Conferences in Research and Practice in Information Technology

Volume 135

Computer Science 2013

Australian Computer Science Communications, Volume 35, Number 1

Computer Science 2013

Proceedings of the
Thirty-Sixth Australasian Computer Science Conference
(ACSC 2013), Adelaide, Australia,
29 January – 1 February 2013

Bruce Thomas, Ed.

Volume 135 in the Conferences in Research and Practice in Information Technology Series.
Published by the Australian Computer Society Inc.

Published in association with the ACM Digital Library.

Computer Science 2013. Proceedings of the Thirty-Sixth Australasian Computer Science Conference (ACSC 2013), Adelaide, Australia, 29 January – 1 February 2013

Conferences in Research and Practice in Information Technology, Volume 135.

Editor:

Bruce Thomas
School of Computer and Information Science
Division of Information Technology, Engineering and the Environment
University of South Australia
Adelaide, SA 5001
Australia
Email: `bruce.thomas@unisa.edu.au`

Series Editors:
Vladimir Estivill-Castro, Griffith University, Queensland
Simeon J. Simoff, University of Western Sydney, NSW
Email: `crpit@scm.uws.edu.au`

Publisher: Australian Computer Society Inc.
PO Box Q534, QVB Post Office
Sydney 1230
New South Wales
Australia.

Conferences in Research and Practice in Information Technology, Volume 135.
ISSN 1445-1336.
ISBN 978-1-921770-20-3.

Document engineering, January 2013 by CRPIT
On-line proceedings, January 2013 by the University of Western Sydney
Electronic media production, January 2013 by the University of South Australia

The *Conferences in Research and Practice in Information Technology* series disseminates the results of peer-reviewed research in all areas of Information Technology. Further details can be found at http://crpit.com/.

Table of Contents

Preface

The Australasian Computer Science Conference (ACSC) series is an annual meeting, bringing together research sub-disciplines in Computer Science. The conference allows academics and other researchers to discourse research topics as well as progress in the field, and policies to stimulate its growth. This conference is unique in its ability to provide a platform for cross-disciplinary research. This volume comprises papers being presented at the Thirty-Sixth ACSC in Adelaide, Australia. ACSC 2013 is part of the Australasian Computer Science Week which runs from January 29 to February 1, 2013.

The ACSC 2013 call for papers solicited 29 submissions from Australia, New Zealand, Iran, Slovenia, Korea, Spain, United Kingdom, Denmark, Germany, India, Japan, China and Thailand. The topics addressed by the submitted papers illustrate the broadness of the discipline. These included algorithms, virtualisation, software visualisation, databases, constraint programming and image processing, computer architecture, compression, to name just a few.

The programme committee consisted of 39 highly regarded academics from Australia, New Zealand, Italy, Japan, Sweden, China, Canada, and USA. Every paper was reviewed by at least three programme committee members, and, in some cases, external reviewers. Of the 29 papers submitted, 14 were selected for presentation at the conference.

The Programme Committee determined that the ”Best Paper Award” should go to Marcus Brazil and Martin Zachariasen Castro for their paper entitled Computational Complexity for Uniform Orientation Steiner Tree Problems. Congratulations.

We thank all authors who submitted papers and all conference participants for helping to make the conference a success. We also thank the members of the programme committee and the external referees for their expertise in carefully reviewing the papers. We are grateful to Professor Simeon Simoff from Univerity of Western Sydney representing CRPIT for his assistance in the production of the proceedings. I thank Professor Tom Gedeon (President) for his support representing CORE (the Computing Research and Education Association of Australasia).

Thanks to the School of Computer and Information Science at The University of South Australia for web support for advertising the conference.

Last, but not least, we express gratitude to our hosts at the University of South Australia and, in particular, Dr. Ivan Lee.

Bruce Thomas
University of South Australia

ACSC 2013 Programme Chair
January 2013, Adelaide, Australia

Programme Committee

Chair

Bruce Thomas, University of South Australia

Members

Matt Adcock, CSIRO (Australia)
Leila Alem, CSIRO (Australia)
Stephane Bressan, National University of Singapore (Singapore)
Fred Brown, University of Adelaide (Australia)
Paul Calder, Flinders University (Australia)
Ash Doshi, University of South Australia (Australia)
Curtis Dyreson, Utah State University (USA)
Julien Epps, University of New South Wales (Australia)
Ken Hawick, Massey University (New Zealand)
Michael E. Houle, National Institute of Informatics (Japan)
Zhiyi Huang, University of Otago (New Zealand)
Tony Huang, CSIRO (Australia)
Shuji Kijima, Kyushu University (Japan)
Paddy Krishnan, Bond University (Australia)
Jiuyong Li, University of South Australia (Australia)
Jixue Liu, University of South Australia (Australia)
Lin Liu, University of South Australia (Australia)
Michael Marner, University of South Australia (Australia)
Wolfgang Mayer, University of South Australia (Australia)
Chris McDonald, University of Western Australia (Australia)
Linda Pagli, University of Pisa (Italy)
Maurice Pagnucco, University of New South Wales (Australia)
Jun Park, Magic Vision Lab, University of South Australia (Australia)
Alex Potanin, Victoria University of Wellington (New Zealand)
Yuping Shen, Institute of Logic and Cognition, Department of Philosophy, Sun Yat-sen University (China)
Mark Smith, Kungliga Tekniska Högskolan (Sweden)
Aaron Stafford, University of South Australia (Australia)
Markus Stumptner, University of South Australia (Australia)
Maki Sugimoto, KEIO University (Japan)
Ewan Tempero, University of Auckland (New Zealand)
David Toman, University of Waterloo (Canada)
Andrew Turpin, University of Melbourne (Australia)
Rudi Vernik, University of South Australia (Australia)
Burkhard C. Wüensche, University of Auckland (New Zealand)
Hua Wang, University of Southern Queensland (Australia)
Xiaofang Zhou, University of Queensland (Australia)
Jianlong Zhou, University of South Australia

Additional Reviewers

Mykola Grechaniuk
Georg Grossmann
Armin Haller
Thomas Köhne
Masud Karim
Selasi Kwashie
Sean Li
Stuart Marshall
Ravi Rao
Peter Schachte
Ian Welch

Organising Committee

Chair

Dr. Ivan Lee

Finance Chair

Dr. Wolfgang Mayer

Publication Chair

Dr. Raymond Choo

Local Arrangement Chair

Dr. Grant Wigley

Registration Chair

Dr. Jinhai Cai

Welcome from the Organising Committee

On behalf of the Organising Committee, it is our pleasure to welcome you to Adelaide and to the 2013 Australasian Computer Science Week (ACSW 2013). Adelaide is the capital city of South Australia, and it is one of the most liveable cities in the world. ACSW 2013 will be hosted in the City West Campus of University of South Australia (UniSA), which is situated at the north-west corner of the Adelaide city centre.

ACSW is the premier event for Computer Science researchers in Australasia. ACSW2013 consists of conferences covering a wide range of topics in Computer Science and related area, including:

– Australasian Computer Science Conference (ACSC) (Chaired by Bruce Thomas)
– Australasian Database Conference (ADC) (Chaired by Hua Wang and Rui Zhang)
– Australasian Computing Education Conference (ACE) (Chaired by Angela Carbone and Jacqueline Whalley)
– Australasian Information Security Conference (AISC) (Chaired by Clark Thomborson and Udaya Parampalli)
– Australasian User Interface Conference (AUIC) (Chaired by Ross T. Smith and Burkhard C. Wünsche)
– Computing: Australasian Theory Symposium (CATS) (Chaired by Tony Wirth)
– Australasian Symposium on Parallel and Distributed Computing (AusPDC) (Chaired by Bahman Javadi and Saurabh Kumar Garg)
– Australasian Workshop on Health Informatics and Knowledge Management (HIKM) (Chaired by Kathleen Gray and Andy Koronios)
– Asia-Pacific Conference on Conceptual Modelling (APCCM) (Chaired by Flavio Ferrarotti and Georg Grossmann)
– Australasian Web Conference (AWC2013) (Chaired by Helen Ashman, Michael Sheng and Andrew Trotman)

In additional to the technical program, we also put together social activities for further interactions among our participants. A welcome reception will be held at Rockford Hotel's Rooftop Pool area, to enjoy the fresh air and panoramic views of the cityscape during Adelaide's dry summer season. The conference banquet will be held in Adelaide Convention Centre's Panorama Suite, to experience an expansive view of Adelaide's serene riverside parklands through the suite's seamless floor to ceiling windows.

Organising a conference is an enormous amount of work even with many hands and a very smooth cooperation, and this year has been no exception. We would like to share with you our gratitude towards all members of the organising committee for their dedication to the success of ACSW2013. Working like one person for a common goal in the demanding task of ACSW organisation made us proud that we got involved in this effort. We also thank all conference co-chairs and reviewers, for putting together conference programs which is the heart of ACSW. Special thanks goes to Alex Potanin, who shared valuable experiences in organising ACSW and provided endless help as the steering committee chair. We'd also like to thank Elyse Perin from UniSA, for her true dedication and tireless work in conference registration and event organisation. Last, but not least, we would like to thank all speakers and attendees, and we look forward to several stimulating discussions.

We hope your stay here will be both rewarding and memorable.

Ivan Lee
School of Information Technology & Mathematical Sciences

ACSW2013 General Chair
January, 2013

CORE - Computing Research & Education

CORE welcomes all delegates to ACSW2013 in Adelaide. CORE, the peak body representing academic computer science in Australia and New Zealand, is responsible for the annual ACSW series of meetings, which are a unique opportunity for our community to network and to discuss research and topics of mutual interest. The original component conferences - ACSC, ADC, and CATS, which formed the basis of ACSW in the mid 1990s - now share this week with eight other events - ACE, AISC, AUIC, AusPDC, HIKM, ACDC, APCCM and AWC which build on the diversity of the Australasian computing community.

In 2013, we have again chosen to feature a small number of keynote speakers from across the discipline: Riccardo Bellazzi (HIKM), and Divyakant Agrawal (ADC), Maki Sugimoto (AUIC), and Wen Gao. I thank them for their contributions to ACSW2013. I also thank invited speakers in some of the individual conferences, and the CORE award winner Michael Sheng (CORE Chris Wallace Award). The efforts of the conference chairs and their program committees have led to strong programs in all the conferences, thanks very much for all your efforts. Thanks are particularly due to Ivan Lee and his colleagues for organising what promises to be a strong event.

The past year has been turbulent for our disciplines. ERA2012 included conferences as we had pushed for, but as a peer review discipline. This turned out to be good for our disciplines, with many more Universities being assessed and an overall improvement in the visibility of research in our disciplines. The next step must be to improve our relative success rates in ARC grant schemes, the most likely hypothesis for our low rates of success is how harshly we assess each others' proposals, a phenomenon which demonstrably occurs in the US NFS. As a US Head of Dept explained to me, "in CS we circle the wagons and shoot within".

Beyond research issues, in 2013 CORE will also need to focus on education issues, including in Schools. The likelihood that the future will have less computers is small, yet where are the numbers of students we need? In the US there has been massive growth in undergraduate CS numbers of 25 to 40% in many places, which we should aim to replicate. ACSW will feature a joint CORE, ACDICT, NICTA and ACS discussion on ICT Skills, which will inform our future directions.

CORE's existence is due to the support of the member departments in Australia and New Zealand, and I thank them for their ongoing contributions, in commitment and in financial support. Finally, I am grateful to all those who gave their time to CORE in 2012; in particular, I thank Alex Potanin, Alan Fekete, Aditya Ghose, Justin Zobel, John Grundy, and those of you who contribute to the discussions on the CORE mailing lists. There are three main lists: csprofs, cshods and members. You are all eligible for the members list if your department is a member. Please do sign up via http://lists.core.edu.au/mailman/listinfo - we try to keep the volume low but relevance high in the mailing lists.

I am standing down as President at this ACSW. I have enjoyed the role, and am pleased to have had some positive impact on ERA2012 during my time. Thank you all for the opportunity to represent you for the last 3 years.

Tom Gedeon

President, CORE
January, 2013

ACSW Conferences and the Australian Computer Science Communications

The Australasian Computer Science Week of conferences has been running in some form continuously since 1978. This makes it one of the longest running conferences in computer science. The proceedings of the week have been published as the *Australian Computer Science Communications* since 1979 (with the 1978 proceedings often referred to as *Volume 0*). Thus the sequence number of the Australasian Computer Science Conference is always one greater than the volume of the Communications. Below is a list of the conferences, their locations and hosts.

2014. Volume 36. Host and Venue - AUT University, Auckland, New Zealand.

2013. **Volume 35. Host and Venue - University of South Australia, Adelaide, SA**.

