of testing necessary on the faulty component to bring in practice its reliability to the desired level. It is therefore easy, once obtained ΔN, to calculate the cost and time for this solution if the unitary cost and time of testing are known.

3.2 Failure without awareness

In this sub-scenario we assume that no specific monitoring action is devised on the working system, therefore the faulty component that originates an unexpected failure cannot be identified. Here no maintenance action on a specific component can be devised like in Section 3.1, thus we introduce an optimization model to search a reconfiguration of the whole architecture while the whole system reliability threshold is raised by a certain amount R to R' and it is required to complete this maintenance action within a time limit T'.

The optimization model can be formulated in similar way to the one presented in [3]. It searches for a reconfiguration of the system architecture while raising the reliability threshold required for the whole system. The basic idea is to raise the whole system reliability threshold by a certain amount, to determine a delivery time, and to solve the optimization model that, given the current solution S, provides a new solution (i.e. a new set of components) that minimizes the reconfiguration cost.

The model solution may suggest either to replace some component i with a COTS instance or to perform on it a number z_i of test cases if it has been in-house developed (i.e. $z_i = max\{0, N_i^{tot} - Ntest_i\}$, where N_i^{tot} is an integer decision variable that represents the total number of tests performed on the instance and $Ntest_i$ is the number of test cases performed when the component has been built).

The structure of this model differs from the one presented in [3] mainly for the number z_i of test cases suggested for the component i at the maintenance phase. In the objective function the cost to perform such additional testing is taken into account (i.e. the cost of the in-house i is equal to $\bar{c}_i(t_i + \tau_i z_i)$, where $\bar{c}_i$ represents the unitary development cost, t_i the estimated development time, and τ_i the average time required to perform a test case). Besides, the delivery time of an in-house instance is modeled as a function of the additional testing as well (i.e. $t_i + \tau_i z_i$).

In order to properly parameterize the model appropriate values have to be provided to R' and T' ([3]). In this sub-scenario without awareness there is no reference to target for those values, therefore only an incremental study can be conducted to analyze the corresponding trend of the total cost.

However, for sake of completeness we have highlighted

[2] Note that this value can be obtained by solving equation (2) for N_i^{target} in place of N_i^{est} and θ_i^{est} in place of θ_i^{exp}. It is $N_i^{target} = log_{(1-Testab_i)} \frac{\theta_i^{est}(1-p_i^{exp})}{(Testab_i - \theta_i^{est})p_i^{exp}}$.

[3] Estimation techniques for other parameters have been discussed in [3].

in Figure 1 the ideal reliability target for this sub-scenario. In fact, if we would know (like in the sub-scenario with awareness) that N_i^{target} is the correct number of test cases to perform in order to achieve the desired reliability, then the target reliability θ_i^{target} of component i is achieved in point **D** with the original model. Therefore, the ideal target for the reliability threshold R' would be in this case the one obtained by using θ_i^{target} as the failure probability of component i. However, this mechanism cannot be applied in this case because there is no awareness of the faulty component and no monitoring mechanism is introduced.

4 Conclusions

In this paper we have showed how to overcome unexpected system failures of a component-based software basing on optimization models. Our models combine the non-functional characteristics of software components (such as cost, reliability, delivery time) to make the characteristics of the whole system. This type of modelling is a sound basis to study the best decisions to take when something new happens in the system. In this case our approach allows to estimate the impact that maintenance decisions (such as testing more extensively a component or buying a set of new COTS components) may have on the characteristic of the whole system. We retain that such type of instruments can be very important in the hands of software engineers, as they allow to estimate/compare different solutions on the basis of specific metrics such as the software cost.

This is only a first work in the direction of using optimization models for component-based maintenance. This domain presents a multitude of problems. In this paper we have only tackled the problem of the failure of exactly one component, whereas specific approaches are needed to face the more complicated scenarios of failure of more components. Also, the problems to meet new requirements or to improve the quality of a software release without offering new functional requirements can be faced with optimization models.

We plan to apply our approaches on realistic examples in order to validate it and to study the approach scalability. In cases where the computation time becomes too large (e.g. because the number of available components becomes huge), heuristic approaches based on algorithms for set covering (see chap. 6 in [10]) or more general metaheuristic techniques (e.g. the tabu-search algorithm, [2]) can be also considered.

Finally, we intend to implement tools that can automatically generate the optimization model by annotating system models (such as UML diagrams) with the appropriate parameters (e.g. reliability value of the components) like, for example, our tool CODER makes for the model that we have introduced for the software architectural phase (see [3]).

5 Acknowledgments

The authors would like to thank Ivica Crnkovic for the interesting discussions and collaboration that have inspired this work. They also are grateful to Fabrizio Marinelli for his great support in the construction of optimization models.

References

[1] Bertolino, A. and Strigini, L., "On the use of testability measures for dependability assessment", *IEEE Trans. on Software Engineering*, 22(2):97-108, 1996.

[2] Blum C., Roli A.: "Metaheuristics in Combinator Optimization: Overview and Conceptual Comparison", *ACM Computing Surveys*,vol.35, n.3, 2003.

[3] Cortellessa, V., Marinelli, F., Potena, P.: "Automated Selection of Software Components Based on Cost/Reliability Tradeoff", *Proc. of EWSA06*, LNCS 4344, 66-81, 2006.

[4] David, P.C., Leger, M., Grall, H., Ledoux, T., Coupaye, T: "A Multi-stage Approach for Reliable Dynamic Reconfigurations of Component-Based Systems", *Proc. of DAIS 2008*, LNCS 5053, 2008.

[5] Goseva-Popstojanova, K., Trivedi, K.S: "Architecture based approach to reliability assessment of software systems", *Performance Evaluation*, vol. 45 (2001), 179-204.

[6] Grassi, V., Mirandola, R., Sabetta, A.: "A model-driven approach to performability analysis of dynamically reconfigurable component-based systems", *Proc. of 6th ACM Workshop on Software and Performance*, 2007.

[7] Harman, M.: "The Current State and Future of Search Based Software Engineering" *Proc. of 29th ICSE, Future of Software Engineering (FoSE)*, 2007.

[8] Harman, M., Tratt, L.: "Pareto optimal search based refactoring at the design level" *Proc. of GECCO 2007*.

[9] Neubauer, T., Stummer, C.: "Interactive Decision Support for Multiobjective COTS Selection" *Proc. of 40th Hawaii International Conference on System Sciences*, 2007.

[10] Reeves, C.R.:"Modern heuristic techniques for combinatorial problems", John Wiley & Sons, Inc. New York, NY, USA.

[11] Trivedi K., "Probability and Statistics with Reliability, Queuing, and Computer Science Applications", J. Wiley and S., 2001.

WCET Analysis of Modern Processors Using Multi-Criteria Optimisation

Usman Khan, Iain Bate
Department of Computer Science, University of York, York, United Kingdom
Email: usmannmkhan@hotmail.com, iain.bate@cs.york.ac.uk

Abstract

The Worst-Case Execution Time (WCET) is an important execution metric for real-time systems, and an accurate estimate for this increases the reliability of subsequent schedulability analysis. Performance enhancing features on modern processors, such as pipelines and caches, however, make it difficult to accurately predict the WCET. One technique for finding the WCET is to use test data generated using search algorithms. Existing work on search-based approaches has been successfully used in both industry and academia based on a single criterion function, the WCET, but only for simple processors. This paper investigates how effective this strategy is for more complex processors and to what extent other criteria help guide the search, e.g. the number of cache misses. Not unexpectedly the work shows no single choice of criteria work best across all problems. Based on the findings recommendations are proposed on which criteria are useful in particular situations.

1 Introduction

The study of real-time systems incorporates all systems which have to respond within a fixed, finite time and where a delayed answer is as bad as a wrong response. More formally, a real-time system is one where "*correctness depends not only on the logical result of the computation, but also on the time at which the results are produced*" [4]. Thus, timeliness is a crucial aspect of a real-time system. One essential measure for all forms of schedulability analysis is the WCET. WCET research has proceeded in two distinct directions.

The method of *static analysis* aims to analyse the hardware and software under test statically i.e. without executing the software, and then use this information to derive an estimate for the WCET. Thus, given a processor architecture and the program to be executed, static analysis works by analysing execution paths and simulating processor characteristics to determine the worst-case path in a program [16, 10]. The worst-case path is the path which produces the maximum run-time. Static analysis has the benefit of guaranteeing a safe upper bound for the WCET making it ideally suited for critical systems but its pessimism and portability can be seen as a weakness for less critical systems [20].

In contrast, the method of *dynamic analysis* takes a measurement-based approach to the task of determining the WCET of a program. Given the program, and the target processor, dynamic analysis executes the program with a large and diverse range of inputs, and measures the execution time of each successive run. This information is used to find the WCET. Often the largest value is taken as the WCET. An exception to this is probabilistic analysis where a statistical estimate is made of the execution time value results to predict a WCET with the required reliability [1].

Dynamic Analysis, however, requires the software under investigation to be tested with a significantly large number of appropriate inputs to achieve a confidence level that the worst-case run has occurred, and therefore, the WCET has been encountered. Two distinctly different approaches exist for dynamic analysis; hybrid approaches which combine static analysis and measurement-based approaches, and so called search-based techniques. The hybrid techniques [1] show significant promise but come with a high degree of complexity [20]. Instead here we explore the extent to which search-based techniques can be improved.

Search-based techniques have been used to generate the input test-data, as the input space of the program can be large and quite complex. Results of these optimisation techniques [13, 15, 16, 17] show that they are effective in sampling large search spaces. They have been applied to both academic benchmarks and more significantly to industrial case studies from both automotive [17] and aerospace applications [15]. In the case of the aerospace application [15], the results were used in preference to those from static analysis which had at least 30% pessimism [5]. Pessimism is defined as the difference between the estimated and actual value. It has been observed that the pessimism produced by static analysis techniques is greater than the optimism linked with evolutionary, search-based techniques [16]. Further, it is widely recognised that the pessimism associated with static analysis will only get worse with more complex processors [20].

In [16] a comparison is performed of these techniques with those of static analysis showing that

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.20

each technique has its own merits and drawbacks with the decision of which to use depending very much on the application context. As with all dynamic analysis approaches they can only guarantee a safe upper bound on the WCET if the appropriate test data is used. All the previous work based on search-based techniques have only used a single criteria function, the WCET itself, in the search for good test-inputs. Wegener [13, 16, 17] has suggested adding other criteria into a combined fitness function as a way of improving the search, i.e. the time to find the maximum execution and the magnitude of this execution time. As modern processors have a number of performance-enhancing features such as caches and pipelines, use of these features can significantly alter the time taken by the software. In some cases, these features even cause a different path to emerge as the new worst-case path. Thus, including these features in the fitness function for the search may beneficially modify the search direction. This type of strategy is widely recognised across search-based software engineering [21]. An example of how a feature may be included is to have a criteria for the number of cache misses encountered.

For example, having an additional criterion could be added to influence the search to derive test cases which maximise the number of cache misses. It has the additional benefit that it may increase the number of instructions executed or data accesses made or find most interesting paths through the software. Of course this does not guarantee to find the WCET but may help. To the best of our knowledge no other work has explored the use of multi-criteria approaches. The closest work to this is Betts [2] who proposed the use of different coverage criteria, instead of the more usual Modified Condition/Decision Coverage [12] which is an often used structural coverage criteria. These other criteria allow for low-level processing effects on the WCET, e.g. the order of instructions through a pipeline. However they have not evaluated the concept, and using coverage criteria is a different approach to what is proposed here, as we do not attempt to achieve complete coverage.

The contribution of this paper, consequently, is to establish the processor and software features which have the most notable impact on the search for the WCET, and to determine the best way in which the effect of these features can be captured in a multi-criteria heuristic function.

The structure of the paper is as follows. Section 2 surveys issues identified in the literature that affect WCET analysis. Next, different methods for WCET analysis using search-based techniques are proposed. In section 4 the results are presented. Finally conclusions are drawn in section 5.

2 Problems in Timing Analysis

A number of problems are encountered in analysing the temporal behaviour of real-time programs. These range from complexity in the real-time programs to be analysed, to the internal features of the hardware. These are discussed in the following sections.

2.1 Causes of Complexity in Real-Time Programs

Groβ [8] defines some aspects of complexity in real-time programs which make it difficult for a dynamic analysis technique, for example a search-based testing method, to *precisely* predict the WCET. The following are identified as important criteria which add to a program's complexity, and consequently, cause difficulty for search-based testing to find the actual solution:

1. High nesting within a program
2. Low path probability of paths within a program, where the low probability is caused by very few values in the domain of the input variable(s), for the program being analysed, leading to the choice of that path.
3. High parameter and algorithmic interdependence, which is caused by the execution time being highly dependent on the values of the input variables, e.g. in a parameter-dependent loop.
4. Size of the program's input, or the range values that input variables can take.

Groβ [8] validates the difficulty of each of these measures by applying them individually to 22 simple test programs. The results show that, generally, increasing the difficulty of each measure makes it more difficult to find the actual WCET. Even though the complexity measures have only been applied to simple test programs, Groβ [8] proposes that the significant number of test programs used, together with the considerable variations in the complexity of the test objects, makes the results generally applicable to larger more complex real-time programs.

2.2 Complexity in Processor Hardware

As with program characteristics there are certain hardware features that make finding the WCET analysis more difficult. These are widely recognised to be as follows [20].

1. Cache – there are three types of cache: instruction, data and unified as well as many different configurations, e.g. direct mapped, set associative or multi-level. Normally as the number of cache misses increases so does the WCET.
2. Pipeline – here three types of complexity are introduced; instruction level parallelism, resource sharing and allocation, and dynamic scheduling where instructions can be executed out of order.
3. Branch prediction – similar to caches there is an aspect of global history to decide which instructions' information is stored for. However in addition there can be complex logic deciding the result of the prediction.

However there are other significant issues that these features introduce. The principal one being timing anomalies which can be introduced when processors feature dynamic scheduling [11]. Timing anomalies are defined as situations where the counter-intuitive influence of the local execution time of one instruction

has an adverse bearing on the global execution time of the whole task [20]. Thus, a faster execution within part of the code can actually cause an increase in the execution time of the whole task, perhaps even leading to the WCET.

An advantage of dynamic analysis techniques, including the search-based techniques discussed here, is the fact that timing anomalies are allowed for in the measures of the actual run-time of the software on the target processor [1]. Thus, an execution containing a timing anomaly but with a larger overall execution time is still considered, and given preference over a non-anomalous execution.

3 Finding the WCET Using Search

In section 2 a number of contributing factors were identified for why WCET analysis is a difficult, generally speaking intractable [20], problem. Each of these factors may also influence the final WCET, e.g. maximising the number of cache misses normally results in a larger execution time. They also mean that there is no way of knowing what the actual WCET is. Therefore the normal approach [20] is to consider how different analysis methods affect the ability to find the WCET. As with all approaches to WCET analysis, it is assumed the software being analysed runs non-preemptively with effects from other software (e.g. context switches, cache pollution etc) being accounted for as part of higher-level schedulability analysis [20].