2012. Volume 34. Host and Venue - RMIT University, Melbourne, VIC.
2011. Volume 33. Host and Venue - Curtin University of Technology, Perth, WA.
2010. Volume 32. Host and Venue - Queensland University of Technology, Brisbane, QLD.
2009. Volume 31. Host and Venue - Victoria University, Wellington, New Zealand.
2008. Volume 30. Host and Venue - University of Wollongong, NSW.
2007. Volume 29. Host and Venue - University of Ballarat, VIC. First running of HDKM.
2006. Volume 28. Host and Venue - University of Tasmania, TAS.
2005. Volume 27. Host - University of Newcastle, NSW. APBC held separately from 2005.
2004. Volume 26. Host and Venue - University of Otago, Dunedin, New Zealand. First running of APCCM.
2003. Volume 25. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Adelaide Convention Centre, Adelaide, SA. First running of APBC. Incorporation of ACE. ACSAC held separately from 2003.
2002. Volume 24. Host and Venue - Monash University, Melbourne, VIC.
2001. Volume 23. Hosts - Bond University and Griffith University (Gold Coast). Venue - Gold Coast, QLD.
2000. Volume 22. Hosts - Australian National University and University of Canberra. Venue - ANU, Canberra, ACT. First running of AUIC.
1999. Volume 21. Host and Venue - University of Auckland, New Zealand.
1998. Volume 20. Hosts - University of Western Australia, Murdoch University, Edith Cowan University and Curtin University. Venue - Perth, WA.
1997. Volume 19. Hosts - Macquarie University and University of Technology, Sydney. Venue - Sydney, NSW. ADC held with DASFAA (rather than ACSW) in 1997.
1996. Volume 18. Host - University of Melbourne and RMIT University. Venue - Melbourne, Australia. CATS joins ACSW.
1995. Volume 17. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Glenelg, SA.
1994. Volume 16. Host and Venue - University of Canterbury, Christchurch, New Zealand. CATS run for the first time separately in Sydney.
1993. Volume 15. Hosts - Griffith University and Queensland University of Technology. Venue - Nathan, QLD.
1992. Volume 14. Host and Venue - University of Tasmania, TAS. (ADC held separately at La Trobe University).
1991. Volume 13. Host and Venue - University of New South Wales, NSW.
1990. Volume 12. Host and Venue - Monash University, Melbourne, VIC. Joined by Database and Information Systems Conference which in 1992 became ADC (which stayed with ACSW) and ACIS (which now operates independently).
1989. Volume 11. Host and Venue - University of Wollongong, NSW.
1988. Volume 10. Host and Venue - University of Queensland, QLD.
1987. Volume 9. Host and Venue - Deakin University, VIC.
1986. Volume 8. Host and Venue - Australian National University, Canberra, ACT.
1985. Volume 7. Hosts - University of Melbourne and Monash University. Venue - Melbourne, VIC.
1984. Volume 6. Host and Venue - University of Adelaide, SA.
1983. Volume 5. Host and Venue - University of Sydney, NSW.
1982. Volume 4. Host and Venue - University of Western Australia, WA.
1981. Volume 3. Host and Venue - University of Queensland, QLD.
1980. Volume 2. Host and Venue - Australian National University, Canberra, ACT.
1979. Volume 1. Host and Venue - University of Tasmania, TAS.
1978. Volume 0. Host and Venue - University of New South Wales, NSW.

Conference Acronyms

ACDC	Australasian Computing Doctoral Consortium
ACE	Australasian Computer Education Conference
ACSC	Australasian Computer Science Conference
ACSW	Australasian Computer Science Week
ADC	Australasian Database Conference
AISC	Australasian Information Security Conference
APCCM	Asia-Pacific Conference on Conceptual Modelling
AUIC	Australasian User Interface Conference
AusPDC	Australasian Symposium on Parallel and Distributed Computing (replaces AusGrid)
AWC	Australasian Web Conference
CATS	Computing: Australasian Theory Symposium
HIKM	Australasian Workshop on Health Informatics and Knowledge Management

Note that various name changes have occurred, which have been indicated in the Conference Acronyms sections in respective CRPIT volumes.

ACSW and ACSC 2013 Sponsors

We wish to thank the following sponsors for their contribution towards this conference.

CORE - Computing Research and Education,
www.core.edu.au

Australian Computer Society,
www.acs.org.au

University of South Australia,
www.unisa.edu.au/

Contributed Papers

A Study of Performance Variations in the Mozilla Firefox Web Browser

Jan Larres[1] **Alex Potanin**[1] **Yuichi Hirose**[2]

[1] School of Engineering and Computer Science
Email: {larresjan,alex}@ecs.vuw.ac.nz

[2] School of Mathematics, Statistics and Operations Research
Email: hirose@msor.vuw.ac.nz
Victoria University of Wellington, New Zealand

Abstract

In order to evaluate software performance and find regressions, many developers use automated performance tests. However, the test results often contain a certain amount of noise that is not caused by actual performance changes in the programs. They are instead caused by external factors like operating system decisions or unexpected non-determinisms inside the programs. This makes interpreting the test results difficult since results that differ from previous results cannot easily be attributed to either genuine changes or noise. In this paper we present an analysis of a subset of the various factors that are likely to contribute to this noise using the Mozilla Firefox browser as an example. In addition we present a statistical technique for identifying outliers in Mozilla's automatic testing framework. Our results show that a significant amount of noise is caused by memory randomization and other external factors, that there is variance in Firefox internals that does not seem to be correlated with test result variance, and that our suggested statistical forecasting technique can give more reliable detection of genuine performance changes than the one currently in use by Mozilla.

Keywords: performance variance; performance evaluation; automated testing

1 Introduction

Performance is an important aspect of almost every field of computer science, be it development of efficient algorithms, compiler optimizations, or processor speed-ups via ever smaller transistors. This is apparent even in everyday computer usage – no one likes using sluggish programs. But the impact of performance changes can be more far-reaching than that: it can enable novel applications of a program that would not have been possible without significant performance gains.

In the context of browsers this is very visible with the proliferation of so-called "web apps" in recent years. These websites make heavy use of JavaScript to create a user experience similar to local applications, which creates an obvious incentive for browser vendors to optimize their JavaScript execution speed to stay ahead of the competition.

A situation like that poses a problem for developers, though. Speed is not the only important aspect of a browser; features like security, extensibility and support for new web standards are at least as important. But more code can negatively impact the speed of an application: start-up becomes slower due to more data that needs to be loaded, the number of conditional tests increases, and increasingly complex code can make it less than obvious if a simple change might have a serious performance impact due to unforeseen side effects.

Automated tests help with this balance by alerting developers of unintended consequences of their code changes. For example, a new feature might have the unintended consequence of slowing certain operations down, and based on this new information the developers can then decide on how to proceed. However, in order to not create a large number of false positives whose investigation creates more problems than it solves the tests need to be reliable. But even though computers are deterministic at heart, there are several factors that can make higher-level operations non-deterministic enough to have a significant impact on these performance measurements, making the detection of genuine changes very challenging.

1.1 Contributions

This paper tries to determine what the most significant factors are that cause non-determinism and thus variation in the performance measurements, and how they can be reduced as much as possible, with the ultimate goal of being able to distinguish between noise and real changes for new performance test results. Mozilla Firefox is used as a case study since as an Open Source project it can be studied in-depth. This will hopefully significantly improve the value of these measurements and enable developers to concentrate on real regressions instead of wasting time on non-existent ones.

In concrete terms, we present:

- An analysis of factors that are outside of the control, i.e. *external* to the program of interest, and how it impacts the performance variance, with suggestions on how to minimize these factors,
- an analysis of some of the *internal* workings of Firefox in particular and their relationship with performance variance, and
- a statistical technique that would allow automated test analyses to better evaluate whether there has been a genuine change in performance recently, i.e. one that has not been caused by noise.

Table 1: The various performance tests employed by Mozilla

Test name	Test subject	Unit
a11y	Accessibility	Milliseconds
dromaeo_basics	JavaScript	Runs/second
dromaeo_css	JS/CSS manipulation	Runs/second
dromaeo_dom	JS/DOM manipulation	Runs/second
dromaeo_jslib	JS libraries	Runs/second
dromaeo_sunspider	SunSpider benchmark through Dromaeo suite	Runs/second
dromaeo_v8	V8 suite benchmark Dromaeo suite	Runs/second
tdhtml	JS DOM animation	Milliseconds
tgfx	Graphics operations	Milliseconds
tp_dist	Page loading	Milliseconds
tp_dist_shutdown	Shutdown time after page loading	Milliseconds
tsspider	SunSpider benchmark	Milliseconds
tsvg	SVG rendering	Milliseconds
tsvg_opacity	Transparent SVG rendering	Milliseconds
ts	Startup time	Milliseconds
ts_shutdown	Shutdown time	Milliseconds
v8	V8 benchmark	Milliseconds

More details and complete plots for all of our experiments can be found in the accompanying technical report (Larres et al. 2012).

1.2 Outline

The rest of this paper is organized as follows. Section 2 gives an overview of the problem using an example produced with the official Firefox test framework. Section 3 looks at external factors that can influence the performance variance like multitasking and hard drive access. Section 4 looks at what is happening inside of Firefox while a test is running and how these internal factors might have an effect on performance variance. Section 5 presents a statistical technique that improves on the current capability of detecting genuine performance changes that are not caused by noise. Section 6 gives an overview of related work done in this area. Finally, Section 7 summarizes our results and gives some suggestions for future work.

2 Background

2.1 The Talos Test Suite

The *Talos* test suite is a collection of 17 different tests that evaluate the performance of various aspects of Firefox. A list of those tests is given in Table 1. The purpose of this test suite is to evaluate the performance of a specific Firefox *build*. This is done as part of a process of *Continuous Integration* (Fowler 2006), where newly committed code gets immediately compiled and tested to find problems as early as possible.

The focus of this work is on the Talos performance evaluation part of the continuous integration process. We will also mostly focus on variance in unchanging code and the detection of regressions in order to limit the scope to a manageable degree (O'Callahan 2010).

2.2 An Illustrative Example

Figure 1 illustrates some example data from the tp_dist part of the test suite over most of the year 2010. This test loads a number of web pages from the local disk and averages over the rendering times. We can see two distinct change patterns in the graph: two big drops in June and August, and seemingly random changes the rest of the time. Since the second drop causes the rest of the results to stay around that level, it suggests a code optimization that led to an overall better performance. The earlier drop of similar magnitude could be a previous application of the optimization that exposed some bugs and was therefore reverted until the bugs were fixed.

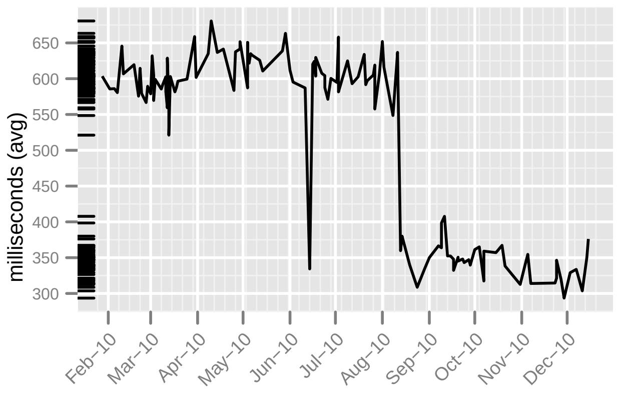


Figure 1: Page load speed tp_dist example sequence with data taken from graphs.mozilla.org

Unfortunately we do not have an explanation for the other changes that is as simple as that. But could we apply the same heuristic that lets us explain the big changes – seeing it "sticking out" of the general trend – and use it in a more statistically sound way to try to explain the other results? To some degree, yes.

The exact details of the best way to do this will be explained in Section 5, but let us first have a very simple look at how we could put a number on the variance of a test suite series. We will do this by running a base line series using a standard setup without any special optimizations.

2.3 Statistics Preliminaries

The Talos suite already employs a few techniques that are meant to mitigate the effect of random variance on the test results. One of the most important is that each test is run 5-20 times, depending on the test, and the results are averaged. A statistical optimization that is already being done here is that the maximum result of these repetitions is discarded before the average is calculated. Since in almost all cases this is the first result, which includes the time of the file being fetched from the hard disk, it serves as a simple case of steady-state analysis where only the results using the cache – which has relatively stable access times – are going to be used.

For our statistical significance analyses we will use the common significance level of 0.05.

2.4 The Base Line Test

2.4.1 Experimental Setup

For this and all the following experiments in this paper we used a Dell Optiplex 780 computer with an Intel Core 2 Duo 3.0 GHz processor and 4 GB of RAM running Ubuntu Linux 10.04 with Kernel 2.6.32. To start with we ran the whole test suite 30 times back-to-back as a series using the same executable in an idle GNOME desktop, with 30 being a compromise between reasonable test run times and possible steady-state detection. The only adjustments

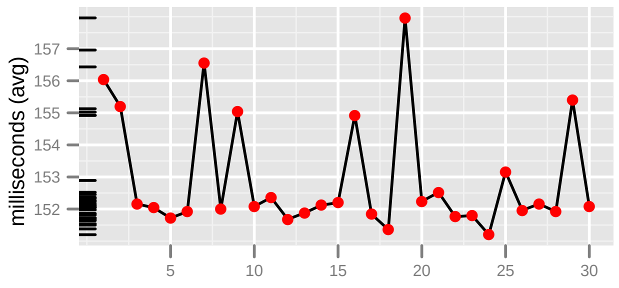


Figure 2: `tp_dist` results of 30 runs

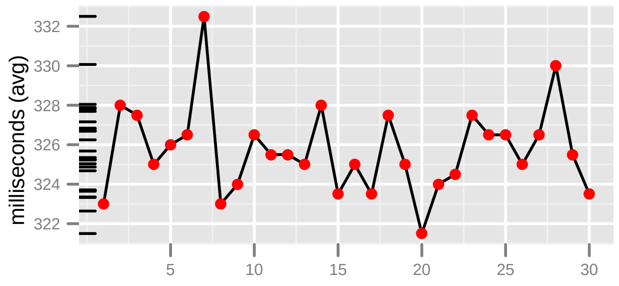


Figure 3: `a11y` results of 30 runs

that we made were two techniques used on the official Talos machines[1], namely replacing the `/dev/random` device with `/dev/urandom` and disabling CPU frequency scaling.

In the following we use the term *run* to refer to a single execution of the whole or part of the Talos test suite and *series* to refer to a sequence of runs, consisting of 30 single runs unless noted otherwise.

2.4.2 Results

Figure 2 shows the results of the `tp_dist` page loading test, and Figure 3 shows the results of the `a11y` accessibility test – both serve as good examples for the complete test suite results. Here we have – as expected – no drastic outliers, but we do still have a non-trivial amount of variance.

Table 2 shows a few properties of the results for the complete test suite. As a typical statistical measure we included the standard deviation and the coefficient of variation (CoV) for easier comparison between different tests. The standard deviation shows us that, indeed, the variation for some of the tests is quite high. The general goal is that we want to be able to detect regressions that are as small as 0.5 % (O'Callahan 2010), so it should be possible to analyse the results in a way so that we can distinguish between genuine changes and noise at this level of precision.

We first look at the maximum difference between all of the values in our series taken as a percentage of the mean, similar to Georges et al. (2007), Mytkowicz et al. (2009) and Alameldeen & Wood (2003). In other words we take the difference between the highest and the lowest value in our series and divide it by the mean. If a new result would increase this value, it would be assumed to not be noise. Looking at the table we can see that almost none of the tests are anywhere near our desired accuracy, so using this method would give us no useful information. If we measure the difference from the mean instead of between the highest and lowest result we can see that the values

[1]`https://wiki.mozilla.org/ReferencePlatforms/Test/FedoraLinux`

Table 2: Results of the base line test

Test name	StdDev	CoV[1]	Max diff (%) Absolute[2]	Max diff (%) To mean[3]
`a11y`	2.23	0.69	3.38	2.08
`dromaeo_basics`	4.41	0.53	2.57	1.62
`dromaeo_css`	11.36	0.30	1.39	0.88
`dromaeo_dom`	1.02	0.41	1.99	1.14
`dromaeo_jslib`	0.53	0.30	1.19	0.60
`dromaeo_sunspider`	5.65	0.54	2.09	1.16
`dromaeo_v8`	2.02	0.86	3.03	1.77
`tdhtml`	0.94	0.33	1.31	0.73
`tgfx`	0.80	5.68	25.60	18.88
`tp_dist`	1.77	1.16	4.42	3.30
`tp_dist_shutdown`	27.09	5.14	16.51	8.72
`ts`	2.27	0.59	2.45	1.66
`ts_shutdown`	7.28	2.00	6.88	3.44
`tsspider`	0.11	1.15	4.04	2.57
`tsvg`	1.43	0.04	0.17	0.10
`tsvg_opacity`	0.62	0.74	3.56	2.02
`v8`	0.11	1.42	4.31	3.59

[1]Coefficient of variation: $\frac{StdDev}{mean}$
[2]Difference between highest and lowest values: $(highest - lowest)/mean * 100$
[3]$max(highest - mean, mean - lowest)/mean * 100$

obviously do look better, but they are still too far away from being actually useful. An additional problem with these techniques is that they have problems with significant genuine changes in the performance like the ones in Figure 1, which are usually much larger than the variance caused by noise.

Section 5 will pursue more sophisticated methods to try to address these concerns. However, even with better statistical methods it will be challenging to reach our goal – the noise is simply too much. Therefore in the next two sections we will first have a look at the physical causes for the noise and try to reduce the noise itself as much as possible before we continue with our statistical analysis.

An important thing to note here is that it is clearly impossible to account for all possible environments that an application may be run in, but that even an artificial environment like ours should still be effective in uncovering the most common issues.

3 External Factors: Hardware, multitasking and other issues

3.1 Overview of External Factors

3.1.1 Multitasking

Multitasking allows several programs to be executed nearly simultaneously, and the kernel tries to schedule them in a way so that the reality of them actually running sequentially (at least on one CPU) is hidden from the user. The consequence of this is that the more programs are running, the less CPU time is available for each one. So the amount of work that can be achieved by any one program in a given amount of real (wall clock) time depends on how many other programs are running. This means that care should be takes as to which programs are active during tests, and also that wall clock time is not very useful for precise measurements. The actual CPU time is of more interest to us. In addition the scheduling may differ from one run to the next, potentially leading to more variance.

3.1.2 Multi-processor systems

In recent years systems with more than one processor, or at least more than one processor core, have become commonplace. This has both good and bad effects on our testing scenario. The upside of it is that processes that use kernel-level threads (as Firefox does) can now be split onto different processors, with in the extreme case only one process or thread running exclusively on one CPU. This prevents interference from other processes as described above. "Spreading out" a process in this way is possible since typical multi-processor desktop systems normally use a shared-memory architecture. This allows threads, which all share the same address space, to run on different processors. The only thing that will not get shared in this case is CPU-local caches – which creates a problem for us if a thread gets moved to a different processor, requiring the data to be fetched from the main memory again. So if the operating systems is trying to balance processes and threads globally and thus moves threads from our Firefox process around this could potentially lead to additional variance.

3.1.3 Address-space layout randomization

Address-space layout randomization (ASLR) (Shacham et al. 2004) is a technique to prevent exploiting buffer overflows by randomizing the address-space layout of a program for each run. This way an attacker cannot know in advance what data structures will lie at the addresses after a specific buffer, making overwriting them with data that facilitates an attack much harder.

Unfortunately, for our purposes this normally very useful technique can do more harm than good. For example, the randomization can lead to data structures being aligned differently in memory during different executions of the same program, introducing variance as observed by Mytkowicz et al. (2009) and Gu et al. (2004).

Additionally, in *Non-Uniform Memory Access* (NUMA) architectures the available memory is divided up and directly attached to the processors, with the possibility of accessing another processor's memory through an interconnect. This decreases the time it takes a processor to access its own memory, but increases the time to the rest of the memory. So depending on where the requested memory region is located the access time can vary. In addition the randomization makes prefetching virtually impossible, increasing page faults and cache misses (Drepper 2007).

3.1.4 Hard disk access

Running Firefox with the Talos test suite involves accessing the hard disk at two important points: when loading the program and the files needed for the tests, and when writing the results to log files. Hard disk access is however both significantly slower than RAM access and much more prone to variance. This is mainly for two reasons: (1) hard disks have to be accessed sequentially, which makes the actual position of data on them much more important than for random-access memory and can lead to significant seek times, and (2) hard drives can be put into a suspended mode that they then have to be woken up from, which can take up to several seconds.

3.1.5 Other factors

Other factors that can play a role are the UNIX environment size and linking order of the program as investigated by Mytkowicz et al. (2009). In our case we worked on the same executables using the same environment and so those effects have not been investigated further.

3.2 Experimental setup

Our experimental setup was designed to mitigate the effect of the issues mentioned in the previous section on the performance variance. The goal was to evaluate how much of the variance observed in the performance tests was actually caused by those external factors as compared to internal ones.

The following list details the way the setup of our test machine was changed for our experiments.

- Every process that was not absolutely needed, including network, was terminated.
- Address-space randomization was disabled in the kernel.
- The Firefox process was exclusively bound to one of our two CPUs, and all other processes to the second one.
- The test suite and the Firefox binary were copied to a RAM disk and run from there. The results and log files were also written to the RAM disk.

Using this setup we ran a test series again and compared the results with our previous results from Section 2.4.2. In our first experiment we tested all of these changes at the same time instead of each individually to see how big the cumulative effect is.

3.3 Results

A comparison of the results of our initial tests and the external optimization approach are shown in Table 3. Overall the results show a clear improvement, most of the performance differences have been significantly reduced. For example, the maximum difference to the mean for the `a11y` test went down from 2.08 % to 0.46 % and for `tsspider` it went down from 2.57 % to 1.34 %.

In order to give a better visual impression of how the results differ Figure 4 shows a violin plot of some of their density functions, normalized to the percentage of their means, with red dots indicating outliers, the white bar the inter-quartile range similar to boxplots and the green dot the median.

Looking at the plots we can see that in the cases of for example `tgfx` and `tp_dist` the modifications got rid of all the extreme outliers. The curious shape of the `v8` plot means that all of the results from the test had the same value, our ideal outcome for all of the tests. Also even though the result table indicates that the max diff metric for `ts` and `tsvg_opacity` increased, the plots show that this is caused by a few extreme outliers and that the rest of the results seem to have gotten better.

3.3.1 The Levene Test

In order to test whether the perceived differences in variance between our setups are actually statistically significant, we made use of the *Levene test for the equality of variances* (Levene 1960, Brown & Forsythe 1974). This test determines whether the null hypothesis of the variances being the same can be rejected or not – similar to the ANOVA test which does the same thing for means. This test is robust against non-normality of the distributions, so even though not all

Table 3: Results after all external optimizations

	StdDev		CoV		Max diff (%)		
Test name	nomod	cumul	nomod	cumul	nomod	cumul	Levene p-value
a11y	2.23	0.54	0.69	0.17	2.08	0.46	< 0.001***
dromaeo_basics	4.41	2.39	0.53	0.29	1.62	1.01	0.028*
dromaeo_css	11.36	7.95	0.30	0.21	0.88	0.46	0.314
dromaeo_dom	1.02	1.00	0.41	0.40	1.14	0.74	0.562
dromaeo_jslib	0.53	0.44	0.30	0.25	0.60	0.79	0.280
dromaeo_sunspider	5.65	3.77	0.54	0.36	1.16	0.74	0.086
dromaeo_v8	2.02	1.20	0.86	0.52	1.77	0.81	0.075
tdhtml	0.94	0.30	0.33	0.10	0.73	0.39	< 0.001***
tgfx	0.80	0.14	5.68	1.37	18.88	2.93	< 0.001***
tp_dist	1.77	0.19	1.16	0.14	3.30	0.35	0.002**
tp_dist_shutdown	27.09	8.59	5.14	1.75	8.72	5.41	< 0.001***
ts	2.27	2.46	0.59	0.74	1.66	3.26	0.282
ts_shutdown	7.28	3.75	2.00	1.19	3.44	2.89	< 0.001***
tsspider	0.11	0.05	1.15	0.64	2.57	1.34	< 0.001***
tsvg	1.43	0.68	0.04	0.02	0.10	0.05	0.006**
tsvg_opacity	0.62	1.11	0.74	1.35	2.02	6.82	0.639
v8	0.11	0.00	1.42	0.00	3.59	0.00	0.008**

nomod: unmodified setup; cumul: cumulative modifications; * $p \leq 0.05$, ** $p < 0.01$, *** $p < 0.001$

of the tests follow a normal distribution the test will still be valid.

Table 3 shows the resulting p-value after applying the Levene test to all of our test results. The results confirm our initial observations: 10 out of 17 tests have a very significant difference, except for most of the dromaeo tests and the ts and tsvg_opacity tests. The dromaeo tests are especially interesting in that most of them are a good way away from a statistically significant difference, and even the one test that does have one is less significant than all the other positive tests. It seems as if the framework used in those tests is less susceptible to external influences than the other, stand-alone tests.

3.4 Isolated Parameter Tests

In order to determine which of our modifications had the most effect on the tests and whether maybe some modifications have a larger impact on their own we also created four setups where only one of our modifications was in use: (1) disabling all unnecessary processes (plain), (2) disabling address-space randomization (norand), (3) exclusive CPU use (exclcpu) and (4) usage of a RAM disk (ramfs).

Table 4 shows the results of comparing the isolated parameters to the unmodified version using the Levene test. We can see that the modification that led to the highest number of significant differences is the deactivation of memory randomization. Especially in the v8 test it was the only modification that had any effect at all – it was solely responsible for the test always resulting in the same value. Equally interesting is that this modification also causes two of the dromaeo tests to become significant that were not in the cumulative case, dromaeo_jslib and dromaeo_sunspider. That suggest that the other modifications seem to "muddle" the effect somehow. Also, in the dromaeo_basics case the disabled memory randomization is the only modification that got rid of all the outliers. Interesting to note is that in the tgfx and tp_dist cases all of the modifications have an influence on the outliers.