Two alternative approaches were considered. Firstly to perform a comprehensive set of evaluations and determine which approach best achieves our desirable properties, i.e. achieving the maximum execution time, repeatability / reliability, and efficiency (i.e. how long the search takes), and in what circumstances. Secondly, to again perform comprehensive evaluations but use an approach such as Principal Component Analysis (PCA) [9] to try and determine dominant patterns. The first approach was taken as it was anticipated that the combination of a large complex problem landscape and the fact the landscape may be different for each problem will make it difficult for PCA to be applied. However with the results from the first approach in place, future work may then apply PCA as it could be used in a more focussed way.

There are four phases to the work:

1. Single criterion – in essence perform WCET analysis as previously demonstrated by Tracey [15], Wegener [16,17] and Groβ [8].
2. Low-level analysis – investigate to what extent adding criteria based on low-level (instruction) features influences the search. Key factors are the number of cache misses and branch mis-predictions. Cache misses can be separated into data and instruction or treated together.
3. High-level analysis – examine to what extent high-level (program flow) features affect the search. The key factor here is the loop count.
4. Integrated analysis – consider how combinations of the previous three phases perform.

The following sub-sections discuss the phases in more detail.

3.1 Phases 1 – Single criterion search

A Genetic Algorithm (GA) was developed for automatically generating input test-data. The algorithm was chosen as previous work [15] has used it, albeit it with minor differences in approaches, and initial trials have shown it to be effective. This algorithm initially uses execution time as its single fitness measure. The representation, that is subsequently manipulated through crossover and mutation, is the set of values for the input variables. This data is the only item under our control.

The design of the GA used is a relatively general one based on those in [18]. Certain common choices and strategies for the algorithm were, however, fixed and used throughout this project to support fair comparison between the different approaches. These choices are:

1. Population Size: 100
2. Number of Generations: 100
3. No. of elitist children in the next generation: 1
4. Selection: Roulette Selection
5. Crossover Strategy: Arithmetic Crossover
6. Mutation Strategy: Random Mutation
7. Crossover Percentage: 60%
8. No. of runs for each experiment: 10

A choice of 100 for the population size ensures that there is sufficient diversity within the population without this value being so large as to significantly slow the computations at each generation. Further, fixing the number of generations to 100 ensures that the trajectory of the search can be examined once it has completed, without being so large as to significantly slow down each experiment. Restricting the number of elitist children in the next generation to 1 allows only the best solution to be carried over from one generation to the next, rather than a large number of best solutions. This helps maintain the diversity of the search. Additionally, roulette selection, arithmetic crossover and random mutation are common strategies adopted by the genetic algorithm, which work well in practice and are simple to implement. A crossover percentage of 60% has also been observed to work well in a genetic algorithm. The last requirement, for each experiment to be conducted 10 times, enables the reliability of the solution, i.e. the degree of repeatability, to be recorded. Here reliability is defined as the likelihood of obtaining similar results in practice.

Simplescalar [3] was chosen as the processor simulator on which the working of the input program would be analysed. Further, the ARM-processor was chosen for the experiments as it was supported by Simplescalar and as configured in this work represents a relatively complex processor, containing two-levels of cache, branch prediction and out-of-order execution. This ensured that, in general, experimental results were representative of executions on a modern processor. Specific details of the processor used are as follows. These are the default configuration of simplescalar and

as such has often been used in other work [14].

- Branch predictor: bimodal
- Branch predictor table size: 2048 entries
- Branch Target Buffer (BTB): 512 blocks
- Data Cache (L1): 128 blocks, 32 bytes block size, Least Recently Used (LRU) replacement policy
- Data Cache (L1) Hit Latency: 1 cycle
- Instruction Cache (L1): 512 blocks, 32 bytes block size, LRU replacement policy
- Instruction Cache (L1) Hit Latency: 1 cycle
- Memory Access Latency: 18 cycles (first block in a multi-fetch instruction), 2 cycles (subsequent blocks)

3.2 Phase 2 – Low-level analysis

Based on a detailed consideration of the low-level microprocessor features that affect the WCET, summarised in section 2, the following is a list of criteria considered as part of a multi-criteria search method. Each of the criteria can be used in isolation or combined with others. For example it may be expected to use combined knowledge of cache misses, i.e. an overall total for the data and instruction cache misses. Number of misses is used instead of a normalised rate (total misses divided by the number of memory accesses) as this will favour software paths with more memory accesses. These figures are obtained by parsing the detailed log files, produced by Simplescalar, for the information needed. Future work could consider other means for obtaining the information, including *in vivo* analysis.

1. Execution Time (ET)
2. Branch Prediction Misses (BPM)
3. Data (Level 1) Cache Misses (DCM)
4. Instruction (Level 1) Cache Misses (ICM)

These measures were used to create the following heuristics. The heuristics feature different ratios (or weightings) used to combine the results from the evaluation of each of the criterion. Different ratios are used between the same sets of criteria so that the effect of biasing can be examined. The ratios were chosen based on the results of some preliminary assessments to determine the typical quantities involved and then to provide some degree of balancing. For example the ratios between BPM and ICM is chosen so the outputs of their respective analyses tends to give a similar magnitude.

(a) BPM only
(b) DCM only
(c) ICM only
(d) DCM and ICM in the ratio 1:1
(e) DCM and ICM in the ratio 3:1
(f) DCM and Instruction Cache Accesses in the ratio 1:1
(g) BPM, ICM and DCM in the ratio 2:1:1
(h) ET and BPM in the ratio 1:10
(i) ET and DCM in the ratio 1:10
(j) ET and ICM in the ratio 1:10
(k) ET, DCM and ICM in the ratio 1:5:5
(l) ET, DCM and ICM in the ratio 2:15:5
(m) ET, BPM, DCM and ICM in the ratio 1:10:10:10
(n) ET, BPM, DCM and ICM in the ratio 7:10:10:10
(o) ET, BPM, DCM and ICM in the ratio 7:1:1:1

An important choice when gathering the metrics for each of the criteria is whether they are measured for the whole program or at specific points. For example, the cache miss rate could be measured separately for each memory access and each measure represented by its own criterion. However this would result in a large number of criteria even for small programs and it is generally accepted that searching across more than six criteria is difficult [6]. For this reason one overall measure (a combination by a weighted sum, using the previously mentioned ratios as weights, of the results from evaluating each individual criteria) is made for the whole program.

3.3 Phase 3 – High-level analysis

As previously stated in section 2, the principal issue affecting the WCET is the number of loop iterations and hence this is chosen as a criterion. In a similar fashion to the criteria for low-level analysis, the metrics are gathered across the whole program.

3.4 Phase 4 – Integrated Analysis

The heuristics used as the fitness functions for this phase were obtained by combining the heuristics from Phases 2 and 3. However, as Phase 3 only used a single new execution measure, the number of loop iterations, the resulting heuristics merely added the number of loop iterations to each of the heuristics developed in Phase 2. The heuristics used for the Integration Phase, thus, consisted of:

(a) BPM and Loop Iterations (LI) in the ratio 10:1
(b) DCM and LI in the ratio 10:1
(c) ICM and LI in the ratio 10:1
(d) DCM, ICM and LI in the ratio 5:5:1
(e) DCM, ICM and LI in the ratio 15:5:2
(f) DCM, ICM and LI in the ratio 5:5:1
(g) BPM, DCM, ICM and LI in the ratio 10:5:5:2
(h) ET, BPM and LI in the ratio 1:10:2
(i) ET, DCM and LI in the ratio 1:10:2
(j) ET, ICM and LI in the ratio 1:10:2
(k) ET, DCM, BPM and LI in the ratio 1:5:5:2
(l) ET, DCM, ICM and LI in the ratio 2:15:5:4
(m) ET, DCM, BPM, ICM and LI in the ratio 1:10:10:10:4
(n) ET, DCM, BPM, ICM and LI in the ratio 7:10:10:10:10

4 Evaluation

A set of benchmark problems is publicly maintained by the Mälardalen WCET research group [7]. This comprises a number of programs used to evaluate the performance of different WCET analysis tools. As the benchmarks have already been categorised by type then a selection of programs were chosen to give a reasonable sample from each category after discounting the more trivial examples, e.g. small pieces of code with no distinctive features. Consequently sixteen of the programs, listed in Table 1, were chosen that had a

range of characteristics.

Benchmark Programs	WCET (mean)	WCET (max)
Factorial	2,053.3	2,188.0
Cover	3,991.0	3,991.0
Insertion Sort (10 inputs)	1,328.8	1,333.0
Insertion Sort (50 inputs)	17,528.2	18,240.0
Insertion Sort (100 inputs)	59,367.9	63,216.0
Discrete Cosine Transformation (DCT)	4,186.0	4,186.0
Extended Petri Net Simulation (PETRI)	16,722.7	18,378.0
Matrix multiplication	21,376.0	21,376.0
Quadratic Equations Root Computation (QERC)	1,069.0	1,069.0
Janne Complex	437.3	443.0
Matrix Inversion	3,787.0	3,787.0
Computing an exponential integral function (EXP)	14,317.5	16,358.0
Quick Sort	2,905.4	2,980.0
Fast DCT (FDCT)	4,712.0	4,712.0
Fast Fourier Transformation (FFT)	14,026.2	14,449.0
Select	10,711.7	10,757.0
Statistics Program	14,429.7	14,470.0
Binary Search	269.0	269.0

Table 1 - Phase 1 Results (units = clock cycles)

Each experiment was run 10 times, in order to evaluate the reliability of the solution produced. Reliability is assessed based on the variance of the resulting execution times. Further, the WCET estimate predicted at the end of 100 generations, with a population size of 100, was recorded, together with the time taken to produce the estimate, in order to evaluate the direction, quality and efficiency of the search.

4.1 Phase 1

The results of the experiments are presented in Table 1 for the WCET estimate produced for each benchmark program. These results show the quality of the solution generated for each program, using execution time as the fitness value, and is, additionally, a measure of the general direction of the search after 100 generations. A higher estimate of the WCET is considered more accurate and therefore, better.

The Phase 1 results for the reliability of the search show that three programs, *Insertion Sort, with 100 inputs, PETRI* and EXP had large variances, (denoted by the square of the difference between the mean WCET and the WCET from each of the 10 repeated trials), and so, the results of these programs were the least reliable. At the other end of the scale, seven programs, e.g. *Cover*, had zero variance and are classed as being simple to examine. The variances, and thus the reliability, of other programs varies from under 100 cycles^2 for *Insertion Sort, with 10 inputs* and *Janne Complex* to almost a million cycles^2 for *Insertion Sort, with 50 inputs*. It should be noted that whilst a million cycles^2 may seem large that this only corresponds to a standard deviation of a thousand cycles which is just over 5% of the WCET(max).

4.2 Phases 2

As a summary for the whole set of benchmarks it was found in eleven of the eighteen cases a single criterion, the execution time, as used in phase 1 gave the best results. However this leaves seven important cases for which it did not perform as well. For reasons of space, a full set of results are not presented in depth only those with more interesting characteristics. Interesting is defined as those for which different criteria made a significant difference in terms of quality, efficiency or reliability. The more interesting benchmarks are insertion sort, factorial, janne complex and quadratic.

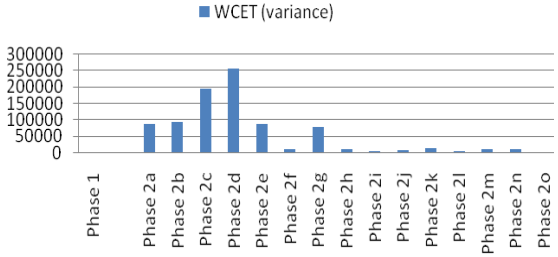


Figure 1- Insertion Sort (10 inputs) (Reliability of Solution)

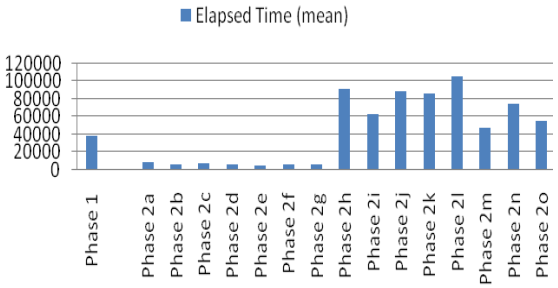


Figure 2 - Insertion Sort (100 inputs) (Efficiency of Solution)

The results of the experiments, for the *interesting benchmarks*, are presented in Figures 1-8, where Phases 2a - 2o represent the heuristics (a) – (o) in section 3.2. The results show that, in general single low-level fitness criteria, for example, only branch mispredictions, data cache misses or instruction cache misses, do not perform well in practice. Thus, in Figures 3-8, the WCET estimate produced by Phases 2a, 2b and 2c ('WCET (max)' for each of these phases), are all among the lowest values predicted. Figure 1 (units for the y-axis are clock cycles^2) shows that these results also have a large variance, and consequently, low reliability as well, thereby reducing their usefulness. The speed of producing these results, however, is high as represented in Figure 2 (units for the y-axis are seconds of actual compute time on a 2.4 GHz Intel processor). Thus, a quick but exceedingly inaccurate estimate for the WCET can be obtained using these low-level criteria individually as the fitness measure. This trend was further demonstrated across the whole set of benchmarks. However if speed and

quality are concerns, then the single criteria of execution time normally offers the best choice.

However, as previously stated, in 7 of the benchmark programs, a multi-criteria fitness measure produced a higher-quality solution than execution time alone. This shows that multi-criteria heuristic fitness functions can still be gainfully used in practice for a program, if appropriate ones are chosen. For example, the Factorial program calls itself recursively passing data at each recursive call. This implies that the program makes heavy use of the data cache, and therefore, the number of data cache misses, and latency caused as a result of these misses, can be a useful metric in guiding the search to the WCET. The results of the Phase 2 experiments on the Factorial program are presented in Figure 5. These results confirm the analysis about the dependence of the Factorial program on the data cache, as data cache misses used alone as the fitness measure (Phase 2b) produces a solution whose quality is only slightly poorer than that of the solution produced by execution time as the single fitness measure (Phase 1). Further, a combination of execution time and data cache misses (Phase 2i) as the fitness measure produces a better solution than execution time used alone, and the highest-quality solution overall, confirming the analysis. It is noted though the wrong combination of criteria has a negative effect as shown by the results of Phase 2m for instance.

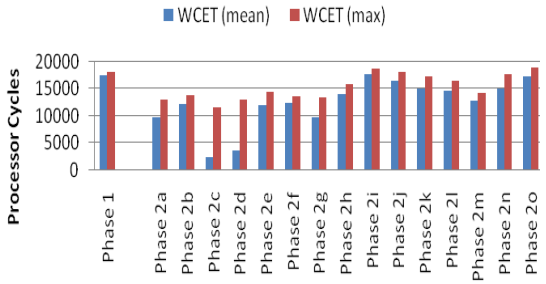


Figure 3 - Insertion Sort (50 inputs) (Quality of Solution)

A program's dependence on the data cache increases with the size of its inputs and the number of calculations performed within it. This information can be gainfully used in devising the best heuristic fitness function to aid the search in reaching the WCET. Thus, a combination of execution time and data cache misses works well in practice, for a program taking a large number of inputs. For example, the Insertion Sort program with 10 inputs (Figure 6), has a relatively low dependence on the data cache. Thus, execution time as the single fitness measure produces a higher quality solution than execution time used with the total number of data cache misses. However, as the size of the inputs is increased, the dependence on the data cache becomes prominent. For example, in the case of the Insertion Sort program with 50 inputs (Figure 3), the combination of data cache misses and execution time as the fitness measure, gives a higher-quality solution than execution time alone, and only a marginally poorer solution than execution time, branch prediction misses, data cache misses and instruction cache misses used altogether as the fitness function, which produces the overall highest-quality solution. However, in the Insertion Sort program with 100 inputs (Figure 7), this situation is reversed. In this program, execution time and data cache misses used together as the fitness function finds the highest-quality solution, thus, validating the analysis. Moreover, the results are produced in only a slightly less efficient way than execution time, branch prediction misses, data cache misses and instruction cache misses used together as the fitness function (Phase 2o), confirming the applicability of this joint fitness measure.