3.5 Conclusions

Our modified test setup was a definite improvement on the default state without any modifications. Even though the results did not quite match our goals, they still signified a step in the right direction. Based on that we can safely assume that part of the originally observed variation is caused by the external factors investigated in this section.

Table 4: Levene p-values for isolated modifications, compared to the unmodified setup

Test	plain	norand	exclcpu	ramfs
a11y	0.141	0.831	0.072	0.419
dromaeo_basics	0.617	0.001**	0.199	0.984
dromaeo_css	0.357	0.156	0.926	0.347
dromaeo_dom	0.226	0.112	0.921	0.316
dromaeo_jslib	0.316	0.020*	0.069	0.212
dromaeo_sunspider	0.915	0.028*	0.401	0.743
dromaeo_v8	0.205	0.443	0.995	0.555
tdhtml	0.626	0.983	0.168	0.248
tgfx	0.018*	< 0.001***	0.005**	0.002**
tp_dist	0.006**	0.041*	0.039*	0.038*
tp_dist_shutdown	0.316	0.213	0.031*	0.697
ts	0.086	0.433	0.291	0.296
ts_shutdown	0.080	0.149	0.002**	0.786
tsspider	0.315	< 0.001***	0.004**	0.001**
tsvg	0.893	0.157	0.951	0.679
tsvg_opacity	0.127	< 0.001***	0.262	0.698
v8	0.851	0.008**	0.550	0.857

* $p \leq 0.05$, ** $p < 0.01$, *** $p < 0.001$

Even with the significant improvements from this section the results do not quite match our expectations, unfortunately: only 6 of the 17 tests have a maximum difference of less than 0.5 %. This shows that there are other factors to consider that we do not yet have accounted for.

4 Internal factors: CPU Time, Threads and Events

After dealing with external influences in the last section we will now look at factors that involve the internals of Firefox, specifically, as the title indicates, the time the Firefox process actually runs and the threads and events that are used by it. This involves both investigating how these factors are handled internally and modifying the source code of Firefox and the test suite in an attempt to reduce the variance created by them. Due to space constraints the experiments in this section are only presented in summarised form here. The complete results are available in the technical report (Larres et al. 2012).

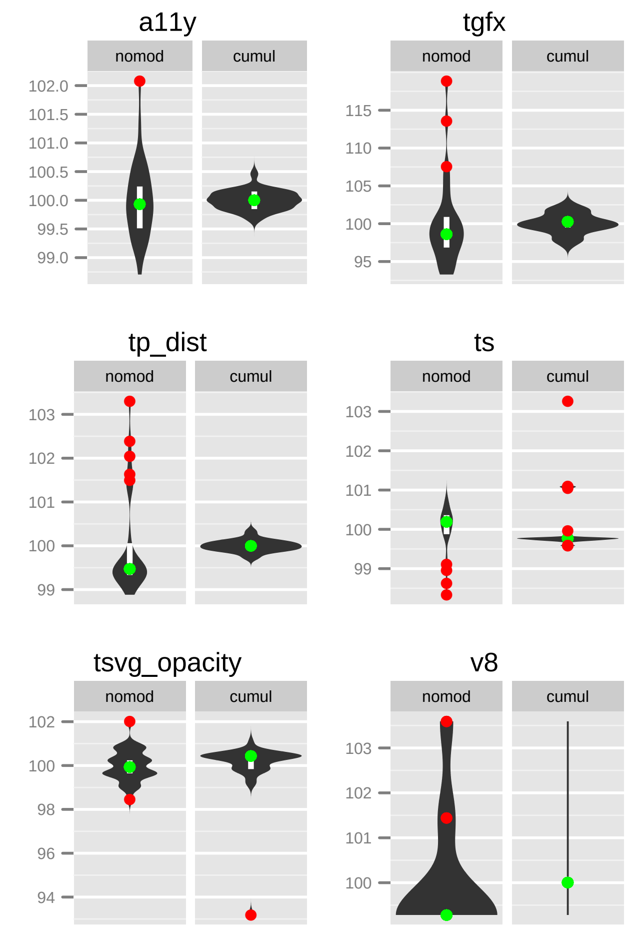


Figure 4: Some of the tests after external optimizations, displayed as the percentage of their mean

4.1 CPU Time

As already mentioned in Section 3.1.1, wall clock time is not necessarily the best way to measure program performance since it will be influenced by other factors of the whole system like concurrently running processes. Since we are running Firefox on an exclusive CPU there is less direct influence by other processes, but context switch time could still matter. We therefore modified Firefox and the test suite to record the CPU time at the start end end of every test run. This was done using the `clock_gettime()` system call for the CLOCK_PROCESS_CPUTIME_ID timer.

Unfortunately only a few tests make direct use of the time that the Talos framework gathers in this manner, namely `tgfx`, `tp_dist`, `tsvg`, and `tsvg_opacity`; most tests, especially the JavaScript tests, do their own timing since they are not interested in the pure page loading time. The results show that only one of them, `tsvg_opacity`, had a statistically significant difference from the results from the external optimizations, and the variance actually seems to have gotten worse (CoV 1.35 to 1.88, $p = 0.005$). This indicates that the method of time recording and the number of context switches are not major factors in contributing to the variance in the tests. Interesting to note is that two other tests that should not have been affected also had significant differences (`dromeao_v8`: CoV 0.52 to 0.7, $p = 0.009$; `tsspider`: CoV 0.64 to 1.02, $p = 0.016$).

Table 5: Correlation analysis for the total number of events

Test name	Coefficient	Pearson p-value
`dromaeo_css`	0.30	0.623
`dromaeo_jslib`	0.36	0.554
`dromaeo_sunspider`	0.76	0.135
`dromaeo_v8`	0.41	0.492
`tgfx`	0.95	0.012*
`tp_dist`	0.97	0.033*
`tsvg_opacity`	−0.76	0.236

* $p \leq 0.05$, ** $p < 0.01$, *** $p < 0.001$

4.2 Thread pools

Firefox uses two different mechanisms for handling work like rendering web pages and UI interaction: threads and events. The majority of work is done using events, but threads are used for a few cases where asynchronous operations like I/O and database transactions are needed, for example for bookmark and history handling. In addition Firefox makes use of a thread pool for one-off asynchronous events. Since this thread pool requires creating and destroying threads on a regular basis, changes in the timings of when a new thread is needed could lead to measurable variance caused by these thread interactions.

We investigated this hypothesis by modifying the thread pool code to only ever create one thread that then stays alive for the entirety of the program lifetime, keeping the pool from creating and destroying threads arbitrarily. Unfortunately the results mirror the ones from our first experiment: only two of the tests had statistically significant differences, and in both cases the variance was worse than without our modifications (`dromaeo_dom`: CoV 0.33 to 0.5, $p = 0.002$; `tgfx`: CoV 1.28 to 1.69, $p = 0.026$). So again the thread pool does not seem to be responsible for the variance that we are seeing.

4.3 Event Variance

As mentioned above, events are the main mechanism by which work is done in Firefox. So for our third experiment we wanted to see whether the events used to execute a certain task, like running a test of the test suite, was always done using the exact same events and in the exact same order of dispatch.

For this we again modified Firefox and the test suite to print out special messages at the points where events get dispatched during the tests, and ran a test series. Due to the size of the generated log files and the time it took to run our analysis script afterwards this series consisted of only five distinct runs.

Using the information from our log analyses we can indeed see that there is variation in the number of events being used during the tests. What is interesting is that there are some events that occur several times in some of the runs but not at all in others, but the overall sum of the events differs far less, proportionally speaking. Since the events are identified by their complete backtrace instead of just their class we suspect that this is because those events get dispatched on a slightly different path through the program even though they belong to the same class.

In order to establish whether the event variance is actually correlated to the test result variance we used the Pearson product-moment correlation coefficient (Rodgers & Nicewander 1988), with the null hypothesis being that there is no correlation between the variables.

Table 6: Correlation analysis for the order of events

Test name	Coefficient	Pearson p-value
`dromaeo_sunspider`	0.58	0.079
`tp_dist`	0.98	< 0.001***
`tsvg`	0.44	0.386
`tsvg_opacity`	0.71	0.113

* $p \leq 0.05$, ** $p < 0.01$, *** $p < 0.001$

The tests that had at least moderate correlation ($abs(\text{coefficient}) \geq 0.3$) are presented in tables 5 and 6. Only two respectively one of them are actually statistically significant, though, which may be a result of our small sample size. But it does demonstrate that it could be worthwhile to investigate this direction further. One finding that should be studied more closely is that the only test that showed significant correlation in both cases is also the one with by far the longest running time, suggesting that the correlation may only become significant after the test has been running for a while, overshadowing other influences beyond that point.

Interesting to note is that most of the events that appear out of order depend on external or at least asynchronous factors, for example ones that interact with database transaction threads or that make use of hardware timers.

5 Forecasting

After trying to actually reduce the variance as much as possible, we will now look at statistical techniques that aim to separate the remaining noise from genuine performance changes. Since we need test results that contain both of these in order to do that, in this Section we will use data taken from the official Mozilla test servers instead of generating our own. Note that this means that all the results used in this section will be from *different* builds, in contrast to our previous experiments.

5.1 t-tests: The current Talos method

There are essentially three cases that a new value in our results could fall into, and the goal is for us to be able to distinguish between them. The first case is that there are no performance-relevant code changes and the noise is so small that it can easily be classified as a non-significant difference from the previous results. The second one is that there are still no relevant code changes, but this time the noise is much larger so that it looks like there may actually be relevant changes. The last one is that there are relevant code changes and the difference in value we see is therefore one that will stay as long as the new code is in place.

This suggests one potential solution to our problem: if we check more than one new value and determine if – on average – they differ from the previous results in a significant way, we know that there must have been a code change that introduced a long-lasting change in performance. Unfortunately this method has a problem of its own: we cannot immediately determine whether a single new value is significantly different, we have to wait for a few more in order to compute the average.

This is essentially what the method that is currently employed by Mozilla does. In more detail, there are two parts to it:

1. Compute the means of the 30 results before the current one (the *back-window*) and of the 5 runs starting from it (the *fore-window*), that is create two *moving averages.*

2. Use a t-test to determine whether the difference between the means is statistically significant.

The size of these windows again has to be a trade-off: the back-window should be relatively immune to short-term noise but also not be distorted by large changes in the past, and the fore-window should be small enough to allow detecting changes quickly without producing too many false positives due to one or two noisy results.

An important thing to note with regard to the fore window is that it *starts* at the value we are currently investigating, not ends. This is because we are interested in the *first* value where a regression happens. If we interpret the performance change as a "step" like in a step-wise function then starting from the first value after the step means that all of the values that are taken into account for the window will share the same change and thus should ideally lead to a mean that reflects that, pointing back at the "step" that caused it.

In order to determine whether there is a significant difference between the two window sample means we need a statistical test, and Mozilla chose the so-called *Welch's t-test* which works for independent samples with unequal variances:

$$t = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

where $\overline{X}_i$, s_i^2 and N_i are the i^{th} sample mean, sample variance and sample size, respectively.

This test statistic t can then be used to compute the significance level of the difference in means as it moves away from zero the more significant the difference is. The default t threshold that is considered to be significant in the Talos analysis is 9. This seems to be another heuristic based on experience, but it can hardly be justified statistically – in order to properly calculate the significance level another value is needed: the *degree of freedom*. Once that is known the significance level can be easily looked up in standard t-test significance tables[2]. However, this degree of freedom has to be computed from the actual data, it cannot be known in advance, and it also would be different for different tests. Using a single threshold for all of the tests is therefore not very reliable.

5.2 Forecasting with Exponential Smoothing

As already mentioned in the previous section, the current method has a few problems. For one thing, the window sizes used are rather arbitrary – they seem to be reasonable, but there is no real statistical justification for them, and the fact that all the values in the window are treated equally presents problems in cases where there have been recent genuine changes. Also, due to the need for the fore window a regression can usually not be found immediately, only after a few more results have come in. Apart from this unfortunate delay this can also lead to changes that go unnoticed because they only exist for a short time, for example because a subsequent change had the opposite effect on performance and the mean would therefore

[2]See for example `http://www.statsoft.com/textbook/distribution-tables/#t`.

hardly be affected. So instead of a potential performance gain the performance will then stay the same since the regression will not get detected.

We therefore need a more statistically valid way that can ideally report outliers immediately and that does not depend on guesses for the best number of previous values to consider.

A common solution to the problem of equal weights in the window average is to introduce weighting, that is a *weighted average*. In the case of our back window we would give the highest weights to the most recent results and gradually less to earlier ones. This would also eliminate the need for a specific window size, since as the weights will be negligible a certain distance away from the current value we can just include *all* (available) previous values in our computation. The only issue in this case is the way in which we assign concrete weights to the previous results.