The presence of a large number of loops or conditional statements within a program can cause instruction cache misses, as the instructions after the target of a conditional branch will need to be loaded into the instruction cache, and, depending on the branch prediction method used, cause branch mispredictions as well. Thus, these two execution measures can be gainfully used for such programs as additional criteria within the fitness function of the genetic algorithm. For example, the Quadratic Equations Root Computation program takes only 3 inputs. However, these are used in a loop and conditional statements within the program. Thus, instruction cache misses and branch mispredictions can be used to aid the search. The results for this program (Figure 8) show that execution time and branch mispredictions together (Phase 2h), and execution time and instruction cache misses together as fitness functions (Phase 2j) produce higher-quality solutions than execution time alone, with execution time and instruction cache misses together producing the highest-quality solution. Further, a combination of execution time, branch mispredictions, instruction cache misses and data cache misses as the fitness measure also attains a higher-quality solution than execution time alone. These results show that branch mispredictions and in particular, instruction cache misses can be gainfully used as additional fitness criteria in programs with similar characteristics.

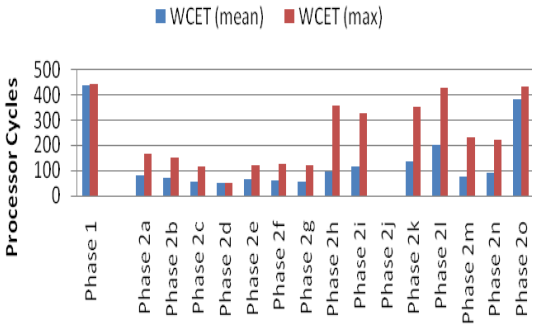


Figure 4 - Janne Complex (Quality of Solution)

In some cases, however, the addition of extra criteria as fitnesses confuses the search algorithm. This is particularly true when the criteria assign conflicting

fitness values to a program execution. For example, the program Janne Complex contains a two-level nested loop. Entry into this loop, is however, dependent on the input values and only a small number of input values cause this loop to be executed. Thus, the path probability of this loop is low. Consequently, if the input values do not permit entry into the loop, a significant number of cache misses outside the loop can confuse the search into giving a high fitness to these program inputs. Thus, executing the loop will then be regarded as an undesirable proposition for the search, as a high fitness solution did not execute it. Clearly with careful tuning of weightings a different result could be achieved, however this is out of scope of this paper. As a result, the loop will remain unexecuted and the WCET estimate produced may be inaccurate. Figure 4 shows the results of applying the search to the Janne Complex program. The result shows that execution time as the single fitness measure (Phase 1) produces the highest-quality solution, while the result for a combination of execution time, branch mispredictions, data cache misses and instruction cache misses, with a heavy bias towards the execution time, as the fitness measure (Phase 2o) is only marginally poorer. However, the rest of the results range from only slightly poorer to much worse. This shows that the fitness assigned to other criteria, such as cache misses and branch mispredictions may cause program inputs resulting in only a few iterations of the loop to be assigned a higher fitness than inputs which execute the loop longer, thereby reducing the program inputs that execute the loop longer from the genetic algorithm's population. The results for Phase 2j show that the highest fitness is dominated by instruction cache misses to the extent that a zero loop count had the highest fitness even though the execution time was low.

Thus, the conflict between criteria is a serious problem in multi-criteria fitness functions as it can inhibit, rather than aid, the search, thus, preventing it from finding the best solution. This problem can be resolved by using the proposed method of program analysis before the search, in order to evaluate the criterion, or combination of criteria that best guides a search to the program's WCET. Later in the paper this is explored further.

4.3 Phases 3 and 4

The results of the experiments are shown in Figures 5-8. The results show that, in general, combining the criteria from phases 2 and 3 do not work well in practice. However this does not preclude the possibility that for other programs the trend will be reversed. The results from Phase 4, additionally, take much longer as they extract a larger amount of execution information, and thus, have a fairly poor speed and efficiency. The reliability of these results, although varying significantly from one program to another, is generally fair and comparable to the results from the other phases.

Moreover, of the remaining 8 programs where execution time does not find the highest-quality solution, in 3 programs, the solution found by execution time is within 1% of the highest-quality solution while, in a further 2 programs, the solution found by execution time is within 5% of the highest-quality solution. The maximum number of highest-quality solutions found by a fitness measure, other than execution time, is 8 out of 18. This is achieved by using execution time, branch prediction misses, data cache misses and instruction cache misses together, with a heavy weight assigned to execution time, as the fitness measure (Phase 2o). Similarly, a combination of execution time and loop counts as the fitness measure (Phase 3b) finds the highest quality solution in 8 (different) cases out of the 18 highest-quality solutions. This shows the general suitability, and applicability, of these fitness measures. However, program analysis should be done beforehand to determine the programs on which use of this particular heuristic as the fitness measure is likely to guide the search to the WCET.

For example, loop iterations (Phase 3a) are seen as the best fitness measure for the Factorial program in Figure 5, as they lead the search to the highest-quality solution. Thus, program analysis beforehand can establish that the number of procedure calls is directly proportional to the execution time and directly dependent on the program inputs. Consequently, use of the number of calls, which subsumes the number of loop iterations (as recursive calls can be considered as loops for the purpose of program analysis) as the fitness heuristic, is the measure most likely to lead a search to the program's WCET.

For the Insertion Sort program, thus, a prior program analysis can establish that loop counts have the most significant bearing for this program, as the program simply consists of a nested loop. Further, the number of iterations of the loop is not dependent on the number of inputs but, rather, the ordering of the inputs. Thus, while increasing the number of inputs will increase the number of data cache accesses and, consequently, data cache misses, the number of times the loop iterates has a more significant bearing on the execution time than the number of data cache misses. Additionally, problems have been seen in the guidance given to the search when two or more criteria are combined. Consequently, it is recommended that only the number of loop iterations, either on its own (Phase 3a) or in combination with execution time (Phase 3b), is used to guide the WCET search. The results show that this analysis holds, as the fittest solution produced by either of these 2 criteria produces the highest-quality solution or a solution whose execution time is within 5% of the best overall estimate for the WCET amongst all the experiments. This is demonstrated in Figure 4, which show high-quality solutions, and other analysis demonstrated it to have an excellent overall reliability.

In contrast, in a program such as Quadratic Equations Root Computation, which contains an input-

data dependent loop and input-data dependent conditional statements, the execution time of the loop is not the primary contributor to the overall execution time. Instead the presence of a loop and conditional statements jointly imply that branch mispredictions or instruction cache misses are likely to affect a significant effect on the execution times. Thus, either of these two measures, when combined with execution time in order to ensure that the search does not proceed in a wrong direction, can be used as an effective fitness measure. The results (Figure 8) show that this holds in practice.

4.4 Proposed Heuristics

Using the results from Phases 1, 2, 3 and 4, the following fitness heuristic is proposed (where the first matching value should be used as the heuristic):

1. If the program has a single path through it, i.e. no conditional statements or loops are dependent on the program inputs, then execution time should be used as the single fitness measure.
2. If the program contains a large number of input-data dependent loops, particularly deeply-nested loops, or an input-data dependent number of self-recursive procedure calls, and few or no conditional statements, then the number of loop iterations and measured execution time should be jointly used as the fitness function. (Large in this context is measured by the percentage of the source code contained within a loop, with a proposed value of 75% or greater constituting a large number of loops within a program.)
3. If the size of the program's input space is large, i.e. the program takes a large number of inputs, and there are multiple paths within the program, where the choice of path is dependent on the program inputs, then data cache misses and execution time should be used together as the fitness function. (In this context, a large number of inputs is defined as the ratio of the size of the inputs to the size of the data cache. If this ratio exceeds a proposed measure of 25%, then the fitness measure proposed in this step should be used.)
4. If the program contains a large number of conditional statements or loops, where the condition is dependent on the program inputs, then instruction cache misses and execution time together should constitute the fitness function. (A large number of conditions is defined as the ratio of the number of conditions to the total number of lines in the source code. A value of 15% or over would constitute large in this context.)
5. If the program contains large basic blocks, that take a long time to execute, then execution time should be used as the fitness measure.
6. If neither of the preceding steps matches the program's characteristics, then execution time should be utilised as the fitness measure.

The choice of heuristic assumes that it will be possible to do program analysis to find the significant characteristics of the program before using a search algorithm on the program to determine its WCET. Such an analysis would benefit the subsequent search significantly. However, if it is not possible to do this analysis, this research recommends the use of the measured program execution time as the fitness measure, as it has been found to be the best-performing general-purpose fitness measure for generating a high-quality (result compared to the best found), reliable (variance of final result over 10 trials) and efficient (time taken to search compared to search with a single objective of execution time) estimate for the WCET.

Benchmark Program	FFP	Performance (%)		
		Quality	Reliability	Efficiency
Factorial	L,E	97.6	164.7	518.5
Cover	E	100.0	100.0	100.0
Insertion Sort (10 inputs)	L,E	100.0	323.5	173.1
Insertion Sort (50 inputs)	L,E	97.9	411.5	81.9
Insertion Sort (100 inputs)	L,E	97.6	200.0	58.2
DCT	E	100.0	100.0	100.0
PETRI	L,E	100.0	100.0	17.6
Matrix multiplication	E	100.0	100.0	100.0
QERC	I,E	100.0	100.0	100.0
Janne Complex	L,E	100.0	331.3	20.6
Matrix Inversion	D,E	99.6	144041.7	93.6
EXP	E	80.4	5,621.3	187.7
Quick Sort	L,E	99.6	195.8	30.0
FDCT	E	100.0	100.0	100.0
FFT	E	100.0	100.0	100.0
Select	L,E	99.9	217.6	212.3
Statistics Program	D,E	99.8	1,088.6	97.5
Binary Search	L,E	100.0	100.0	91.5
Overall		**98.5**	**8,522.0**	**121.3**

Table 2 - Testing the Proposed Heuristics

The proposed fitness function for each benchmark program is listed in Table 2 which also gives the results for the performance metrics (quality / accuracy, reliability and efficiency) and the Fitness Function Proposed (FFP), where L denotes loop count, I the instruction cache misses, D the data cache misses and E the execution time. The quality, reliability and efficiency of the proposed fitness in this table are measured in comparison to the highest-quality solution produced for the respective benchmark problems. In contrast to earlier in the paper, in Table 2 reliability is computed by variance of the highest-quality solution divided by the variance found with the proposed fitness function. Efficiency is computed by time taken to find the highest quality solution divided by the time taken with the proposed fitness function.

5 Conclusions

The paper shows how existing work on search-based WCET analysis can be extended. However the choice of criteria is not always straightforward. In particular the work has showed that simply introducing a wide range of criteria gives bad results and no single set of

criteria works across all the problems. Based on the detailed evaluation performed, recommendations are formed and shown to be effective via further evaluation.

References

[1] G. Bernat, A. Colin and S. Petters, WCET Analysis of Probabilistic Hard Real-Time Systems. In: Proceedings of 23rd IEEE Real-Time Systems Symposium, pp. 279-288, 2002.

[2] A. Betts, G. Bernat, Raimund Kirner, Peter Puschner, and Ingomar Wenzel, WCET Coverage for Pipelines, Techincal report for the ARTIST2 Network of Excellence, August 2006.

[3] D. Burger and T. Austin, The Simplescalar tool set, version 2.0. SIGARCH Computer Architecture News. 25(3), 13-25, 1997.

[4] A. Burns, and A. Wellings, Real-Time Systems and Programming Languages, 3rd Edition, Addison Wesley 2001.

[5] R. Chapman, Static Timing Analysis and Program Proof, PhD thesis, University of York, 1995.

[6] C. Coello, A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques. Knowledge and Information Systems. 1(3), 269-308, 1999.

[7] A. Ermedahl and J. Gustafsson, WCET Project / Benchmarks. Accessed: 5 May 2008. Available at: www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[8] H. Groβ, Measuring Evolutionary Testability of Real-Time Software. Ph.D. thesis, University of Glamorgan/Prifysgol, 2000.

[9] I. Jolliffe, Principal Component Analysis, Wiley, 2005.

[10] R. Kirner, P. Puschner and I. Wenzel, Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation. In Proceedings of the 4th Euromicro Workshop on Worst Case Execution Time Analysis, 2004.

[11] T. Lunqvist and P. Stenstrom, Timing Anomalies in Dynamically Scheduled Microprocessors, In: Proceedings of the 20th IEEE Real-Time Systems Symposium, pp. 12-21, 1999.

[12] P. McMinn, Search-based software test data generation: A survey, Software Testing, Verification and Reliability, 14(2), pp. 105-156, 2004.

[13] H. Pohlheim and J. Wegener, Testing the Temporal Behavior of Real-Time Software Modules using Extended Evolutionary Algorithms. In Proceedings of Genetic and Evolutionary Computation Conference, 1999.

[14] L. Tan, The Worst Case Execution Time Tool Challenge 2006: The External Test, 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006), pp. 241-248, 2006

[15] N. Tracey, J. Clark, and K. Mander, The Way Forward for Unifying Dynamic Test Case Generation: The Optimisation-Based Approach, In Proceedings of The International Workshop on Dependable Computing and Its Applications, 1998.

[16] J. Wegener and F. Mueller, A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. Real-Time Systems Journal, 21(3), 241–268, 2001.

[17] J. Wegener, H. Sthamer, B. Jones and D. Eyres, Testing real-time systems using genetic algorithms. Software Quality Journal, 6(2), 127-135, 1997.

[18] D. Whitley, A Genetic Algorithm Tutorial. Statistics and Computing, 4, 65-85, 1994.

[19] D. Whitley, Genetic Algorithms and Evolutionary Computing. Van Nostrand, 2002.

[20] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. and P. Stenstrom, The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools, ACM Transactions on Embedded Computing Systems, 7(3), 1-53, 2008.

[21] M. Harman, The current state and future of Search Based Software Engineering, In Proceedings of the Future of Software Engineering 2007, pp. 342-357, 2007.

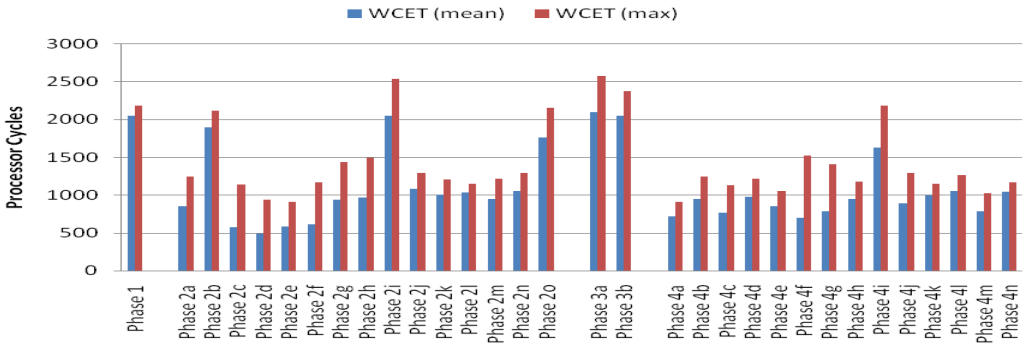


Figure 5 - Factorial (Quality of Solution)

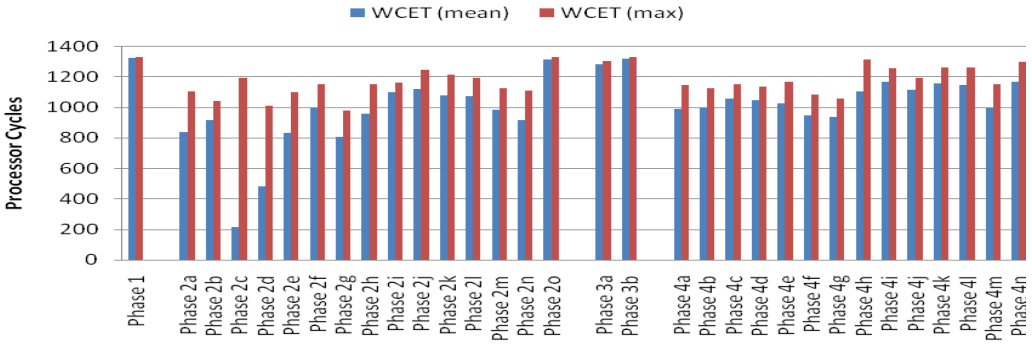


Figure 6 - Insertion Sort (10 inputs) (Quality of Solution)

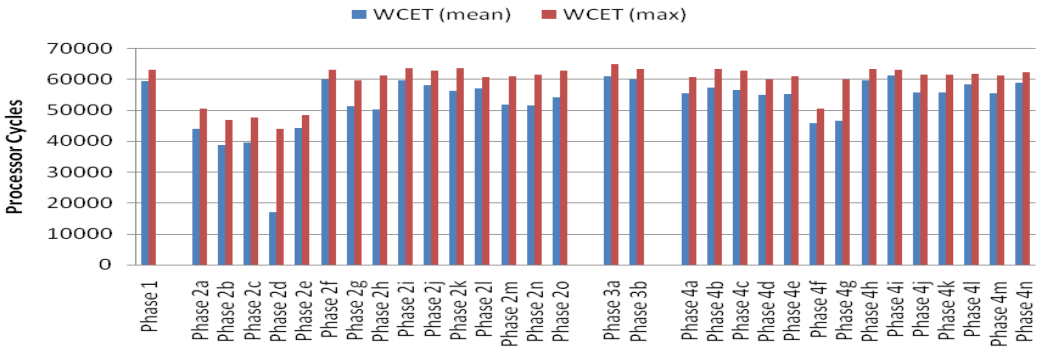


Figure 7 - Insertion Sort (100 inputs) (Quality of Solution)

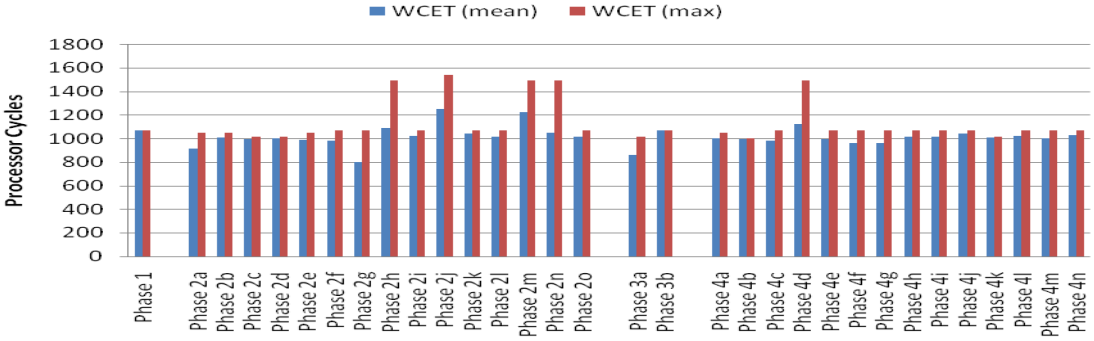


Figure 8 - Quadratic Equations Root Computation (Quality of Solution)

Full Theoretical Runtime Analysis of Alternating Variable Method on the Triangle Classification Problem

Andrea Arcuri
School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, UK.
email: a.arcuri@cs.bham.ac.uk

Abstract

Runtime Analysis *is a type of theoretical investigation that aims to determine, via rigorous mathematical proofs, the* time *a search algorithm needs to find an optimal solution. This type of investigation is useful to understand* why *a search algorithm could be successful, and it gives insight of how search algorithms work. In previous work, we proved the runtimes of different search algorithms on the test data generation for the Triangle Classification (TC) problem. We theoretically proved that Alternating Variable Method (AVM) has the best performance on the coverage of the most difficult branch in our empirical study. In this paper, we prove that the runtime of AVM on all the branches of TC is* $O((\log n)^2)$. *That is necessary and sufficient to prove that AVM has a better runtime on TC compared to the other search algorithms we previously analysed. The theorems in this paper are useful for future analyses. In fact, to state that a search algorithm has worse runtime compared to AVM, it will be just sufficient to prove that its lower bound is higher than* $\Omega((\log n)^2)$ *on the coverage of at least one branch of TC.*

1 Introduction

Although there has been a lot of research on *Search Based Software Engineering* [6, 2, 5] in recent years (e.g, in software testing [15]), there exists few theoretical results. The only exceptions we are aware of are on computing unique input/output sequences for finite state machines [13, 12], the application of the Royal Road theory to evolutionary testing [7], and our previous work on test data generation for the Triangle Classification (TC) problem [1].

To get a deeper understanding of the potential and limitations of the application of search algorithms in software engineering, it is essential to complement the existing experimental research with theoretical investigations. *Runtime Analysis* is an important part of this theoretical investigation, and brings the evaluation of search algorithms closer to how algorithms are classically evaluated.

The goal of analysing the runtime of a search algorithm on a problem is to determine, via rigorous mathematical proofs, the *time* the algorithm needs to find an optimal solution. In general, the runtime depends on characteristics of the problem instance, in particular the problem instance *size*. Hence, the outcome of runtime analysis is usually expressions showing how the runtime depends on the instance size. This will be made more precise in the next sections.

The field of runtime analysis has now advanced to a point where the runtime of relatively complex search algorithms can be analysed on classical combinatorial optimisation problems [20]. We advocate that this type of analysis in Search Based Software Engineering will be helpful to get insight on how search algorithms behave in the software engineering domain. The final aim is to exploit the gained knowledge to design more efficient algorithms.

Branch coverage is the testing task we want to solve. We do a different search for each branch in the code. The employed fitness function is the commonly used approximation level plus the branch distance [15]. Because the number of branches is a constant, the overall runtime for the fulfilment of the test criterion is given by the most expensive search. The size of the problem is given by the constraints on the range of the input variables.

In our previous work [1], we proved runtimes for three different search algorithms on the coverage of one branch of the TC problem. The analysed search algorithms are: Random Search, Hill Climbing and Alternating Variable Method (AVM). The analysed branch is the one related to the classification of the triangle as equilateral (that empirically seems the most difficult to cover).

In that previous work, we proved that AVM has a runtime of $O((\log n)^2)$, that is strictly better than the ones of the other search algorithms we analysed (i.e., $\Theta(n)$ for Hill Climbing and $\Theta(n^2)$ for Random Search). However, that is not sufficient to claim that AVM has a better runtime on TC. In fact, although it has been empirically shown that the "equilateral branch" is the most difficult to cover, that is not

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.16

necessarily true for high values of the size that have not been empirically tested. Other branches might be more difficult. This is one reason why theoretical analyses are necessary.

In this paper, we prove that the expected runtime of AVM on all the branches of TC is $O((\log n)^2)$. That is necessary and sufficient to prove that AVM has a better runtime on TC compared to the other search algorithms we previously analysed. We also carried out an empirical study to integrate the theoretical analysis.

The theorems in this paper are also useful for future analyses. In fact, to state that a search algorithm has worse runtime compared to AVM, it will be just sufficient to prove that its lower bound is higher than $\Omega((\log n)^2)$ on the coverage of at least one branch of TC.

The paper is organised as follows. Section 2 gives background information about runtime analysis. Section 3 describes in detail the TC problem, whereas Section 4 describes the AVM search algorithm. Theoretical analyses are presented in Section 5, whereas the empirical study is discussed in Section 6. Finally, Section 7 concludes the paper.

2 Runtime Analysis

To make the notion of runtime precise, it is necessary to define time and size. We defer the discussion on how to define problem instance size for software testing to the next section, and define time first.

Time can be measured as the number of basic operations in the search heuristic. Usually, the most time-consuming operation in an iteration of a search algorithm is the evaluation of the cost function. We therefore adopt the *black-box scenario* [4], in which time is measured as the number of times the algorithm evaluates the cost function.

Definition 1 (Runtime [3, 8])**.** *Given a class $\mathcal{F}$ of cost functions $f_i : S_i \to \mathbb{R}$, the* runtime $T_{A,\mathcal{F}}(n)$ *of a search algorithm A is defined as*

$$T_{A,\mathcal{F}}(n) := \max \{T_{A,f} \mid f \in \mathcal{F} \text{ with } \ell(f) = n\},$$

where $\ell(f)$ is the problem instance size, and $T_{A,f}$ is the number of times algorithm A evaluates the cost function f until the optimal value of f is evaluated for the first time.

A typical search algorithm A is randomised. Hence, the corresponding runtime $T_{A,\mathcal{F}}(n)$ will be a random variable. The runtime analysis will therefore seek to estimate properties of the distribution of random variable $T_{A,\mathcal{F}}(n)$, in particular the *expected runtime* $E[T_{A,\mathcal{F}}(n)]$ and the *success probability* $\Pr[T_{A,\mathcal{F}}(n) \leq t(n)]$ for a given time bound $t(n)$. More details can be found in [1].

The last decades of research in the area show that it is important to apply appropriate mathematical techniques to get good results [22]. Initial studies of exact Markov chain models of search heuristics were not fruitful, except for the the simplest cases.

A more successful and particularly versatile technique has been so-called drift analysis [8, 19], where one introduces a potential function which measures the distance from any search point to the global optimum. By estimating the expected one-step drift towards the optimum with respect to the potential function, one can deduce expected runtime and success probability.

In addition to drift analysis, the wide range of techniques used in the study of randomised algorithms [17], in particular Chernoff bounds, have proved useful also for evolutionary algorithms.

3 Triangle Classification Problem

TC is the most famous problem in software testing. It opens the classic 1979 book of Myers [18], and has been used and studied since early 70s. Nowadays, TC is still widely used in many publications (e.g., [14, 23, 15, 11, 16, 24]).

We use the implementation for the TC problem that was published in the survey by McMinn [15] (see Figure 1). Some slight modifications to the program have been introduced for clarity.

A solution to the testing problem is represented as a vector $I = (x, y, z)$ of three integer variables. We call (a, b, c) the permutation in ascending order of I. For example, if $I = (3, 5, 1)$, then $(a, b, c) = (1, 3, 5)$.

There is the problem to define what is the *size* of an instance for TC. In fact, the goal of runtime analysis is not about calculating the exact number of steps required for finding a solution. On the other hand, the runtime complexity of an algorithm gives us insight of scalability of the search algorithm. The problem is that TC takes as input a fixed number of variables, and the structure of its source code does not change. Hence, what is the *size* in TC? We chose to consider the range for the input variables for the size of TC. In fact, it is a common practise in software testing to put constraints on the values of the input variables to reduce the search effort. For example, if a function takes as input 32 bit integers, instead of doing a search through over four billion values, a range like $\{0, \ldots, 1000\}$ might be considered for speeding up the search.

Limits on the input variables are always present in the form of bit representation size. For example, the same piece of code might be either run on machine that has 8 bit integers or on another that uses 32 bits. What will happen if we want to do a search for test data on the same code that runs on a 64 bit machine? Therefore, using the range of the input variables as the size of the problem seems an appropriate choice.

In our analyses, the size n of the problem defines the range $R = \{-n/2 + 1, \ldots, n/2\}$ in which the variables in I can be chosen (i.e., $x, y, z \in R$). Hence, the search space S is defined as $S = \{(x, y, z)|x, y, z \in R\}$, and it is composed of n^3 elements. Without loss of generality n is even. To obtain full coverage, it is necessary that $n \geq 8$, otherwise the branch regarding the classification as *scalene* will never be covered. Note that one can consider different types of R (e.g., $R' = \{0, \ldots, n\}$), and each type may lead to different behaviours of the search algorithms. We based our choice on what is commonly used in literature. For simplicity and without loss of generality, search algorithms are allowed to generate solutions outside S. In fact, R is mainly used when random solutions need to be initialised.

The employed fitness function f is the commonly used approximation level $\mathcal{A}$ plus the branch distance δ [15]. For a target branch z, we have that the fitness function f_z is:

$$f_z(I) = \mathcal{A}_z(I) + \omega(\delta_w(I)) \; .$$

Note that the branch distance δ is calculated on the node of *diversion*, i.e. the last node in which a critical decision (not taking the branch w) is made that makes the execution of z not possible. For example, branch z could be nested to a node N (in the control flow graph) in which branch w represents the *then* branch. If the execution flow reaches N but then the *else* branch is taken, then N is the node of diversion for z. The search hence gets guided by δ_w to enter in the nested branches.

Let be $\{N_0, \ldots, N_k\}$ the sequence of diversion nodes for the target z, with N_i nested to all $N_{j>i}$. Let be D_i the set of inputs for which the computation diverges at node N_i and none of the nested nodes $N_{j<i}$ is executed. Then, it is important that $\mathcal{A}_z(I_i) < \mathcal{A}_z(I_j)\ \forall I_i \in D_i, I_j \in D_j, i < j$. A simple way to guarantee it is to have $\mathcal{A}_z(I_{i+1}) = \mathcal{A}_z(I_i) + \zeta$, where ζ can be any positive constant (e.g., $\zeta = 1$) and $\mathcal{A}_z(I_0) = 0$.

Because an input that makes the execution closer to z should be rewarded, then it is important that $f_z(I_i) < f_z(I_{i+1})\ \forall I_i \in D_i, I_{i+1} \in D_{i+1}$. To guarantee that, we need to scale the branch distance δ with a scaling function ω such that $0 \leq \omega(\delta_j) < \zeta$ for any predicate j. Note that δ is never negative. We need to guarantee that the order of the values does not change once mapped with ω, for example $h_0 > h_1$ should imply $\omega(h_0) > \omega(h_1)$. We can use for example either $\omega(h) = (\zeta h)/(h+1)$ or $\omega(h) = \zeta/(1+e^{-h})$, where $h \geq 0$.