Exponential smoothing is a popular statistical technique that employs this idea by assigning the weights in an exponentially decreasing fashion, modulated by a smoothing factor, and is therefore also called *exponentially weighted moving average*. The simplest and most common form of this was first suggested by Holt (1957) and is described by the following equations:

$$\begin{aligned} s_1 &= x_0 \\ s_t &= \alpha x_{t-1} + (1-\alpha)s_{t-1} \\ &= s_{t-1} + \alpha(x_{t-1} - s_{t-1}), t > 1 \end{aligned}$$

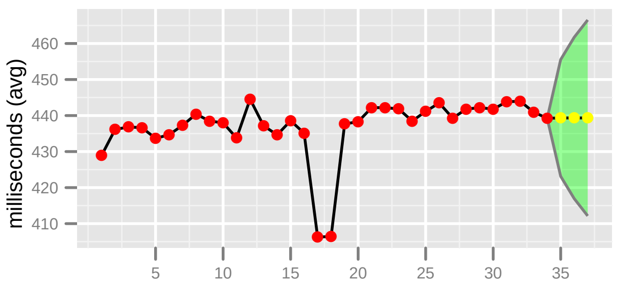
Here s_t is the smoothed statistic and α with $0 < \alpha < 1$ is the smoothing factor mentioned above. Note that the higher the smoothing factor, the *less* smoothing is applied – in the case of $\alpha = 1$ the resulting function would be identical to the original one, and in the case of $\alpha = 0$ it would be a constant with the value of the first result.

The obvious question here is: what is the optimal value for α? That depends on the concrete values of our time series. Manually determining α is infeasible in our case, though, so we would need a way to do it automatically. Luckily this is possible: common implementations of exponential smoothing can use a method that tries to minimize the squared one-step prediction error in order to determine the best value for α in each case[3].

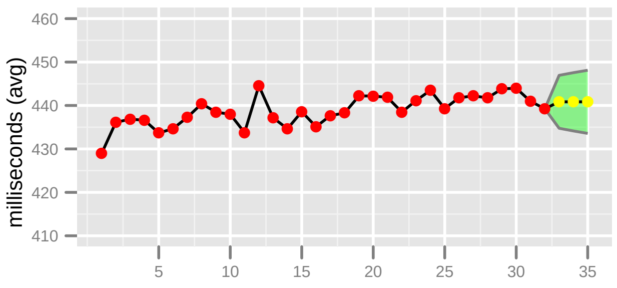
The property that is most important to us about this technique is that it allows us to *forecast* future values based on the current ones. This relieves us of the need to wait for a few new values before we can compute the proper moving average for our fore window, and instead we can operate on a new value immediately. Similarly we do not have to wait until we have enough data for our back window before we can start our analysis. In theory we can start using it with only one value, although in practice we would still need a few values for our analysis to "settle" before the forecasts become reliable.

Normally the exponential smoothing forecast will produce a concrete new value, which is useful for the field of economics where it is most commonly applied. In our case, however, we want to instead know whether a new value that we already have can be considered an outlier. For this we need a modification that will produce *confidence intervals*. Yar & Chatfield (1990) developed a technique for that using the assumption that the underlying statistical

[3]see for example `http://stat.ethz.ch/R-manual/R-patched/library/stats/html/HoltWinters.html`



(a) Jump included



(b) Jump removed

Figure 5: Prediction intervals for three values

model of exponential smoothing is the ARIMA (autoregressive integrated moving average) model, calling the intervals *prediction intervals*. The details of the method are not really relevant here and are also rather complex, so we will refer interested readers to the actual paper instead of repeating them here. We used this modification as it was implemented in the `HoltWinters` package for R (R Core Team 2012).

Figure 5a shows an example from the `tp_dist` test with official test server data and the 95 % prediction interval for the next three values. We used three here to make the interval easier to identify, but in practice only one would be needed.

The figure also demonstrates what influence big changes in the past have on the prediction intervals. The big jump in performance in the middle is still reflected in the intervals at the end, although the results themselves would by now clearly lie outside of them if they were to reoccur. Figure 5b shows the same data except that the two outliers have been removed, and we can immediately see that the prediction intervals are now much more narrow – for example the first value would now lie outside of them, which was not the case in the previous figure. Therefore in the case of such apparently genuine changes that have been reverted it might still make sense to remove the values from the ones that are used for future predictions to avoid intervals that are unnecessarily wide.

Note that there are a few extensions to this simple exponential smoothing technique that have been developed in order to deal with data that exhibits trends, but our data does not contain any trends and therefore we did not make use of any of these extensions.

5.3 Comparison of the Methods

We now want to compare our two methods on an example to give an impression of how they differ in their ability to distinguish between noise and genuine changes. For this we used a long stretch of official test data for the `tp_dist` test and ran both methods on it, marking the points where they reported a significant change.

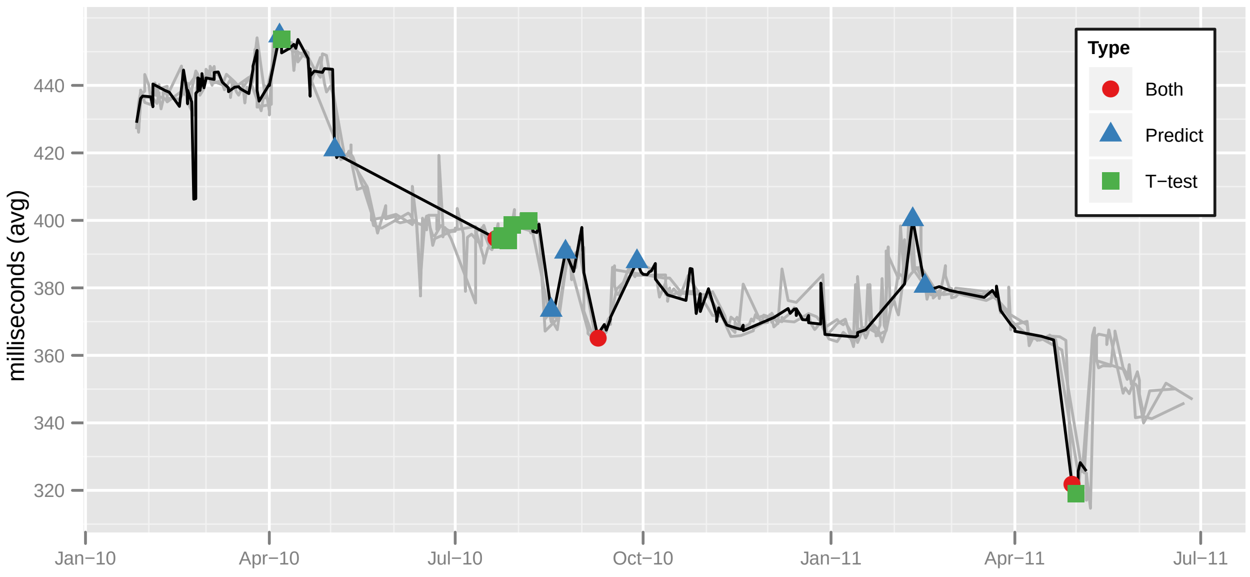


Figure 6: Comparison of the two analysis methods

Figure 6 shows the result of this comparison. The test results from three other machines are also depicted greyed out in the background to easier determine which changes are genuine and which are noise, since the genuine changes will show up in all of the machines.

Two things can be learned from the graph: first, and most importantly, our prediction interval method detects more of the genuine changes than the current *t*-test method. For example, the big jumps in August 2010 and February 2011 go undetected by the current method since they are followed by equally big jumps back soon after. This is a result of the need for more than one value in the respective analysis, obscuring single extreme values in the process. On the other hand, all of the changes that are detected by the old method are also detected by our suggested method, thus demonstrating that previously detectable changes would not get lost with it.

The second difference can be seen during July/ August 2010: the current method can sometimes report the same change multiple times for subsequent values, so additional care has to be taken to not raise more alarms than necessary.

This example demonstrates that our proposed statistical analysis offers various benefits over the one that is currently employed. Not only does it give better results, it also needs only the newest value in order to run its analysis. In addition it is also straightforward to implement, several implementations already exist in popular software like R[3] and Python[4].

One disadvantage of our method should be mentioned, however. If there is a series of small regressions, each too small to be detected as an outlier, then the performance could slowly degrade without any warnings being given. Depending on the exact circumstances this degradation might be able to be detected by the old method, but it would probably be better to develop a different method that is specifically tuned for this case and use this method in addition to ours.

6 Related Work

Mytkowicz et al. (2009) investigated the effects of UNIX environment size and the program link order on performance measurements. The found that those factors can have an effect of up to 8 % and 4 %, respectively, on benchmark results and attributed the variance to memory layout changes. As a partial solution they proposed randomizing the setup. Gu et al. (2004) came to a similar conclusion of memory layout changes through the introduction of new code, but found that this variance was not well correlated with the benchmark variance.

Multi-threading variability was investigated by Alameldeen & Wood (2003), including the possibility of executing different code paths due to OS scheduling differences, which they called *space variability*. Georges et al. (2007) demonstrated that performance measurements in published papers often lack a rigorous statistical background and presented some standard techniques that would lead to more valid conclusions.

Kalibera et al. (2005) investigated the dependency of benchmarks on the initial, random state of the system, finding that the between-runs variance was much higher in their experiments than the within-runs variance. They proposed averaging over several benchmark runs to counter this as much as possible, which is similar to what our experiments did.

Tsafrir et al. (2007) demonstrated that influences outside of the control of a benchmark can lead to disproportionally large variance in the results, and suggested “shaking”/fuzzing the input by carefully adding noise so as to make averages more reliable.

7 Conclusions and future work

This paper had three main goals: (1) Identifying the cause(s) of variance in performance tests on the example of Mozilla Firefox, (2) trying to eliminate them as much as possible, and (3) investigating a statistical technique that would allow for better distinction between real performance changes and noise.

Section 3 demonstrated that all of the external factors that we investigated had a certain degree of influence on the variance, with memory randomization being the most influential one. This is consistent with much of the work mentioned in Section 6 that identified memory layout as having a significant impact on performance measurements. We also proposed some

[4] http://adorio-research.org/wordpress/?p=1230

strategies to minimize this variance without the additional resources needed for the averaging solutions that others have suggested.

The studying of the internal factors in Section 4 proved to be less useful than we had hoped for, but it provided us with evidence that they did *not* have a significant amount of influence on the result variance. This suggests that whatever variance remains more likely has to do with the external environment instead of the internal workings of the applications to be measured, allowing better focused future studies.

Finally, in Section 5 we presented a statistical technique for assessing whether a new result in a test series falls outside of the current trend and is therefore most likely not noise. This technique was shown to have various benefits over the currently used one, most importantly it could report some changes that the one that is currently being used by Mozilla missed. Additional advantages include being able to run the analysis on new values immediately instead of having to wait for a certain number of values that are needed for a moving average, and similarly the analysis can start when only a few values are available for a machine unlike the 30 values that are required for the current moving average.

In summary we managed to achieve a certain degree of success for all three of our goals. We identified various external influences and offered solutions to mitigate them, and suggested a statistical technique that improves the quality of change detection. Unfortunately we did not conclusively find a connection between the inner workings of Firefox and the measured variance, but we did find a certain amount of internal variance. Investigating this discrepancy could be a promising topic for future work.

Another worthwhile direction would be to apply our research to other applications, especially other browsers like Google Chrome. This was outside the scope of this paper, not the least because those browsers use entirely different – and not in all cases even publicly accessible – performance test suites. The general principle should be the same, though, so it would be interesting to see whether there are any differences between the amount of and the causes of variance. At least our statistical technique is not tied to any specific application and should work for anything that can be represented as a time series, regardless of how the data was produced.

Acknowledgements

The authors would like to thank Mozilla for funding this research and especially Robert O'Callahan for his invaluable insights whenever a question about Firefox internals came up.

References

Alameldeen, A. R. & Wood, D. A. (2003), Variability in architectural simulations of multi-threaded workloads, *in* 'High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on', pp. 7–18.

Brown, M. B. & Forsythe, A. B. (1974), 'Robust tests for the equality of variances', *Journal of the American Statistical Association* **69**(346), 364–367.

Drepper, U. (2007), 'What every programmer should know about memory', `http://people.redhat.com/drepper/cpumemory.pdf` [22 April 2012].

Fowler, M. (2006), 'Continuous integration', `http://www.martinfowler.com/articles/continuousIntegration.html` [22 April 2012].

Georges, A., Buytaert, D. & Eeckhout, L. (2007), Statistically rigorous java performance evaluation, *in* 'Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07', Montreal, Quebec, Canada, p. 57.

Gu, D., Verbrugge, C. & Gagnon, E. (2004), 'Code layout as a source of noise in JVM performance', *In Component and Middleware Performance Workshop, OOPSLA* .

Holt, C. C. (1957), 'Forecasting seasonals and trends by exponentially weighted moving averages', *International Journal of Forecasting* **20**(1), 5–10.