Having $\zeta > 0$ and $\gamma > 0$, the fitness functions for the 12 branches (i.e., f_i is the fitness function for branch ID_i) are shown in Figure 2. Note that the branch distance depends on the status of the computation (e.g., the values of the local variables) when the predicates are evaluated. For simplicity, in an equivalent way we show the fitness functions based only on the inputs I.

```
 1: int tri_type(int x, int y, int z) {
 2:   int type;
 3:   int a=x, b=y, c=z;
 4:   if (x > y) { /* ID_0 */
 5:     int t = a; a = b; b = t;
 6:   } else { /* ID_1 */}
 7:   if (a > z) { /* ID_2 */
 8:      int t = a; a = c; c = t;
 9:   } else { /* ID_3 */}
10:   if (b > c) { /* ID_4 */
11:      int t = b; b = c; c = t;
12:   } else { /* ID_5 */}
13:   if (a + b <= c) { /* ID_6 */
14:      type = NOT_A_TRIANGLE;
15:   } else { /* ID_7 */
16:       type = SCALENE;
17:       if (a == b && b == c) {
18:           /* ID_8 */
19:           type = EQUILATERAL;
20:       } else /* ID_9 */
21:              if (a == b || b == c) {
22:                  /* ID_10 */
23:                  type = ISOSCELES;
24:              } else {/* ID_11 */}
25:     }
26:   return type;
27: }
```

Figure 1. Triangle Classification (TC) program, adapted from [15]. Each branch is tagged with a unique ID.

$$f_0(I) = \begin{cases} 0 & \text{if } x > y\ , \\ \omega(|y-x|+\gamma) & \text{otherwise .} \end{cases}$$

$$f_1(I) = \begin{cases} 0 & \text{if } x \leq y\ , \\ \omega(|x-y|+\gamma) & \text{otherwise .} \end{cases}$$

$$f_2(I) = \begin{cases} 0 & \text{if } min(x,y) > z\ , \\ \omega(|z-min(x,y)|+\gamma) & \text{otherwise .} \end{cases}$$

$$f_3(I) = \begin{cases} 0 & \text{if } min(x,y) \leq z\ , \\ \omega(|min(x,y)-z|+\gamma) & \text{otherwise .} \end{cases}$$

$$f_4(I) = \begin{cases} 0 & \text{if } max(x,y) > max(z,(min(x,y)))\ , \\ \omega(|max(z,(min(x,y))) - max(x,y)|+\gamma) & \text{otherwise .} \end{cases}$$

$$f_5(I) = \begin{cases} 0 & \text{if } max(x,y) \leq max(z,(min(x,y)))\ , \\ \omega(|max(x,y) - max(z,(min(x,y)))|+\gamma) & \text{otherwise .} \end{cases}$$

$$f_6(I) = \begin{cases} 0 & \text{if } a+b \leq c\ , \\ \omega(|(a+b)-c|+\gamma) & \text{otherwise .} \end{cases}$$

$$f_7(I) = \begin{cases} 0 & \text{if } a+b > c\ , \\ \omega(|c-(a+b)|+\gamma) & \text{otherwise .} \end{cases}$$

$$f_8(I) = \begin{cases} \zeta + f_7(I) & \text{if } a+b \leq c\ , \\ 0 & \text{if } a == b \wedge b == c \wedge a+b > c\ , \\ \omega(|a-b|+|b-c|+2\gamma) & \text{otherwise .} \end{cases}$$

$$f_9(I) = \begin{cases} \zeta + f_7(I) & \text{if } a+b \leq c\ , \\ 0 & \text{if } (a \neq b \vee b \neq c) \wedge a+b > c\ , \\ \omega(2\gamma) & \text{otherwise .} \end{cases}$$

$$f_{10}(I) = \begin{cases} 2\zeta + f_7(I) & \text{if } a+b \leq c\ , \\ \zeta + f_9(I) & \text{if } a == b \wedge b == c \wedge a+b > c\ , \\ 0 & \text{if } (a \neq b \vee b \neq c) \wedge a+b > c \wedge (a == b \vee b == c)\ , \\ \omega(min(|a-b|+\gamma, |b-c|+\gamma)) & \text{otherwise .} \end{cases}$$

$$f_{11}(I) = \begin{cases} 2\zeta + f_7(I) & \text{if } a+b \leq c\ , \\ \zeta + f_9(I) & \text{if } a == b \wedge b == c \wedge a+b > c\ , \\ 0 & \text{if } a \neq b \wedge b \neq c \wedge a+b > c\ , \\ \omega(\gamma) & \text{otherwise .} \end{cases}$$

Figure 2. Fitness functions f_i for all the branches ID_i of TC. The constants ζ and γ are both positive, and $0 \leq \omega(h) < \zeta$ for any h.

4 Alternating Variable Method

AVM is similar to a Hill Climbing, and was employed in the early work of Korel [10]. The algorithm starts on a random search point I, and then it considers modifications of the input variables, one at a time. The algorithm applies an *exploratory search* to the chosen variable, in which the variable is slightly modified (in our case, by ± 1). If one of the neighbours has a better fitness, then the exploratory search is considered successful. Similarly to a Hill Climbing, the better neighbour will be selected as the new current solution. Moreover, a *pattern search* will take place. On the other hand, if none of the neighbours has better fitness, then AVM continues to do exploratory searches on the other variables, until either a better neighbour has been found or all the variables have been unsuccessfully explored. In the latter case, a restart from a new random point is done if a global optimum was not found.

A pattern search consists of applying increasingly larger changes to the chosen variable as long as a better solution is found. The type of change depends on the exploratory search, which gives a direction of growth. For example, if a better solution is found by decreasing the input variable by 1, then the following pattern search will focus on decreasing the value of that input variable.

A pattern search ends when it does not find a better solution. In this case, AVM will start a new exploratory search on the same input variable. In fact, the algorithm moves to consider one other variable only in the case that an exploratory search is unsuccessful.

To simplify the writing of the AVM implementation, and for making it more readable, it is not presented in its general form. Instead, it is specialised in working on vector solutions of length three. The general version, that considers this length as a problem parameter, would have the same computational behaviour in terms of evaluated solutions.

Definition 2 (Alternating Variable Method (AVM))**.**

while *termination criterion not met*
  Choose I *uniformly in* S.
  while I *improved in last* 3 *loops*
    $i :=$ *current loop index.*
    Choose $T_i \in \{(1,0,0),(0,1,0),(0,0,1)\}$ *such that* $T_i \neq T_{i-1} \wedge T_i \neq T_{i-2}$.
    $found := true$.
    while $found$
      for $d := 1$ *and* $d := -1$
        $found := exploratory\_search(T_i, d, I)$.
        if $found$, **then**
          $pattern\_search(T_i, d, I)$

Definition 3 ($exploratory\_search(T_i, d, I)$)**.**

$I' := I + dT_i$.
if $f(I') \geq f(I)$, **then**
  return $false$.
else
  $I := I'$.
  return $true$.

Definition 4 ($pattern\_search(T_i, d, I)$)**.**

$k := 2$.
$I' := I + kdT_i$.
while $f(I') < f(I)$
  $I := I'$.
  $k := 2k$.
  $I' := I + kdT_i$.

5 Theoretical Analysis

Proposition 1. *Given x and y uniformly and independently distributed in R, then their expected difference with $y \geq x$ is $E[y - x | y \geq x] = \frac{n-1}{3} = \Theta(n)$. The largest difference would be $y - x = n - 1 = \Theta(n)$*

Proof.

$$\begin{aligned} E\left[y - x \mid y \geq x\right] &= \left(n + 2(n-1) + 3(n-2) + \ldots + n(1)\right) \\ &\quad / \tfrac{n(n+1)}{2} - 1 \\ &= \textstyle\sum_{i=0}^{n} (i+1)(n-i) \cdot \frac{2}{n(n+1)} - 1 \\ &= \frac{n(n+1)(n+2)}{6} \cdot \frac{2}{n(n+1)} - 1 \\ &= \frac{n-1}{3} \\ &= \Theta(n)\ . \end{aligned}$$

The highest value that y can take is $n/2$. The lowest value x can take is $-n/2+1$. Hence, $n/2 - (-n/2+1) = n - 1$. □

Lemma 1. *For any branch, if the probability that the random starting point is a global optimum is lower bounded by a positive constant $k > 0$, then AVM needs at most a constant number $\Theta(1)$ of restarts to find a global optimum.*

Proof. If we consider only the starting point, the AVM behaves as a random search, in which the probability of finding a global optimum is bigger than k. That can be described as a Bernoulli process (see our theorems on random search in [1]), with expected number of restarts that is lower or equal than $1/k$. □

Theorem 1. *The expected time for AVM to find an optimal solution to the coverage of branches ID_0 and ID_1 is $O(\log n)$.*

Proof. The probability that $x > y$, with X and Y the random variables representing them, is:

$$\Pr[X > Y] = \sum_{y} \sum_{i=y+1}^{n/2} \Pr[X = i \mid Y = y] \cdot \Pr[Y = y]$$
$$= \frac{1}{2} - \frac{1}{2n},$$

The probability of $x \leq y$ is hence $\frac{1}{2} + \frac{1}{2n}$

Considering that the search is done for values of n bigger or equal than 8, then both the searches for ID_0 and ID_1 start with a random point that is a global optimum with a probability lower bounded by a positive constant. Therefore, AVM needs at most a constant number of restarts (Lemma 1), independently of the presence and number of local optima.

For both branches, either the starting point is a global optimum, or the search will be influenced by the distance $x - y$ that is $\Theta(n)$ (Proposition 1). We hence analyse this latter case.

Until the predicate is not satisfied, the fitness function $\omega(|y - x| + \gamma)$ (with γ a positive constant) based on the branch distance rewards any reduction of that distance. The third variable z does not influence the fitness function, hence an exploratory search fails on that. For the coverage of ID_0, the variable x has gradient to increase its value, and y has gradient to decrease. For ID_1 it is the opposite. The distance $|x - y|$ can be covered in $O(\log n)$ steps of a pattern search.

□

Theorem 2. *The expected time for AVM to find an optimal solution to the coverage of branches ID_2, ID_3, ID_4 and ID_5 is $O(\log n)$.*

Proof. The sentences in lines 5, 8 and 11 of the source code (Figure 1) only swap the value of the three input variables. Hence, the predicate conditions of branches ID_2, ID_3, ID_4 and ID_5 are directly based on the values of two different input variables.

The type of predicates is the same of branches ID_0 and ID_1 (i.e., $>$), and the condition of the comparison is the same (i.e., $>$ on two input variables). The three input variables are uniformly and independently distributed in R, and by Proposition 1 the maximum distance among them is $\Theta(n)$. There are the same conditions of Theorem 1 apart from the fact that the variables could be swapped during the search, i.e. the fact that lines 5 and 8 are executed or not can vary during the search.

For branches ID_2 and ID_3, starting from z no variation of the executed code is done until the branch is covered. For branch ID_3, for either x or y a search starting from the maximum of them would result in no improvement of the fitness function. The minimum of x and y has a gradient to decrease, and while it does so the relation of their order is not changed. Hence, no variation of the executed code is done. On the other hand, for branch ID_2, the minimum has gradient to increase, but the pattern search would stop once it becomes the maximum of the two (e.g., $x > y$ if the search started on x with $x < y$). That happens in at most $O(\log n)$ steps because their difference is at most $\Theta(n)$ (Proposition 1). If the next variable considered by AVM is not z, then the above behaviour will happen again. However, the next variable will be necessarily z, hence we have at most $O(\log n)$ steps done 3 times, that still results in $O(\log n)$ steps.

For any pair of values we have that $min(x, y) \leq max(x, y)$. For branch ID_4, if it is not executed, then $max(x, y) \leq max(z, min(x, y))$ and necessarily it would be $z \geq max(x, y) \geq min(x, y)$. Hence, z would have gradient to decrease down until $max(x, y)$, in which case ID_4 gets executed after $O(\log n)$ steps. A modification of the minimum value between x and y does not change the fitness value. For the maximum value, it can increase up to z, in which case ID_4 gets executed after $O(\log n)$ steps. The relation of the order of the input variables would not changed during those searches.

For branch ID_5, if it is not executed, then $max(x, y) > max(z, min(x, y))$ and necessarily it would be $x \neq y$ and $max(x, y) > z$. Starting the search from the maximum of x and y would have gradient to decrease down to $max(z, min(x, y))$, that would be done in $O(\log n)$ steps that will make ID_5 executed. If $z < min(x, y)$, modifying z would have no effect to the fitness function, whereas the minimum of x and y has gradient to increase up to $max(x, y)$. In the other case $z \geq min(x, y)$, it is the other way round, i.e. z can increase whereas the minimum between x and y cannot change. In both cases, in $O(\log n)$ steps branch ID_5 gets executed with no change in the relation of the order of the input variables.

The expected time for branches ID_2, ID_3, ID_4 and ID_5 is therefore the same as for branches ID_0 and ID_1, i.e. $O(\log n)$.

□

Theorem 3. *The expected time for AVM to find an optimal solution to the coverage of branch ID_6 is $O(\log n)$.*

Proof. If the predicate $a + b \leq c$ is not true, the fitness function would be $\omega(|a + b - c| + \gamma)$ (with γ a positive constant). For values $a \leq 0$, the predicate is true because $a + b \leq b \leq c$.

There is gradient to decrease a and b, and there is gradient to increase c. If the search starts from either a or b, in $O(\log n)$ steps of a pattern search the target variable assumes a negative value (the highest possible starting value is $n/2$). In particular, if the search starts from b, at a certain

point the input variable representing b will instead represent a. Otherwise, it sufficient to increase c up to the value $a+b \leq n/2+n/2 = n$, that can be done in $O(\log n)$ steps of a pattern search.

□

Theorem 4. *The expected time for AVM to find an optimal solution to the coverage of branch ID_8 is $O((\log n)^2)$.*

Proof. This theorem has been proved in our previous work [1]. □

Lemma 2. *For search algorithms that use the fitness function only for direct comparisons of candidate solutions, the expected time for covering a branch ID_w is not higher than the expected time to cover any of its nested branches ID_z.*

Proof. Before a target nested branch ID_z is executed, its "parent" branch ID_w needs to be executed. Until ID_w is not executed, the fitness function f_z^w (i.e., search for ID_z and ID_w is not covered) will be based on the predicate of the branch ID_w. Hence, that fitness function would be equivalent to the one f_w used for a direct search for ID_w. In particular, $f_z^w(I) = \zeta + f_w(I)$, that because the approximation level would be different. However, because the constant $\zeta > 0$ would be the same to all the search points, the behaviour of a search algorithm, that uses the fitness function only for direct comparisons of candidate solutions, would be same on these two fitness functions (and AVM satisfies this constraint).

Because the time to solve (i.e., finding an input that minimises) f_z^w is not higher than the time needed for f_z and because f_z^w is equivalent to f_w, then solving f_w cannot take in average more time than solving f_z.

□

Theorem 5. *The expected time for AVM to find an optimal solution to the coverage of branch ID_7 is $O((\log n)^2)$.*

Proof. The branch ID_8 is nested to branch ID_7, hence by Lemma 2 and Theorem 4 the expected time is $O((\log n)^2)$. □

Theorem 6. *The expected time for AVM to find an optimal solution to the coverage of branch ID_9 is $O((\log n)^2)$.*

Proof. By Theorem 5, the branch ID_7 can be covered in $O((\log n)^2)$ steps. The branch ID_9 (that is nested to ID_7), will be covered if $\neg(a = b \wedge b = c)$. If that predicate is not true, a single exploratory search of AVM makes it true because it is just sufficient to either increase or decease any input variable by 1. The only case in which this is not possible is for $I = (1, 1, 1)$, because it is the only solution that satisfies $a = b \wedge b = c \wedge a - 1 + b \leq c \wedge a + b \leq c + 1 \wedge a + b > c$. In that case, a restart is done.

With a probability that is lower bounded by a constant, in a random starting point each input variable is higher than $n/4$. In that case, $a+b > c$, because $(n/4)+1+(n/4)+1 > (n/2)$. By Lemma 1, we need only $\Theta(1)$ restarts.

□

Theorem 7. *The expected time for AVM to find an optimal solution to the coverage of branch ID_10 is $O((\log n)^2)$.*

Proof. By Theorem 6, the branch ID_9 can be covered in $O((\log n)^2)$ steps. The branch ID_10 (that is nested to ID_9), will be covered if only two input variables are equal (and not all three equal to each other at the same time).

If when the branch ID_7 (branch ID_9 is nested to it) is executed all the three input variables are equal (in that case branch ID_8 is executed), then a single exploratory search is sufficient to execute branch ID_10, because we just need to change the value of a single variable.

The other case in which all the three variables are different is quite complex to analyse. Instead of analysing it directly, we prove the runtime by a comparison with the behaviour of AVM on the branch ID_8 (that is more complex and we already proved it in our previous work [1]).

Once branch ID_9 is executed, the fitness function f_{10}^9 for covering branch ID_10 is based on $min(\delta(a = b), \delta(b = c))$, whit δ the branch distance function for the predicates. For simplicity, let consider $\delta(a = b) < \delta(b = c)$. The other case can be studied in the same way.

An exploratory search cannot accept a reduction of the distance $c - b$, because the value of f_{10}^9 would not improve. A search on a would leave the distance $c - b$ unchanged. About b, only a decrease of its value would be accepted, and in that case the distance $c-b$ would increase (but that has no effect on the fitness function because it takes the minimum of the two distances). Because the branch distance δ only rewards the reduction of the distance $b - a$, a search starting from either a or b will end in $a = b$ by modifying only the value of only one of these variables (AVM keeps doing searches on the same variable till an exploratory search fails). During that search, the fitness function would hence be based on $\delta(a = b)$.

In a search for covering branch ID_8, if the branch ID_7 (in which both ID_8 and ID_10 are nested) is executed, then the fitness function f_8^7 depends on $\delta(a = b)+\delta(b = c)$. A search starting from a would finish in $a = b$ for the same reasons explained before or it would finish in $a' > b$ (with a' the latest accepted point for a that will become the new b in the next exploratory search). During that search, the value of $\delta(b = c)$ does not change, so it can be considered as a constant. Because AVM uses the fitness function only on direct comparisons, the presence of a constant does not influence its behaviour. Therefore, in this particular context (i.e., $\delta(a = b) < \delta(b = c)$, branch ID_7 executed and

search starting from a) the behaviour of AVM on f_{10}^9 and f_8^7 will be the same until $a = b$ or $a' > b$.

In the case $a = b$, branch ID_10 gets executed and the search for that branch ends. In the other case $a' > b$, the previous b becomes the new a_k and a' becomes the new b_k. Modifications on the variable c does not change the value of either a_k or b_k. The previous analysis can hence be recursively applied to the new values a_k and b_k. If $a' > c$, then $a_k = b$, $b_k = c$, $c_k = a'$ and it will become the case $\delta(a = b) > \delta(b = c)$.

It is still necessary to analyse the behaviour of AVM on f_{10}^9 when the search starts on b rather than a. That is similar to the case of f_8^7 when the variable c is decreased down to b. In that context, the two fitness functions are of the same type because in f_8^7 the distance $\delta(a = b)$ would be a constant until $b = c$ or $c' < b$ (with c' the latest accepted point for c that will become the new b in the next exploratory search). Therefore, the runtime for AVM on f_{10}^9 to obtain $a = b$ would be the same.

By Theorem 4, the expected time for covering branch ID_8 is $O((\log n)^2)$. Because we proved that the coverage of ID_8 takes more time than the coverage of ID_10, then the expected runtime for covering ID_10 is $O((\log n)^2)$. □

Theorem 8. *The expected time for AVM to find an optimal solution to the coverage of branch ID_11 is $O((\log n)^2)$.*

Proof. By Theorem 6, the branch ID_9 can be covered in $O((\log n)^2)$ steps. The branch ID_11 (that is nested to ID_9), will be covered if $a \neq b \wedge a \neq c \wedge b \neq c$. In the moment that the branch ID_9 is executed, then the three variables cannot assume all the same value (otherwise the branch ID_8 would have been executed). If that predicate is not true, a single exploratory search of AVM makes it true because it is just sufficient to increase by 1 any of the two variables that have same values.

□

6 Empirical Study

We ran an implementation of AVM on each branch of TC for values of n such that $n = 2^i$, with $i \in \{4, 5, 6, \ldots, 29, 30\}$. For each size of n, we ran 1000 trials (with different random seeds) and recorded the number of fitness evaluations done before reaching a global optimum.

Following [9], for each setting of algorithm and problem instance size, we fitted different models to the observed runtimes using non-linear regression with the Gauss-Newton algorithm. Each model corresponds to a one term expression $\eta \cdot g(n)$ of the runtime, where the model parameter η corresponds to the constant to be estimated. The residual sum of squares of each fitted model was calculated to identify the model which corresponds best with the observed runtimes. This methodology was implemented in the statistical tool R [21].

Table 1. Results of empirical experiments.

Branch ID	Runtime
ID_0	$0.48 \log_2 n$
ID_1	$0.50 \log_2 n$
ID_2	$1.03 \log_2 n$
ID_3	$0.36 \log_2 n$
ID_4	$0.34 \log_2 n$
ID_5	$1.62 \log_2 n$
ID_6	$0.10 \log_2 n$
ID_7	$5.31 \log_2 n$
ID_8	$0.53 (\log_2 n)^2$
ID_9	$5.10 \log_2 n$
ID_10	$7.47 \log_2 n$
ID_11	$5.01 \log_2 n$

The used models were $\eta n^t \log(n)^v$, where $t \in \{0, 1, 2\}$ and $v \in \{0, \ldots, 10\}$. The models with lowest error are shown in Table 6.

The results in Table 6 are consistent with our theoretical results. They are able to provide the constants for the runtime models. It is worth noting that running so many experiments (i.e., $324,000$) was possible because the runtime of AVM is $O((\log n)^2)$. In the case for example of a runtime $\Theta(n^2)$ (e.g., Random Search [1]), running so many experiments would have likely been unfeasible. The resulting estimated models could have been hence not precise.

At any rate, we ran our experiments only with values of n up to 2^{30}. We cannot know for sure what could happen for higher values. On the other hand, our theoretical analysis is valid for each value of n.

7 Conclusions and Future Work

In this paper, we proved that the runtime of AVM on all the branches of TC is $O((\log n)^2)$. This is necessary and sufficient to prove that AVM has a better runtime on TC compared to the other search algorithms we previously analysed, i.e. Random Search and Hill Climbing. In the future, to state that a search algorithm performs worse than AVM on TC, it will be just sufficient to prove that its lower bound is higher than $\Omega((\log n)^2)$ on the coverage of at least one branch of TC.

Previously, we proved that AVM requires on average $O((\log n)^2)$ steps for covering the branch related to the classification of the triangle as equilateral (ID_8). However, we needed to prove the runtime on each single branch, because ID_8 is not necessarily the most difficult to cover. Although empirical studies in literature have shown that

branch ID_8 seems the most difficult to cover, that is not sufficient, because different behaviours might arise for very high values of the size.

This type of analysis is important to understand the behaviour of search algorithms on software engineering problems. Unfortunately, the fact that they are difficult to carry out limits its scope. Therefore, theoretical runtime analysis is not meant to replace empirical studies. However, for the problems for which theoretical analyses can be done, we get stronger and more reliable results than any obtained with empirical studies.

For the future, we want to analyse other search algorithms, in particular Genetic Algorithms. Considering other implementation of TC would be interesting to see if there is any difference in the overall runtimes of the analysed algorithms. In the long term, it will be interesting to analyse more complex software to get more insight on how search algorithms work.

8 Acknowledgements

The author is grateful to Xin Yao and Per Kristian Lehre for insightful discussions. This work is supported by EPSRC grant EP/D052785/1.

References

[1] A. Arcuri, P. K. Lehre, and X. Yao. Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem. In *International Workshop on Search-Based Software Testing (SBST)*, pages 161–169, 2008.

[2] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.

[3] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276:51–81, 2002.

[4] S. Droste, T. Jansen, and I. Wegener. Upper and lower bounds for randomized search heuristics in black-box optimization. *Theory of Computing Systems*, 39(4):525–544, 2006.

[5] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE)*, pages 342–357, 2007.

[6] M. Harman and B. F. Jones. Search-based software engineering. *Journal of Information & Software Technology*, 43(14):833–839, 2001.

[7] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 73–83, 2007.

[8] J. He and X. Yao. A study of drift analysis for estimating computation time of evolutionary algorithms. *Natural Computing*, 3(1):21–35, 2004.

[9] T. Jansen. On the brittleness of evolutionary algorithms. In *Foundations of Genetic Algorithms*, pages 54–69, 2007.

[10] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, pages 870–879, 1990.

[11] B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from b formal models: Research articles. *Software Testing, Verification and Reliability*, 14(2):81–103, 2004.

[12] P. Lehre and X. Yao. Crossover can be constructive when computing unique input output sequences. In *Proceedings of the International Conference on Simulated Evolution and Learning (SEAL)*, pages 595–604, 2008.

[13] P. K. Lehre and X. Yao. Runtime analysis of (1+1) ea on computing unique input output sequences. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1882–1889, 2007.

[14] J. C. Lin and P. L. Yeh. Automatic test data generation for path testing using GAs. *Information Sciences*, 131(1-4):47–64, 2001.

[15] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[16] J. Miller, M. Reformat, and H. Zhang. Automatic test data generation using genetic algorithm and program dependence graphs. *Information and Software Technology*, 48(7):586–605, 2006.

[17] M. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[18] G. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[19] P. Oliveto and C. Witt. Simplified drift analysis for proving lower bounds in evolutionary computation. In *Proceedings of the international conference on Parallel Problem Solving from Nature*, pages 82–91, 2008.

[20] P. S. Oliveto, J. He, and X. Yao. Time complexity of evolutionary algorithms for combinatorial optimization: A decade of results. *International Journal of Automation and Computing*, 4(3):281–293, 2007.

[21] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[22] I. Wegener. Methods for the analysis of evolutionary algorithms on pseudo-boolean functions. Technical Report CI-99/00, Universität Dortmund, 2000.

[23] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[24] M. Xiao, M. El-Attar, M. Reformat, and J. Miller. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering*, 12(2):183–239, 2007.

Widening the Goal Posts: Program Stretching to Aid Search Based Software Testing.

Kamran Ghani and John A. Clark
Department of Computer Science
University of York
Heslington, York,
YO10 5DD, UK
{*kamran, jac*}*@cs.york.ac.uk*

Abstract

Search based software testing has emerged in recent years as an important research area within automated software test data generation. The general approach of couching the satisfaction of test goals as numerical optimisation problems has been applied to a variety of problems such as satisfying structural coverage criteria, specification falsification, exception generation, breaking unit pre-conditions and software hazard discovery. However, some test goals may be hard to satisfy. For example, a program branch may be difficult to reach via a search based technique, because the domain of the data that causes it to be taken is exceedingly small or the non-linearity of the "fitness landscape" precludes the provision of effective guidance to the search for test data. In this paper we propose to "stretch" relevant conditions within a program to make them easier to satisfy. We find test data that satisfies the corresponding test goal of the stretched program. We then seek to transform the stretched program by stages back to the original, simultaneously migrating the obtained test data to produce test data that satisfies the goal for the original program. The "stretching" device is remarkably simple and shows significant promise for obtaining hard-to-find test data and also gives efficiency improvements over standard search based testing approaches.

1. Introduction

1.1. Dynamic Testing

Dynamic testing — "the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behaviour" [1] — is used to gain confidence in almost all developed software. Various static approaches can be used to gain further confidence but it is generally felt that only dynamic testing can provide confidence in the correct functioning of the software in its intended environment.

We cannot perform exhaustive testing because the domain of program inputs is usually too large and also there are too many possible execution paths. Therefore, the software is tested using a suitably selected set of test cases. A variety of coverage criteria have been proposed to assess how effective test sets are likely to be. Historically, criteria exercising aspects of control flow, such as statement and branch coverage [2], have been the most common. Further criteria, such as data flow [3], or else sophisticated condition-oriented criteria such as MC/DC coverage [4] have been adopted for specific application domains. Many of these criteria are motivated by general principles (e.g. you cannot have much confidence in the correctness of a statement without exercising it); others target specific commonly occurring fault types (e.g. boundary value coverage).

Finding a set of test data to achieve identified coverage criteria is typically a labour-intensive activity consuming a good part of the resources of the software development process. Automation of this process can greatly reduce the cost of testing and hence the overall cost of the system. Many automated test data generation techniques have been proposed by researchers. We can broadly classify these techniques into three categories: random, static and dynamic [5] [6].

Random approaches generate test input vectors with elements randomly chosen from appropriate domains. Input vectors are generated until some identified criterion has been satisfied. Random testing may be an effective means of gaining an adequate test set for simple programs but may simply fail to generate appropriate data in any reasonable timeframe for more complex software (or more sophisticated criteria).

With static techniques an enabling condition is typically generated that is satisfied by test data achieving the identified goal. For example, symbolic execution can be used to extract an appropriate path traversal condition for an identified path. Such enabling conditions are solved by constraint solving techniques. However, current application of static code analysis techniques to generate data is not widespread. Despite much research, these approaches do not scale well, and are problematic for some important code elements, such as loops, arrays and pointers [7].

Recently Search Based Software Engineering (SBSE) [8], [9] has evolved as a major research field in the software engineering community. Major work in the area has con-

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.26

centrated on software testing. In fact it is fair to say that search based software testing has, in many respects, lead the way in SBSE. Work can be dated back to 1976 [10]. The major work in search based software testing itself is search based test data generation. McMinn [7] published a survey paper back in 2004. There has been extensive activity in the field since then. A recent survey on the use of search based techniques to test non-functional properties has recently been published. [11].

2. Search Based Test Data Generation

2.1. General Approach

In search based test data generation (SBTDG) achieving a test requirement is modeled as a numerical function optimisation problem and some heuristic is used to solve it. The techniques typically rely on the provision of "guidance" to the search process via feedback from program executions. For example, suppose we seek test data to satisfy the condition $X <= 20$. We can associate with this predicate a cost that measures how close we are to satisfying it, e.g. $cost(X <= 20) = max(X - 20, 0)$. The value $X == 25$ clearly comes closer to satisfying the condition than does $X == 50$, and this is reflected in the lower cost value associated with the former. The problem can be seen as minimising the cost function $cost(X <= 20)$ over the range of possible values of X.

SBTDG for functional testing generally employs a search/optimisation technique with the aim of causing assertions at one or more points in the program to be satisfied. We may require each of a succession of branch predicates to be satisfied (or not) to achieve an identified execution path; we may require the program preconditions to be satisfied but the postconditions to be falsified (i.e. falsification testing — finding test data that breaks the specification) [12]; or else we may simply require a proposed invariant to be falsified (e.g. breaking some safety condition, or causing a function to be exercised outside its precondition) [13].