Kalibera, T., Bulej, L. & Tuma, P. (2005), 'Benchmark precision and random initial state', *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems* pp. 853—862.

Larres, J., Potanin, A. & Hirose, Y. (2012), A study of performance variations in the mozilla firefox web browser, Technical Report 12-14, Victoria University of Wellington.

Levene, H. (1960), Robust tests for equality of variances, *in* I. Olkin, S. G. Ghurye, W. Hoeffding, W. G. Madow & H. B. Mann, eds, 'Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling', Stanford University Press, pp. 278–292.

Mytkowicz, T., Diwan, A., Hauswirth, M. & Sweeney, P. F. (2009), Producing wrong data without doing anything obviously wrong!, *in* 'Proceeding of the 14th international conference on Architectural support for programming languages and operating systems', ACM, Washington, DC, USA, pp. 265–276.

O'Callahan, R. (2010), 'Private communication'.

R Core Team (2012), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Rodgers, J. L. & Nicewander, W. A. (1988), 'Thirteen ways to look at the correlation coefficient', *The American Statistician* **42**(1), 59–66.

Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N. & Boneh, D. (2004), On the effectiveness of address-space randomization, *in* 'Proceedings of the 11th ACM conference on Computer and communications security', CCS '04, ACM, Washington DC, USA, p. 298–307.

Tsafrir, D., Ouaknine, K. & Feitelson, D. G. (2007), Reducing performance evaluation sensitivity and variability by input shaking, *in* 'Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS '07. 15th International Symposium on', pp. 231–237.

Yar, M. & Chatfield, C. (1990), 'Prediction intervals for the Holt-Winters forecasting procedure', *International Journal of Forecasting* **6**(1), 127–137.

replay: Visualising the Structure and Behaviour of Interconnected Systems

Alex Murray **Duncan Grove**

Defence Science Technology Organisation
PO Box 1500, Edinburgh, South Australia 5111
Email: `alex.murray@dsto.defence.gov.au`

Abstract

Visualisation is often used to help understand complex systems and in particular scale-free networks which are present in many systems, from object-oriented software, to real-world and on-line social networks. While a number of tools already exist to visualise these systems, most focus on presenting the network as a whole and neglect to include information on the possibly concurrent behaviour of individual nodes. In this paper we present *replay* which aims to meet these demands, by visualising both the structure and evolution of the network through time, as well as the behaviour of individual nodes and the communications between nodes. We describe the unique and novel aspects of *replay*, including its three different but related visualisations of the underlying system, as well as its plug-in architecture, which allows *replay* to be extended and applied to visualise different networked systems. We also demonstrate the utility and flexibility of *replay* with a number of real-world visualisation examples, as well as present possible directions for future work.

Keywords: Graph and network visualisation, concurrency visualisation, interaction visualisation, interconnected systems, concurrent systems.

1 Introduction

Visualisation is increasingly being used to aid understanding of complex systems. In particular, scale-free networks [5] have recently become a focus for visualisation [26, 16], as a means to understand their underlying structures and hierarchies, as well as their evolution through time [19]. Many networked systems have been found to exhibit scale-free properties, ranging from social networks [5] (both real-world and on-line) to object-oriented software [30, 8, 24].

By their nature, scale-free networks are quite complex, comprising many nodes with numerous edges, and hence visualisations have focused on ways of simplifying the overall visualisation while still retaining the salient features of the network [16, 19]. These methods concentrate on visualising the network as a whole, and so while these tools have generally been successful in helping to understand the overall network structure, they provide little capacity for insight into the detailed behaviour of individual nodes – which we define as the node's activity and any messages it exchanges with other nodes.

In many cases, understanding the behaviour of individual nodes is crucial to properly understanding the overall behaviour of the network, since the evolution of the network through time is critically affected by the actions of its nodes. For example, the need for visualisations that show both the network's overal structure as well as the concurrent behaviour of its individual nodes has been identified as an educational tool to aid in the understanding of object-oriented software, especially the interactions between concurrent objects and how this influences the object graph as a whole [9].

We believe that to truly understand complex, networked, concurrent systems, visualisation tools must be capable of effectively exploring these systems at both macroscopic and microscopic levels of detail, while sweeping arbitrarily backwards and forwards through time.

We have therefore developed a visualisation tool called *replay* to meet these requirements. Section 2 describes the system while Section 3 presents some case-studies showing *replay* in action. We then describe related work in Section 4, future directions for our research in Section 5 and conclude in Section 6.

2 An overview of *replay*

replay was designed with a number of features for visualising complex, concurrent networked systems: a simple event based data model, three different but related visualisations of the underlying event model which are always synchronised, the ability to filter information, and a plug-in based extension system. Each of these features will be described in the following sections.

2.1 The *replay* event model

The primary elements represented within *replay* are *nodes*, *edges*, *activities* and *messages*, where nodes can be executing activities and are connected via edges to form the graph, and messages are sent between nodes along edges within the graph. *replay* employs a simple event-based data model which allows the behaviour and structure of diverse concurrent networked systems to be visualised. The plug-in interface (described in Section 2.7) provides programmatic access to drive the generation of events, allowing arbitrary systems to interface with *replay* at runtime using a diverse range of communication mechanisms. Each event specifies the time at which it occurred, as well as identifying the elements concerned. The four basic elements (*nodes*, *edges*, *activities* and *messages*) are all uniquely named within separate name-spaces and can have arbitrary properties. The following describes the standard events associated with each element.

Node create / set properties Specifies the unique identifier for the node and a list of associated properties for the node in question (such as a label or colour).

Node delete Deletes the node with the given identifier.

Activity start Specifies the start of a uniquely named activity upon a particular node with a list of associated properties for the activity (such as colour or level).

Activity end Specifies the end of the activity with the given identifier.

Edge create Specifies the unique identifier for the edge, the identifiers for the head and tail nodes of the edge, whether the edge is unidirectional or bidirectional and a list of associated properties for the edge in question (such as a label, colour or weight etc).

Edge set properties Specifies the unique identifier of an edge and a list of associated properties to set for the edge at the given time of the event.

Edge delete Deletes the edge with the given identifier.

Message send Message send events specify an identifier for the message, the identifier for the sending node, a potentially associated edge via which the message travels and a list of properties for the message (such as a human readable description to display in the various visualisations). To model causality of message events, these events also specify a parent message which caused this message send to occur.

Message receive Specifies the unique identifier of the message and the node which is receiving the message.

Since all events specify a time-stamp, *replay* is able to reconstruct the sequence of events for the system through time, and allows the ability to step through the sequence both forwards and backwards through time. From the sequence of events, *replay* constructs three different but related views of the system. Figure 1 shows a screen capture of the main *replay* window displaying these three views:

Timeline view This is positioned at the top of the window and shows the state of nodes and their interactions through time.

Causal message tree view This view is placed at the left of the window and is designed to show the causal relationship between messages sent / received between nodes.

Network graph view This is situated on the right hand side of the main window, and is designed to show the graph of nodes within the system and how they are interconnected, along with their individual states, at a given point in time.

2.2 Timeline view

The timeline view presents a two dimensional view of the behaviour of nodes through time. Nodes are listed along the vertical axis, while time is plotted along the horisontal axis. A number of visual attributes are used to show the different states of nodes through time:

Node lifetime A thin line drawn in the node's base colour is drawn from the time of the node create event to the time of the corresponding node delete event.

Node activity The timeline view represents activity in a similar way to the network graph, using the activity level to determine the intensity of the activity colour. A thick coloured line is drawn in the current activity colour / level, and runs from the time of each activity change event to the next. An activity level of zero (the idle state) is indicated by the absence of this line.

Message flow Message send / receive event pairs are indicated by arrows drawn from the node which sent the message to the node which received the message.

Current time The timeline clearly indicates the current point in time using a thin line, with the region in the past shaded behind it.

This view is designed to show the concurrency and message passing characteristics of the system across time, and is similar to existing visualisations for parallel message passing systems [18, 29, 14]. By pairing together events, the timeline view is able to clearly show the duration of each interval, such that communication patterns, message passing latencies, and active / idle times are clearly visible.

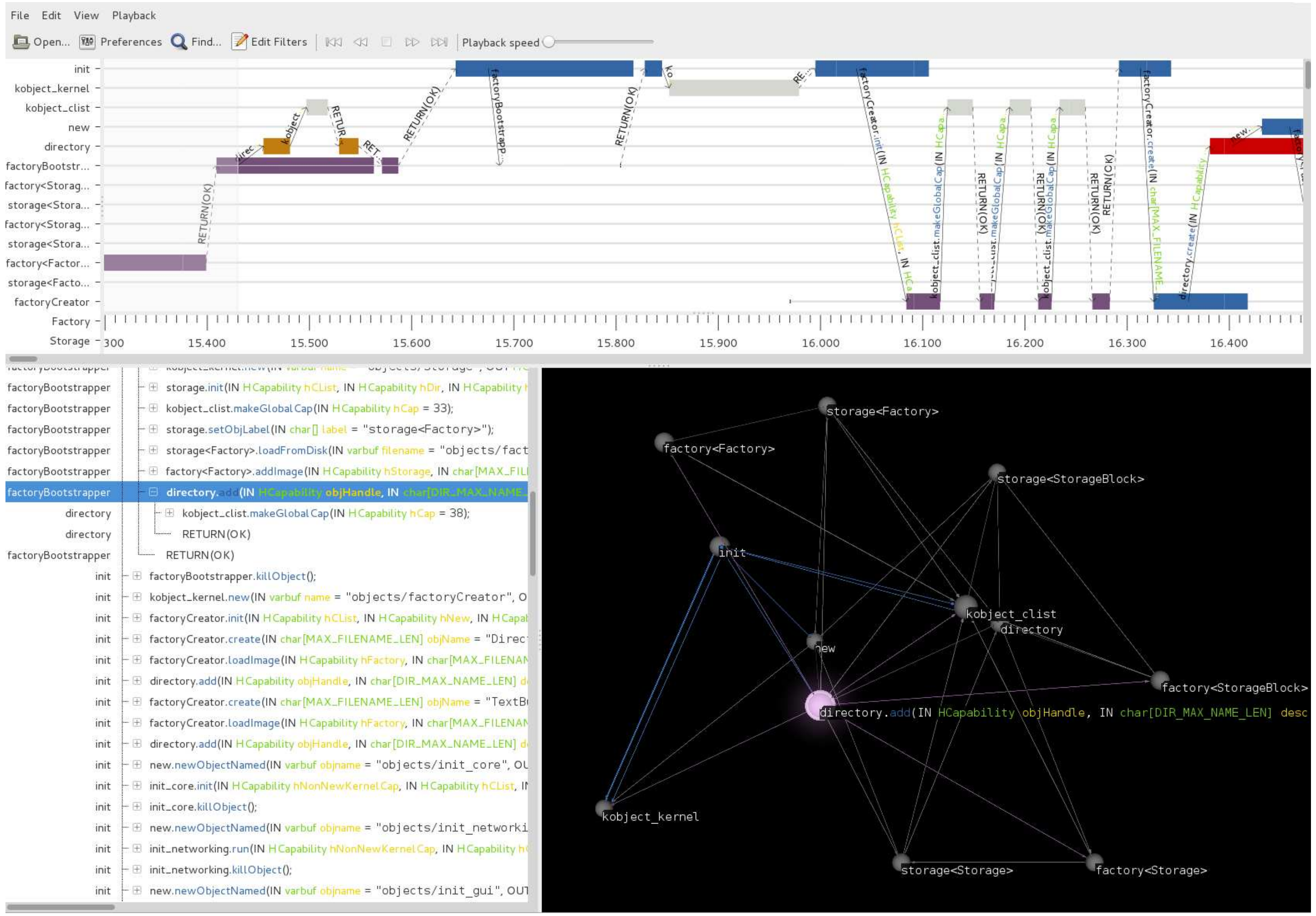
The timeline view allows the user to zoom in and out, providing an infinite zoom resolution to allow the exact timing of events to be clearly represented and determined.

2.3 Causal message tree view

While *replay* allows events to be stepped through sequentially, the representations provided by the other two views give limited insight into the causal relationship of messages within the system. To address this, *replay* includes a third view, the causal message tree. Message send / receive events specify an identifier for the current message, as well as a potential parent message identifier which refers to the message event (if any) which caused the current one. This allows the message tree to be easily specified and constructed. Message send and receive events for the same message are aggregated into a single entry within the tree, as the causality of these events is directly linked (the receipt of a message is always the result of the corresponding send).

This view is situated at the lower left of the window, and lists the node which sent the message on the left, along with the message label on the right. Nodes are coloured with their corresponding base / activity colour, depending on their activity level at the time of the message send event.