There has been a growing interest in such techniques and we see more and more applications of these techniques to software testing. Some of the techniques that have been successfully applied to test data generation are Hill Climbing (HC) [10], [14], Simulated Annealing (SA) [15], Genetic Algorithms (GAs) [16], [17], [18], [13], Tabu Search (TS) [6], Ant Colony Optimisation (ACO) [19], Artificial Immune Systems (AISs) [20], Estimation of Distribution Algorithms (EDAs) [21], Scatter Search (SS) [22] and Evolutionary strategies (ESs) [23].

2.2. Limitations of Search Based Approaches

SBTDG revolves around finding data to cause one or a sequence of assertions (predicates) within a program to be satisfied. Current approaches to search based test data generation have explored the use of various optimisation techniques and various fitness functions to find test data satisfying some particular goal. The approaches tried have seen considerable success, but there are also clear limitations.

Even with the predicate structures involving Boolean relational operators, some targets will be difficult to achieve. It may be because of the very small input domain of data satisfying the goal or because the program structures are not very amenable to the search process. For example, handling Boolean flag variables is hard. In such cases one approach has been to transform the program to an equivalent version more suited to SB approaches [24], [25], [26]. Program transformation has been used to make generation of test data easier. Once the data is found the transformed program may be thrown away. Below we take the idea of program transformation further, viewing transformation not as a one-off activity, but as a fundamental part of the search process.

3. Program Stretching

Currently almost all traditional program transformations are semantics preserving. This is not essential [27]; we can obtain the required test data from a transformed program semantically different from the actual program. In this paper we propose a radical generalisation of current transformation approaches, illustrated with respect to branch and path coverage goals.

We will 'stretch' difficult branches thus making them easier to cover. This can be achieved by generating mutant programs by adding auxiliary variables to the predicates in the difficult branches. To clarify the idea consider the following example.

$if\ (expr1 < expr2)$ .....(I)
{
statements
}
$else$
{
statements
}

$if((expr3 > expr4) \wedge (expr5 < expr6))$.....(II)
{
statements
}
$else$
{
statements
}

Lets suppose that (II) is the branch that SBTDG seems

unable to cover. We now add additional input variables, i.e, var1 and var2 to it as shown below.

$if\ (expr1 < expr2)$ .....(I)
{
statements
}
$else$
{
statements
}

$if\ ((expr3 + var1) > expr4 \wedge (expr5 < (expr6 + var2))$.....(II)
{
statements
}
$else$
{
statements
}

Initially we set the values of $var1$ and $var2$ reasonably large so that it is easy (even trivial) to find test data such that $(expr3 + var1) > (expr4)$ and $expr5 < (expr6 + var2)$. The ranges of additional variable values define a set of "stretched" programs. Our search now proceeds over the set of such stretched programs and test inputs for them. The search trajectory comprises a sequence of pairs $\langle (prog_1, td_1), (prog_2, td_2), \ldots, (prog_{final}, td_{final}) \rangle$. The aim is to end up with end up with an "unstretched" program $prog_{final}$ (with all auxiliary variables set to 0) and test data td_{final} that satisfies the required constraints. Essentially, the search space comprises the original test data input space combined with the set of all variable assignments to auxiliary variables. Although we know the desired eventual value of each auxiliary variable (i.e. 0) allowing it to take positive intermediate values can facilitate the solution of the overall problem. Essentially we find test data satisfying our goal for a highly stretched program, and evolve the test input data and auxiliary variables together to achieve the original aim. This requires a very simple modification to existing SBTDG mechanisms.

3.1. Fitness Function

We use the basic fitness function proposed by [15] for our work as shown in Table 1. This is a modified form of the work proposed by [14]. The fitness function is based on evaluating branch predicates. It gives a value of 0 if the branch predicate evaluates to the desired value and a positive value otherwise. The lower the value, the better is the solution. The table indicates the cost for specific assertions. Where more than one assertion is of interest (e.g. when a sequence of branch predicates must be satisfied to follow an identified path) then the basic costs per predicate are combined in some way. For a sequence of assertions to be satisfied we combine them using the following equation;

$$f(x)_{path} = f(x)_{branch_c} + KN$$

Where K is a constant, $branch_c$ is the current branch and N is the number of uncovered branches of the path to be satisfied.

We need also to incorporate the effect of auxiliary variables. For this purpose we add a new term to the fitness function, which is the summation of all the new variables. i.e,

$$f(x)_{total} = f(x)_{path} + \sum_{i=0}^{n} abs(v_i)$$

where v_i is the ith auxiliary variable

We then bring down $f(x)_{total}$ to zero by decreasing the value of $\sum_{i=0}^{n} abs(v_i)$ in a 'controlled' way.

Table 1. Fitness-Functions

Element	Value#1
$a == b$	if $abs(a-b) = 0$ then 0 else $abs(a-b)$+K
$a \neq b$	if $abs(a-b) \neq 0$ then 0 else K
$a < b$	if $(a-b) < 0$ then 0 else $(a-b)$+K
$a \leq b$	if $(a-b) \leq 0$ then 0 else $(a-b)$+K
$a > b$	if $(b-a) < 0$ then 0 else $(b-a)$+K
$a \geq b$	if $(b-a) \geq 0$ then 0 else $(b-a)$+K
$a \vee b$	$min(cost(a), cost(b))$
$a \wedge b$	$cost(a) + cost(b)$

Table 2 shows how assertions in a program may be stretched.

In most cases this is straightforward. For the case of equality $(a == b)$ we first transform to the equivalent
$(a \geq b \wedge b \geq a)$
before mutating.

A zero cost solution to an identified goal provides test data that achieves the goal for the original program. The extension of the cost function to include punishment of non-zero values of additional variables does not assume any particular mechanism for the traditional cost function component. We have chosen the traditional one with which we are most familiar, but others' approaches can be simply extended in the way we suggest.

4. Experiments and Evaluation

To evaluate the validity of the above concept we performed two sets of experiments. In the first set, we consid-

Table 2. Branch Transformation

Predicates	Transformed Form	Remarks
$a == b$	$(a + var1) \geq b \wedge$ $(b + var2) \geq a$	
$a \neq b$	$a \neq b$	No need for transformation
$a < b$	$a < (b + var)$	
$a > b$	$(a + var) > b$	
$a \leq b$	$a \leq (b + var)$	
$a \geq b$	$(a + var) \geq b$	
$a \vee b$	Apply transformation to expr a and\or b using the above rules	
$a \wedge b$	Apply transformation to expr a and\or b using the above rules	

ered three small case studies of program code of varied McCabe structural complexity. These programs are given in Appendix B. In the second set of experiments, we applied the concept at the architectural level to MATLAB® Simulink® models. We compared performance of the programs on the basis of (i) coverage, i.e. input test data found for branches against total branches to satisfy a coverage criterion, (ii) success rate, i.e. how many times test data was found when a program was run multiple times to satisfy a coverage criterion and, (iii) the average number of executions taken to find the test data.

4.1. Experiment set 1: Code Examples

We used Simulated Annealing (SA) with the following parameters for our first set of experiments . Further experiments can be done to optimise the parameters. However, since we kept the same parameters for both approaches, we believe the results will be similar for the optimise set as well.

Move strategy: fixed

Geometric Temperature decrease rate: 0.8

Number of iteration in inner loop: 250

Maximum number of iteratoin in outer loop: 1000

Stopping criterion: Either a solution is found or maximum number of iterations are reached.

Case study 1 is relatively easy having a McCabe structural complexity [28] of 10 and is experimented with to find the applicability of the stretching principle to different relational operators. Case Study 2 though has a McCabe structural complexity of 8 but is relatively difficult from a search point of view because of the branching structure. Case study 3 has a McCabe structural complexity of 14 and the branching structure is more difficult for search. In each of the above programs, our goal is to achieve the coverage of the last branch as indicated in the appendix. To reach that goal we 'stretched' the intermediate difficult branches.

With program 1, we were able to obtain 100% coverage and success rate using the traditional search based approach and also with program stretching. However, as expected, the average number of iterations to cover the required target in stretched program approach were more than for the traditional approach. Although the goal is satisfied more quickly with the stretched program approach, the search process must expend additional effort reducing the program to the original and dragging the test data with it to maintain goal satisfaction. For easy goals, we believe that the traditional approach should be used instead of program stretching.

In program 2, the search was again 100% successful in each case. However, the task here is more difficult and the results are better for the program stretching approach. Program stretching, in this case, would appear to give efficiency advantages.

In program 3, the results were even better. In this case the search process was successful in 92% of runs for the standard approach and 99.8% of runs using program stretching. The average number of function evaluations (program executions) for successful runs were also better for the program streching approach.

Table 3 and 4 summarise the above results.

Table 3. Number of executions to generate test data Via Standard SBTDG (Based on 500 runs)

	Program1	Program2	Program 3
Average	4825	14248	78212
Max	15037	24315	250050
Min	451	2663	26604
SD	2876	5564	58747
Success Rate(%)	100	100	92

Table 4. Number of executions to generate test data via Program Stretching (Based on 500 runs)

	Program1	Program2	Program 3
Average	7222	11976	61960
Max	24900	18405	140606
Min	1341	5514	14411
SD	3671	2195	35853
Success Rate(%)	100	100	99.8

4.2. Experiment set 2: Simulink Models

As stated above, for the second set of experiments we used MATLAB Simulink models. Appendix A gives a short introduction to Simulink. For this set of experiments we used the models from Zhan and Clark [29], which used input variables of type double with a range of -100 to 100. We used the following configuration for SA parameters. The parameters setting is based on the experimental work by Zhan and Clark[30].

Move strategy: fixed.

Geometric Temperature decrease rate: 0.9
No of inner loop iterations: 100
Max. No. of outer loop iterations: 300
Stopping criteria: Either a solution is found or maximum number of iterations are reached.

We considered only those branches for analysis for which the traditional approach was not hundred percent successful. Zhan and Clark [29] used four models. Model *SmplSw* is rather straightforward and hence is not considered for experiments. *Quadratic* model is also relatively simple but the search is made more difficult by introducing local minima. This was achieved by changing the range of input variables.

Quadratic contains three Switch blocks and hence eight 'paths' or, more specifically combinations. *RandMdl* has four Switch blocks and hence has sixteen combinations. *Combine* has seven Switch blocks and hence one hundred and twenty eight combinations. We targeted each of these combinations in the case of *Quadratic* and *RandMdl*. In case of *Combine* model, we targeted only the case where all switch blocks take the value 'true'. Each model was run thirty times for each of the targets to achieve statistically significant results. For example, for *RandMdl*, we did four hundred and eighty runs in total.

Quadratic model gave much better results for the stretching approach. However, for models *RandMdl* and *Combine* the results in both approaches are very similar in terms of success rate. But the 'stretching' is more expensive as it requires more number of execution to find the input data to cover the required combination. The results have been summarised in Tables 5 and 6.

In both set of experiments, the neighbourhood is searched by considering a small change in any of the variables, either actual or auxiliary, and then the modified fitness function is evaluated. The move is accepted, if the fitness function value is improved. However, by restricting the neighbourhood search, we can get significant improvement in the results. In such case when the value of auxiliary variable is changed, the actual variable is also changed by a fraction of that value. The actual fitness function due to all the variables, actual or auxiliary, as well as the modified fitness function are evaluated. The move is accepted if both fitness functions are improved or if there is an improvement in any of the fitness functions but the other remains unchanged. Table 7 summarises the results for Simulink models for this strategy. We can see that there is a clear improvement in the average number of executions as well as the success rate.

Table 5. SBTDG for Simulink Models without Program Stretching (Based on 30 runs)

	Quadratic	RandMdl	Combine
Total No of Branches	8	16	116
No of Branches for Analysis	4	12	2
Mean Success Rate(%)	49.16	51.9	33
Coverage(%)	100	100	100
Mean No of Executions	587	1931	3122

Table 6. SBTDG for Simulink Models with Program Stretching (Based on 30 runs)

	Quadratic	RandMdl	Combine
Success Rate(%)	90.83	50	36.333
Coverage(%)	100	100	100
Mean No of Executions	5769	5230	7350

5. Previous Work

Program Transformation has been applied to programs most commonly to the removal of problematic program elements such as GOTO statement [31], [32]. In the realm of software testing the work by Harman et al [27], based on previous work [25] is noteworthy. They proposed a simple theory of testability transformation and applied the technique to the removal of flag problem for Evolutionary test data generation. The flag-free programs provided more determinism to the search process improving both the performance of evolutionary test data generation and the adequacy level of the test data so-generated. Hierons et al. [24] applied the technique to transformation of a program with one or more `exit` statement to a branch-coverage equivalent program thus preserving the branch-coverage adequate sets of test inputs. McMinn et al [26] applied the approach to the nested search targets. The approach was applied to the nested-if statement by converting the nested structure to a straight line one. Case studies were performed on two small programs which showed an improvement actor of more than two times in terms of efficiency in the search process. However, the technique still faces challenges posed by some common program constructs: loops, arrays and pointers.

6. Conclusions and Future Work

The program stretching principle is in its initial stages. Initial results are promising and we believe that the technique can be extended to more complex systems. The technique was motivated by the need to satisfy "difficult" goals. The studies indeed show that program stretching has advantages in these cases, either showing greater success or greater efficiency (or both) than standard search based test data generation.

The approach is basically a form of continuous program transformation. A key difference here is that the transformations do not preserve program semantics. The idea of stretching was originally motivated as a form of "topological" deformation. Ideally the input domains corresponding to

Table 7. SBTDG for Simulink Models with Program Stretching and 'Controlled' Neighbourhood (Based on 30 runs)

	Quadratic	RandMdl	Combine
Success Rate(%)	90.83	92.33	77
Coverage(%)	100	100	100
Mean No of Executions	3885	6018	6534

satisfying test goals in the stretched program would map in a straightforward way to those of the original program's test goals. This is not essential. One could easily imagine some input partitions vanishing as the program is stretched — some paths through the program may no longer be possible for example.

This paper has shown how program stretching can be used to find hard-to-find branches, but really this is about generating test data to satisfy hard-to-find conditions. This can easily be extended to the following applications:

Invariant Falsification: Suppose we have a proposed invariant inv for some identified point in the program. If we insert a program statement "if(!inv);" at that point we can view the falsification of the invariant as a branch reachability problem. As a consequence can easily apply the program stretching technique.

Exception Generation: Consider assignment statements of the form "var=expr;". When expr is evaluated the result must be a value inside the type bounds,if an exception is not to be raised. In program reasoning it is common to refer to refer to such constraints as healthiness pre-conditions. We can insert a statement of the form "if(!healthiness); and proceed much as before.

Higher Level Models: In this paper we attempted application of program stretching to the 'path coverage' or more specifically to the 'Combination' coverage [29] of Simulink models. Zhan and Clark [30] also proposed techniques for a more practical branch coverage criterion. We also aim at extending the approach to such criteria. We also believe that our stretching technique may find applications to other higher level models such as statecharts. Similarly, automated SBTDG from specification might be facilitated.

In future we aim to further explore the idea and identify the program structures where we can apply this technique with a greater degree of confidence. The paper has provided an indication that the stretching concept may be used to enable difficult test data to be found more easily. Our evaluation has been limited to a few programs. We believe that more extensive testing is clearly necessary to come to definitive conclusions about the utility of the approach. We have used a rather blunt notion of stretching (adding an auxiliary variable to one side of a relational expression). It remains an open question whether more sophisticated approaches to stretching can be found.

7. Application Outside SBSE

We believe that program stretching is a novel yet appealing concept. Although developed to solve a specific problem in SBTDG, it seems plausible that the approach could find application elsewhere. In abstract terms the fundamental idea is that the "problem" and "solution" are developed together. The problem is essentially relaxed (made easier) until a satisficing solution is found. An attempt is then made to migrate the problem to the original problem of interest with the solution being "dragged" along to maintain satisfaction of the changing constraints. We see no reason in principle why this form of relaxing and dragging should not find wider application within the search based engineering disciplines. We recommend this area to the research community.

Appendix

1. Simulink

Simulink is a software package for modelling, simulating, and analysing system-level designs of dynamic systems. Simulink models/systems are made up of blocks connected by lines. Each block implements some function on its inputs and outputs the results. Outputs of blocks form inputs to other blocks (represented by lines joining the relevant input/output ports). Models can be hierarchical. Each block can be a subsystem comprising other blocks and lines.

Simulink models have their special way of forming branches compared to programs. Blocks such as 'if-else', and Switch are used to form branches. We have used only Switch blocks in our work for branching structure. However, the work can be easily extended to include other blocks as well. A Switch block has three 'in' ports, one 'out' port and a control parameter 'Threshold'. When the value of the second 'in' port is greater than or equal to the threshold parameter, the output will equal to the value carried on the first 'in' port, otherwise the value carried on the third 'in' port will be channelled through to the output. Therefore a Switch block can map to an 'if-then-else' branching structure in code.

Two other important blocks that need to be introduced here are LogicalOperator and RelationalOperator blocks.

A LogicalOperator block has two parameters: operator parameter (which can be 'AND', 'OR', 'NAND', 'NOR', 'XOR', or 'NOT') and input-number parameter (which can be any integer number except that when the operator

parameter is 'NOT', in which case, the input-number must be '1').

A RelationalOperator block has two inputs. There is a parameter defining the desired relation between the two inputs. If the relation is TRUE, the output will be '1'; otherwise, the output will be '0'. There are six options for the relational parameter: ==, $\sim=$, $<$, $<=$, $>=$, and $>$. Figure 1. is an example of a Simulink model showing the above mentioned blocks. This model has three input variables; 'IN-A', 'IN-B' and 'IN-C'. The 'Product' block multiply all it's inputs.The model calculates if an equation of the form:

$$ax^2 + bx + c = 0$$

is a quadratic equation and if it has real-valued solution(s). If it is a quadratic equation and it has one or two real-valued roots, '1' is output; otherwise, the output is '-1'

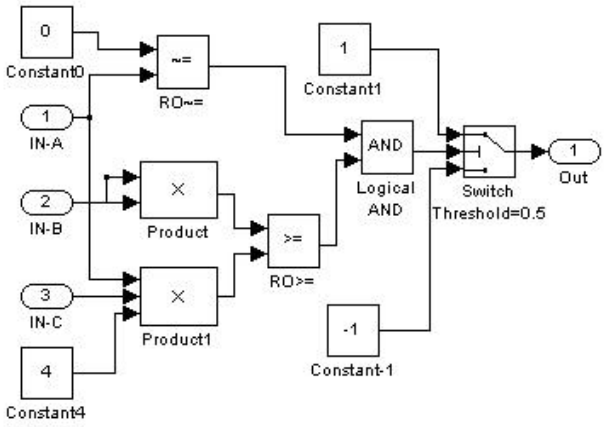


Figure 1. A Simulink Model for the Quadratic Equation.

Simulink models execute (calculate the outputs of) all branches of the models, whether the branches are selected or not, while for programs, only the selected branches are executed. For example, the following code matches the model in Figure 2.

```
program calculation;
 input x,y;
 output z;
    begin
       if x>=y
         z=x-y;
       else
        z=y-x;
     end
```

In the Simulink model, both *'x-y'* and *'y-x'* are calculated although only one of these results is channelled through to the output by the Switch block. However, in the code, only one of them will be executed depending on the evaluation of the predicate *'$x \leq y$'*.

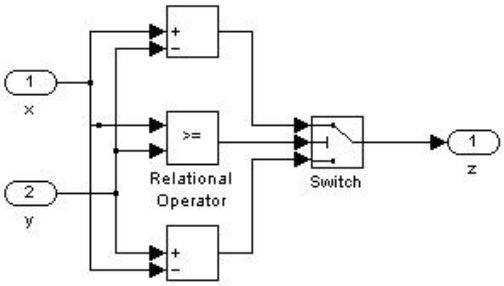


Figure 2. A Simple Simulink Model

2. SBTDG for Simulink Models

Simulink has been popularly used as a higher level system prototyping or design tool in many domains, including aerospace, automobile and electronics systems. This facilitates investigation (e.g. for both verification and validation purposes as well as optimisation) of the system under consideration at an early stage of development. Code can then be generated either manually or automatically. Simulink is playing a more and more important role in system engineering and the verification and validation of Simulink models is becoming vital to users. SBTDG techniques have seen little application to Simulink models, which is surprising since the execution model of Simulink would seem to allow analogous SBTDG techniques to be applied as for code. The only work that is known to us, to the best of our knowledge, is by Zhan and Clark [29], [33], [34]. The work in this paper is based on their earlier work [29].

3. Program 1

```
public class Program1  {


 public void opCheck(int j, int k,
int l, int m){

  if ((j>10) && (j< 15)){

  System.out.println("j>10 && j<15:
executed");


  if (k== j) {

   System.out.println("k=j:executed");

  if ((l!=k) && (m>=j) &&(m==k)) {
```

```
   System.out.println("Target
Reached"); ..........(1)
     }
    }
  }
 }
}
```

3.1. Transformed Program 1.

```
public class Program1Var {

 public void opCheck(int x, int y,
                 int z,int m,
                 int var1,int var2,
                 int var3,int var4){

  if (x+var1>10 && x-var2<15){

    System.out.println("x>10 &&
x<15: executed");

   if (y+var3>=x&&x+var3>=y){

    System.out.println("y==x:executed");

      if (z !=y && m>=x && m==y){

           System.out.println("Target
         reached");
     }
   }
  }
 }
}
```

4. Program 2

```
public class Program2 {

 public void complexCheck(int x, int y, int z) {
   if ((x<10) &&(y>1000) && (z>999)) {

   System.out.println("Branch 1 executed");
  }
  else if ((x > 10) &&(y > 1000) && (z> 999)){

   System.out.println("Branch 1
else-if executed");

   if (((z + y) < 2003) && (x + y< 1400)
&& (x > 390)) {
    System.out.println
   ("Target reached");..........(2)

   }
  }
  else {

System.out.println("the false branch taken");
   }
 }
}
```

4.1. Transformed Program 2.

```
public class Program2Var {

 public void complexCheck(int x, int y,
                    int z, int var1,
                    int var2,int var3){

  if ((x<10) &&(y>1000)
&& (z>999)){

   System.out.println("Branch-1 executed");

   }

  else if ((x+var1 > 10) &&(y+var2 > 1000)
  &&(z+var3> 999)){

   System.out.println("Branch-1
else-if executed");

   if (((z + y) < 2003) && (x + y< 1400)
                              && (x > 390)){

    System.out.println
   ("Target reached");
   }
  }
  else  {

   System.out.println("False branch taken");
  }
}
}
```

5. Program 3

```
public class Program3 {

  public void complexCheck(int a, int b,
int c, int d, int e)  {

  if ((a<10) && (b>1000)&&(c>999  )) {

    System.out.println("Branch-1
 executed");

  }
  else if ((a>10) && (b>1000)&& (c>999)
&& (d>999&& d<1001)) {

   System.out.println("Branch-1 else
 if executed");

   if (c+b<2003&& a+b<1400&&a>390) {

    System.out.println("Inner if
executed");

    if (d+e>2000 && d+e<2003){

     System.out.println
   ("Target reached");..........(3)
   }
  }
}
   else {
```

```
   System.out.println("False
 branch taken");
  }
}
}
```

5.1. Transformed Program 3.

```
public class Program3Var {

 public void complexCheck(int a, int b,
int c,int d,
int e, int var1,
int var2,int var3,
int var4,int var5,
          int var6)  {

  if ((a<10) &&(b> 1000) &&(c> 999)) {

   System.out.println("Branch-1
 executed");

  }
  else if (((a+var1> 10) &&
(b+var2 >1000)) &&
                c+var3> 999)&&
               (d+var4>999)&&
               (d<1001)) {

   System.out.println("Branch-1
else if executed");

   if ((c + b<2003+var5) &&
(a + b<1400)&&(a+var6>390)){

    System.out.println("Inner if
executed");

    if ((d + e> 2001) && (d + e<2003)) {

     System.out.println("Target
reached");
    }
   }
  }
  else {
   System.out.println("False
branch taken");
  }
 }
}
```

References

[1] IEEE, *Guide to software engineering body of knowledge*, 2004, ch. 5, p. 1.

[2] G. Myers, *The Art of Software Testing*. John Wiley & Sons, 1979.

[3] S. Rapps and E. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions of Software Engineering*, vol. 11, no. 4, pp. 367– 375, April 1985.

[4] RTCA, "Software considerations in airborne systems and equipment certification," pp. 2455–2464, 1992.

[5] C. C. Michael, G. McGraw, M. Schatz, and C. C. Walton, "Genetic algorithms for dynamic test data generation," in *Automated Software Engineering*, 1997, pp. 307–308. [Online]. Available: citeseer.ist.psu.edu/michael97genetic.html

[6] E. Diaz, J. Tuya, and R. Blanco, "Automated software testing using a metaheuristic technique based on tabu search," in *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, Oct. 2003, pp. 310 – 313. [Online]. Available: citeseer.ist.psu.edu/diaz03automated.html

[7] P. McMinn, "Search-based software test data generation: a survey: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[8] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem," *IEE Proceedings — Software*, vol. 150, no. 3, pp. 161–175, 2003.

[9] M. Harman, "The current state and future of search based software engineering," in *Future of Software Engineering 2007*, L. Briand and A. Wolf, Eds., Los Alamitos, California, USA, 2007, pp. 342–357.

[10] W. Miller and D. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 223–226, Sept. 1976.

[11] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, 2009, in Press, Corrected Proof. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0B-4VHXDTD-1/2/9da989f9d874eb88d1f82d9a0878114b

[12] N. Tracey and J. C. K. Mander, "Automated program flaw finding using simulated annealing'," in *proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, Clearwater Beach, Florida, United States, 1998, pp. 73 – 81.

[13] N. J. Tracey, "A search-based automated test-data generation framework for safety-critical softwares," DPhil, University of York, 2000.

[14] B. Korel, "Automated software test data generation." *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 870–879, 1990. [Online]. Available: http://www.computer.org/tse/ts1990/e0870abs.htm

[15] N. Tracey, J. Clark, K. Mander, and J. A. McDermid, "An automated framework for structural test-data generation," in *Automated Software Engineering*, 1998, pp. 285–288.

[16] B. F. Jones, H. H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, vol. 11, no. 5, pp. 299–306, Sep 1996.

[17] J. T. Alander, T. Mantere, and P. Turunen, "Genetic algorithm based software testing," in *Artificial Neural Nets and Genetic Algorithms*. Wien, Austria: Springer-Verlag, 1998, pp. 325–328.

[18] R. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Journal of Software Testing, Verification and Reliability*, vol. 9, no. 3, pp. 263–282, Sep 1999.

[19] H. Li and C. P. Lam, "Software test data generation using ant colony optimization." in *International Conference on Computational Intelligence*, 2004, pp. 1–4.

[20] P. May, K. Mander, and J. Timmis, "Mutation testing: An artificial immune system approach," in *UK-Softest. UK Software Testing Workshop*, University of York. UK., September 2003. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2003/1700

[21] R. Sagarna and J. Lozano, "On the performance of estimation of distribution algorithms applied to software testing." *Applied Artificial Intelligence.*, vol. 19, no. 5, pp. 457–489., 2005.

[22] Sagarna and J. Lozano, "Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms." *European Journal of Operational Research*, vol. 169, no. 2, pp. 392–412., 2006.

[23] E. Alba and J. Chicano, "Software testing with evolutionary strategies." in *The 2nd Workshop on Rapid Integration of Software Engineering Techniques (RISE05)*, vol. 169, no. 2, Heraklion, Greece, September 2005., pp. 50–65.

[24] R. M. Hierons, M. Harman, and C. J. Fox, "Branch-coverage testability transformation for unstructured programs," *Comput. J.*, vol. 48, no. 4, pp. 421–436, 2005.

[25] M. Harman, L. Hierons, A. Baresel, and H. Sthamer, "Improving evolutionary testing by flag removal," in *Genetic and Evolutionary Computation Conference (GECCO 2002).*, 2002. [Online]. Available: citeseer.ist.psu.edu/harman02improving.html

[26] P. McMinn, D. Binkley, and M. Harman, "Testability transformation for efficient automated test data search in the presence of nesting," 2005.

[27] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.

[28] McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, 1976.

[29] Y. Zhan and J. A. Clark, "Search based automatic test-data generation at an architectural level," in *Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO'04)*. Springer, 2004, pp. 1413–1424.

[30] Y. Zhan, "Search based test data generation for simulink models," Ph.D. dissertation, University of York, 2005.

[31] L. Ramshaw, "Eliminating go to's while preserving program structure," *J. ACM*, vol. 35, no. 4, pp. 893–920, 1988.

[32] A. M. Erosa and L. J. Hendren, "Taming control flow: A structured approach to eliminating goto statements," in *ICCL, IEEE Computer Society 1994 International Conference on Computer Languages*, Toulouse, France, May 1994, pp. 229–240.

[33] Y. Zhan and J. Clark, "Search-based mutation testing for simulink models," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05)*. ACM, 2005, pp. 1061–1068.

[34] Y. Zhan and J. A. Clark, "The state problem for test generation in simulink," in *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO '06)*. ACM, 2006, pp. 1941–1948.

Author Index

IEEE Computer Society Conference Publications Operations Committee

IEEE Computer Society Publications

The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets. Visit the CS Store at *http://www.computer.org/portal/site/store/index.jsp* for a list of products.

IEEE Computer Society *Conference Publishing Services* (CPS)

The IEEE Computer Society produces conference publications for more than 250 acclaimed international conferences each year in a variety of formats, including books, CD-ROMs, USB Drives, and on-line publications. For information about the IEEE Computer Society's *Conference Publishing Services* (CPS), please e-mail: cps@computer.org or telephone +1-714-821-8380. Fax +1-714-761-1784. Additional information about *Conference Publishing Services* (CPS) can be accessed from our web site at: *http://www.computer.org/cps*

Revised: 1 March 2009

CPS Online is our innovative online collaborative conference publishing system designed to speed the delivery of price quotations and provide conferences with real-time access to all of a project's publication materials during production, including the final papers. The ***CPS Online*** workspace gives a conference the opportunity to upload files through any Web browser, check status and scheduling on their project, make changes to the Table of Contents and Front Matter, approve editorial changes and proofs, and communicate with their CPS editor through discussion forums, chat tools, commenting tools and e-mail.

The following is the URL link to the ***CPS Online*** Publishing Inquiry Form:
http://www.ieeeconfpublishing.org/cpir/inquiry/cps_inquiry.html