This view is similar to the message-order view of Causeway [27], a message oriented postmortem debugger, and provides a visual representation of the causality for the current message event. This view is particularly suited to analysing interactions between



```
factoryBootstrapper   ├ ⊞ storage.init(IN HCapability hCList, IN HCapability hDir, IN HCapability
factoryBootstrapper   ├ ⊞ kobject_clist.makeGlobalCap(IN HCapability hCap = 33);
factoryBootstrapper   ├ ⊞ storage.setObjLabel(IN char[] label = "storage<Factory>");
factoryBootstrapper   ├ ⊞ storage<Factory>.loadFromDisk(IN varbuf filename = "objects/fact
factoryBootstrapper   ├ ⊞ factory<Factory>.addImage(IN HCapability hStorage, IN char[MAX_FIL
factoryBootstrapper   ├ ⊟ directory.add(IN HCapability objHandle, IN char[DIR_MAX_NAME_
          directory   │  ├ ⊞ kobject_clist.makeGlobalCap(IN HCapability hCap = 38);
          directory   │  └ RETURN(OK)
factoryBootstrapper   └ RETURN(OK)
               init ├ ⊞ factoryBootstrapper.killObject();
               init ├ ⊞ kobject_kernel.new(IN varbuf name = "objects/factoryCreator", O
               init ├ ⊞ factoryCreator.init(IN HCapability hCList, IN HCapability hNew, IN HCapal
               init ├ ⊞ factoryCreator.create(IN char[MAX_FILENAME_LEN] objName = "Direc
               init ├ ⊞ factoryCreator.loadImage(IN HCapability hFactory, IN char[MAX_FILENAM
               init ├ ⊞ directory.add(IN HCapability objHandle, IN char[DIR_MAX_NAME_LEN] d
               init ├ ⊞ factoryCreator.create(IN char[MAX_FILENAME_LEN] objName = "TextB
               init ├ ⊞ factoryCreator.loadImage(IN HCapability hFactory, IN char[MAX_FILENAM
               init ├ ⊞ directory.add(IN HCapability objHandle, IN char[DIR_MAX_NAME_LEN] d
               init ├ ⊞ new.newObjectNamed(IN varbuf objname = "objects/init_core", OU
               init ├ ⊞ init_core.init(IN HCapability hNonNewKernelCap, IN HCapability hCList, IN
               init ├ ⊞ init_core.killObject();
               init ├ ⊞ new.newObjectNamed(IN varbuf objname = "objects/init_networki
               init ├ ⊞ init_networking.run(IN HCapability hNonNewKernelCap, IN HCapability h
               init ├ ⊞ init_networking.killObject();
               init ├ ⊞ new.newObjectNamed(IN varbuf objname = "objects/init_gui", OUT
```

Figure 1: Main *replay* window showing the three unified views

nodes, such as in software development (i.e. representing the control flow via method calls between objects in an object-oriented software system).

2.4 Network graph view

The network graph view is situated to the right of the main window, and displays a three dimensional representation of the overall graph of the system. As the sequence of events is stepped through in time, the graph is constructed using the node create / delete and edge add / remove events. The primary purpose of this view is to show the connectivity of nodes within the system, as well as their current state, and finally to annotate the view with messages as they pass between nodes. As a result, a number of specific features have been incorporated into this view.

Like many existing visualisations which focus on the connectivity within a graph [16, 19] a force-directed model is used to layout the graph. All nodes repel each other with an inverse gravitational (Coulombic repulsion) force, while neighbors attract one another using a spring-modeled force. Unlike the two visualisation models cited previously, nodes in *replay* have a 'mass' which is proportional to the total number of connections they have. Nodes are then drawn with a size proportional to this mass (assuming a constant density), and the inverse gravitational repulsive force calculations take this value of mass into account. By adding this property, nodes which are highly connected (and hence, for example, have greater 'authority' as defined by [17]) are significantly larger and are placed further away from less connected nodes. This creates a visual representation where the highly connected nodes are easily discerned due to their placement and size within the overall graph.

Nodes are coloured using the 'color' property value, and activities are drawn as a glow around the node in the designated colour and at the designated intensity (using the 'color' and 'level' properties of the activity respectively). Multiple concurrent activities on a node result in blending of their respective colours at their respective intensities.

The graph is also annotated with the labels of message send and receive events as these events occur, along with the name for the corresponding node. By overlaying these labels alongside the node with which they are associated, the flow of messages within the system is able to be represented along with the state of the system as these events occur.

This view is able to be controlled by the user, providing the ability to center the view on a particular node of interest, zoom in and out, or arbitrarily rotate the viewpoint around the current center. By default node and edge properties are hidden, but are exposed when the user places the mouse over the node or edge of interest. This allows the graph to be easily understood by showing only the vital information, but provides an effective way to allow the user to reveal relevant information as required. Finally, the user can also interact directly with the graph and move the nodes within it to determine how this affects the overall graph layout.

2.5 Synchronisation of views and interaction

While each of the three views provides its own unique representation of underlying the system, we believe the real advantage of *replay* is the combination of all three views. As a result, all three views use similar representation (such as node colour) and remain synchronised at all times, to ensure a consistent representation of the event sequence, and hence the underlying system itself. This is an important feature, since it helps to highlight relationships between the views and allows the different information presented within the views to complement one another [28]. Also, by using consistent representations within all three views, *replay* reduces the cognitive load on the user to understand the underlying data. As a result, this frees the user, allowing them to process large amounts of complex data quite easily, due to the natural cognitive abilities of the human visual system [7].

replay also employs user interaction within all views to allow the user to jump to certain events, and to manipulate the displays of the views. For example, selecting a message within the message tree causes both the network graph and timeline views to jump to that event in the event sequence, and similarly, events can also be selected in the timeline view.

Finally, both the message tree and graph view allow the user to search for a message or node by name respectively to easily locate items of interest.

2.6 Filtering

In many large systems the number of nodes, and their interconnections and communications, can produce quite complex visualisations where the finer details of the system are obscured. To deal with this complexity, a number of techniques for automatically simplifying the overall graph structure have been explored [8, 19]. In *replay* we also provide a means for filtering the graph, providing the user with direct control over which properties to filter from the display as well as providing the ability to implement automatic filtering through the plug-in extension system as described in Section 2.7.

Filters can be created which specify a list of specific nodes, or a glob [12] style pattern to match the names of nodes, against which the filter is applied. The filter can then specify that these nodes are either grouped, or hidden, or can override the properties (colour etc.) of the nodes. Groups are then represented as the aggregate of their component nodes within the different visualisations. The timeline view uses a single entry which draws the timelines of the component nodes overlayed upon one-another, while the network graph represents a group as a single node with the combined mass of its components, hiding all internal edges between nodes within the group. Hidden nodes are removed from all views (and any edges or messages in which they are involved).

This allows the user to selectively hide extraneous information while retaining that which is pertinent to the current analysis. This in turn allows the user to reduce their cognitive load and hence focus on the problem at hand. The use of such filtering has been successfully demonstrated in the analysis of Annex object capability based software [23], which will be explored in Section 3.1.

2.7 Plug-in / Extension system

Originally *replay* was built as a tool to help analyse and debug the Annex object capability system and, as a result, was initially tailored to suit the specifics of Annex. However, it was soon realised that the different visualisations within *replay* could be very useful in analysing other systems including other object-capability / object-oriented programming systems or social networks. A plug-in system was developed to enable *replay* to be easily extended and used to visualise other diverse, inter-connected systems.

Plug-ins can be used to extend *replay* in multiple ways:

Event Sources The initial motivation for the development of the plug-in system was to provide support for the translation of custom data sets into the specific events described in Section 2.1. Hence plug-ins can provide and register event sources for the main *replay* application, allowing *replay* to easily support a wide range of systems. Multiple types of event sources are supported, including disk-based file sources for offline visualisation, or network connected sources for visualisation of live systems.

Analysis Plug-ins also have access to the various data-structures within the core application, such as the sequence of events, the node-edge graph, and the list of filters. This allows for a number of extensions to be implemented, such as performance analysis or automatic filtering by the creation of custom filters. As an example, a plug-in could easily analyse the graph at a given point in time to determine disjoint sub-graphs. By accessing the filter list, it could then create filters to select the nodes in each separate graph and override their colours. This would then provide a simple visual cue of the separate graphs to the user without the need for manual intervention.

Extended Functionality The plug-in system has been used to implement a number of the core features for *replay*, including playback controls (allowing the user to automatically play forwards and backwards through the events), as well as the filtering system presented in Section 2.6.

A number of plug-ins have been developed to extend the utility of *replay*.

Annex The original Annex specific code from *replay* was re-factored into a single plug-in which interprets the custom Annex event log and produces appropriate *replay* events. An example of the output from this is seen in Figure 1.

Java To demonstrate the utility of *replay* as a general tool for the visualisation and analysis of object-oriented programming languages, a plug-in is being developed to interface with the output from the OKTECH Profiler [2] for the Java Virtual Machine to allow generic Java programs to be visualised. This currently provides support to visualise the object reference graph through time, differentiating references obtained through object creation, method invocation and return value by using different colours for each. This plug-in also provides the ability to view the execution of methods upon objects through time including their method signatures.

Due to limitations in the OKTECH profiler and the nature of the Java garbage collector there is currently no support for determining when references are dropped, and so references simply accumulate in the graph. Even without this complete support, we believe that with this existing plug-in *replay* provides almost complete support for the visualisation of Java programs, for which a clear need has previously been identified [9].

FDR A plug-in has been developed to aid in the task of formal analysis of object-capability security patterns [22] which translates the output of the FDR [10] model checker into appropriate *replay* events. This will be discussed further in Section 3.2.

Graphvis A plug-in has also been developed to translate the Graphvis [11] dot-format graph descriptions into *replay* node and edge events to allow these graphs to be visualised in an interactive, three-dimensional display using the force-directed layout of the network graph view.

Causeway The previously mentioned Causeway message-oriented debugger was designed to debug concurrent message passing systems such as the object-capability E programming language [20] and the Waterken web server [4]. We have developed a plug-in to translate the Causeway message log format [1] into *replay* events to allow these systems to be visualised.

By separating the logic required to parse and interpret custom data sets into different plug-ins, we have been able to focus on the core visualisation technologies within *replay* itself. The following section will discuss the use of *replay* in the analysis of real-world systems, describing its utility and benefits as a general purpose visualisation tool.

3 Case studies

3.1 The Annex Capability System

The original motivation behind the development of *replay* was to develop a tool for debugging and analysing the security properties of the Annex object-capability system (which will be referred to as simply Annex for the sake of brevity). Annex serves as the Trusted Computing Base (TCB) in a number of secure devices developed by DSTO Australia [13, 23]. The Annex TCB is used to control the security policy for these devices, and hence the correctness of the Annex system is crucial to ensuring the security of the devices as a whole.

Within Annex, and other object-capability systems, objects can only communicate with one-another by message passing, and they can only pass messages if they have an appropriate capability which designates the other object. As a result the collection of capabilities which an object possesses defines the authority of the object within the system [20]. As capabilities can be delegated from one object to any other that they are already in communication with, the object graph (where objects are nodes, connected via capabilities) is a dynamic entity which is constantly changing as the system evolves. The ability to easily visualise this graph and hence quickly 'see' the security policy / posture of the system embodied by the graph was the primary motivation in the development of *replay*. Once the graph view was developed, it was also realised that as well as visualising the security properties of the system, the ability to visualise the behaviour of the system through time would also help in analysis and debugging. Hence the timeline view was added. Similarly, the need to track the causality of messages (what caused this message to be sent) was identified, and resulted in the addition of the message tree view.

replay has served as a significant tool in the development of Annex by allowing the entire execution of the system to be visualised. This has been particularly useful in a number of situations, some of which are summarised in the follow sections.

3.1.1 Timing and race-conditions

Annex objects interact by method calls / returns in a turn-based fashion using an asynchronous promise [20] model. An object is active and uninterruptable while processing a call (and this defines a single turn for the object), but it is idle while waiting for the response to other calls it makes. As a result, while waiting on the result from a call an object has made, it can be invoked by either another incoming call, or a response to a different call it has previously made, which will start the execution of a new, independent turn. This feature of the Annex systems allows high levels of concurrency, since multiple turns (and hence multiple method calls) can potentially be interleaved without needing to wait for each to synchronously execute, and also allows a high level of parallelism to be achieved, while still guaranteeing atomicity between turns. However, without careful attention to turn boundaries this feature can also lead to potential race-conditions and security critical bugs. *replay* has proved useful in helping to track down these particular issues by clearly showing the concurrent and interleaved execution of calls within a single object. The Annex plug-in colours each different call separately and so the interleaving of different calls to a single object is clearly visible within the timeline view, as seen in Figure 2.

This Figure shows the execution of the *tortureAsync* application which comprises 1 driver object (*tortureAsync*) and 16 worker objects *oTortureAsync* repeatedly calling one-another, and is designed to stress-test Annex's message-passing performance. The execution of the top-most *oTortureAsync* object clearly demonstrates this interleaving. This object is initially called by the driver *tortureAsync* and starts execution (shown by the blue activity line) and proceeds to call 4 of the other worker objects, including itself. It then suspends execution to wait for the returns from these calls. Almost immediately a return is received from the first object which it called (again shown by the same colour blue activity line), at which point the initial call is resumed to store this result, and execution is again suspended. However, since the object is now idle, the call which it made to itself is now delivered, shown by the green activity line. This starts a new turn, which is separate from the one used to execute the original call (indicated by the different colours) and clearly demonstrates that these two calls have been interleaved on this object. In interleaving these calls, if the second call happens to modify state which the first call is expecting to remain constant, then a race-condition will result. However, the timeline view of *replay* clearly allows this to be identified and flagged to the programmer. It should be noted that the causal view will not help identify this same potential for error since it only highlights the causal, i.e. partial ordering, not total ordering.

Figure 2 also clearly shows the ability to measure the time taken to execute particular calls - the time taken to execute the call from *tortureAsync* to each of the worker *oTortureAsync*'s is clearly longer than the time taken to execute the calls made between each worker object.

3.1.2 Authority analysis

As the purpose of the *tortureAsync* application is to simply make repeated calls between each of the worker objects, there is no need for these objects to have capabilities to any other object within the system, except for the other worker objects. This design follows the principle of least authority, which states that an object should only have the minimum authority required to perform its intended function, and no more [21]. We can easily analyse the authority of the application by inspecting the object graph within the network graph view of *replay*, as shown in Figure 3.

From simple inspection of the Figure, we can see the 16 worker objects of the application situated in the bottom right, with the rest of the objects comprising the other applications of the system in the left of the Figure. It is clear that these are two distinct and separate graphs, i.e. there are no capabilities connecting the worker objects to the rest of the system. *replay* therefore provides the ability to quickly verify the intended security-related isolation properties of this system by simple inspection of the graph. While this is clearly useful, the usability of visual inspection decreases with the complexity of the system at hand. Therefore for more complex analysis, as previously mentioned, the plug-in system provides the ability to directly access various data structures maintained by *replay* (such as the graph structure), allowing programmatic analysis of various properties of the system to be implemented as needed.

3.2 FDR Model Checker

The FDR plug-in was developed to visualise the trace output from the FDR model checker [10] when analysing CSP [15, 25] models of object-capability security patterns [22]. Communicating Sequential Processes (CSP) is a process algebra used to describe concurrent message-passing systems and allows formal models of such systems to be constructed. The correctness of such formal models can be stated and tested using *refinement checks* which can be evaluated by the Failures-Divergences-Refinements (FDR) model-checker to ensure correctness of the system. Murray [22] describes the use of CSP to model object-capability security patterns and the use of FDR to test the security properties of such models: CSP is used to construct a model for the object-capability pattern, which expresses the desired security properties for the system as well as the potential behaviour of its components. FDR is then used to test whether these security properties hold, and if not will return a counter example of the system's behaviour which violates the properties. One such counter-example, taken from the work of Murray [22] in analysing the *Sealer-Unsealer* pattern for object-capability systems is as follows:

```
TheDriver.Alice.Call.null,
Alice.TheUnsealer.Call.Alice,
TheUnsealer.TheSlot.Call.null,
TheSlot.TheUnsealer.Return.null,
TheUnsealer.Alice.Call.null,
Alice.TheDriver.Return.null,
TheDriver.Bob.Call.null,
Bob.TheBox.Call.null,
TheBox.TheSlot.Call.TheCash,
TheSlot.TheBox.Return.null,
TheBox.Bob.Return.null,
Bob.TheDriver.Return.TheBox,
TheDriver.Alice.Call.null,
Alice.TheUnsealer.Return.null,
TheUnsealer.TheSlot.Call.null,
TheSlot.TheUnsealer.Return.TheCash,
TheUnsealer.Alice.Return.TheCash,
Alice.TheCash.Call.TheDriver
```

From this trace output alone, and with no prior background information as to the example, it is almost impossible to determine the error which this counter-example expresses. However when visualised by *replay* (Figure 4), one aspect stands out as anomalous.

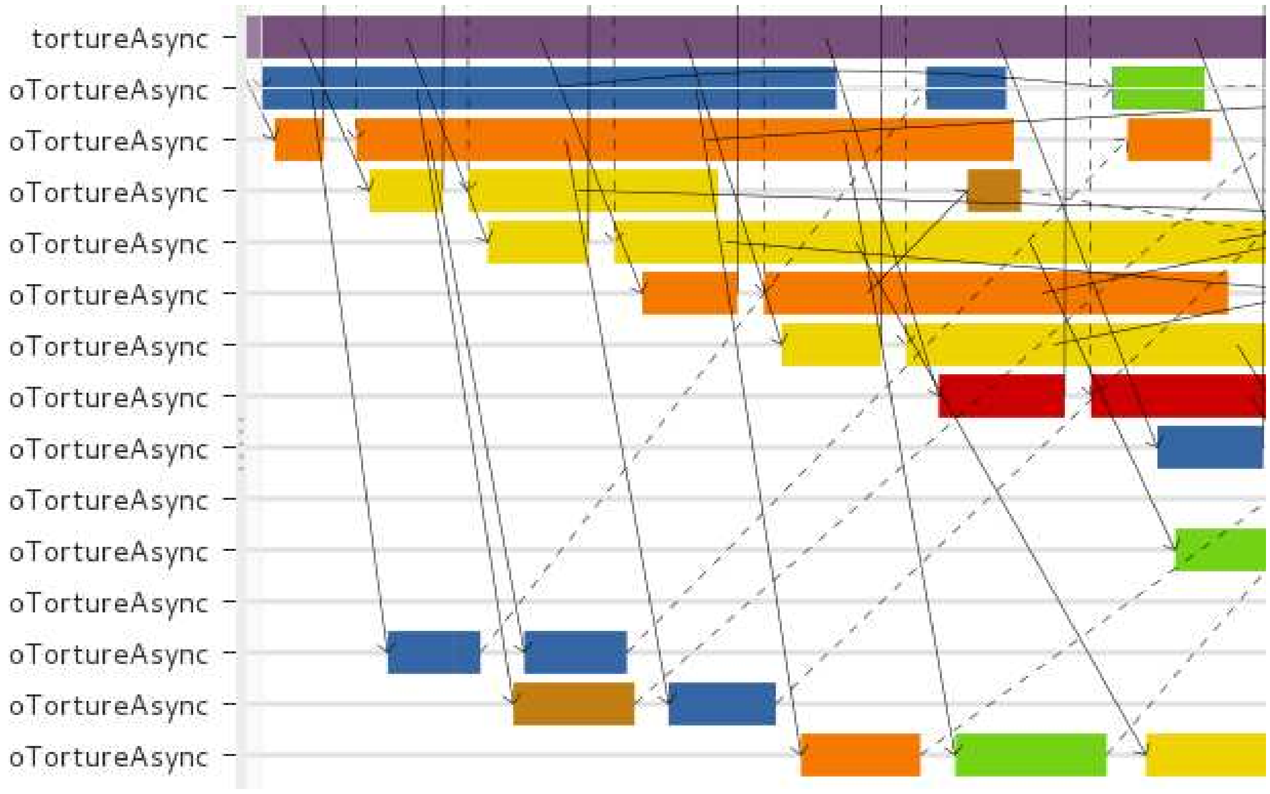


Figure 2: Timeline view of *replay* highlighting the interleaving of calls within objects in the *tortureAsync* application of the Annex system

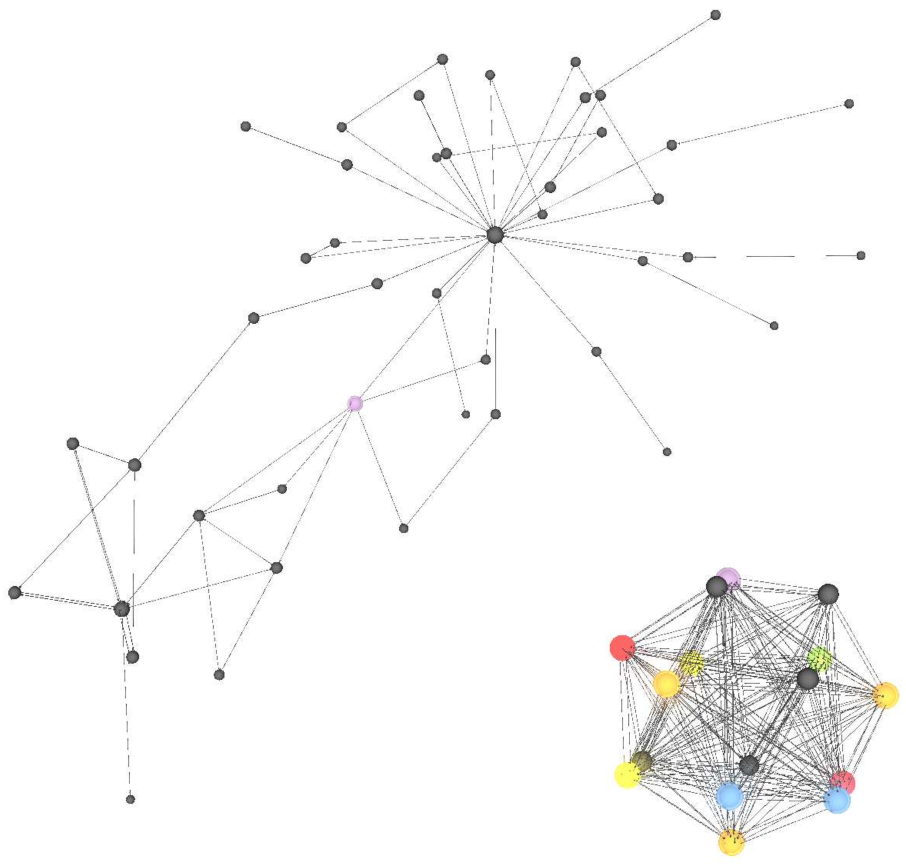
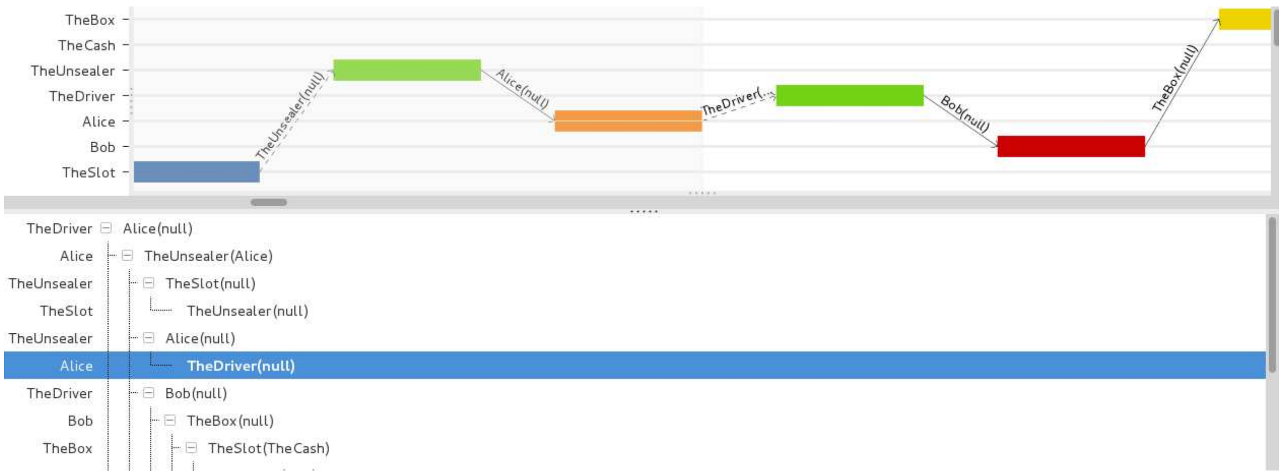


Figure 3: The Object-Capability Graph for the Annex system executing the *tortureAsync* application as depicted by the network graph view of *replay*. Note the complete isolation of the two graphs.



```
TheDriver    ⊟ Alice(null)
Alice          ├⊟ TheUnsealer(Alice)
TheUnsealer        ├⊟ TheSlot(null)
TheSlot            │   └ TheUnsealer(null)
TheUnsealer        ├⊟ Alice(null)
Alice              │   └ TheDriver(null)
TheDriver          ├⊟ Bob(null)
Bob                │  ├⊟ TheBox(null)
TheBox             │  │  ├⊟ TheSlot(TheCash)
```

Figure 4: Message tree and timeline views of *replay* when analysing the output from the FDR model checker

Without any other knowledge of the system to inform our analysis, we can see the potential error clearly: the object *Alice* is called by *TheUnsealer* but then proceeds to return to *TheDriver*. This sequence of events is impossible in languages which impose strict call / return semantics, and indeed is responsible for the error in the model. By using *replay* to visualise this sequence of events we are able to quickly and easily identify the error in the model, a task which is not so easily accomplished by simply looking at the raw trace output alone. This example demonstrates the utility of *replay* in translating the user *unfriendly* output from this formal analysis tool into a much more easily understood visual form.

4 Related Work

Various *individual* aspects of *replay* are similar to existing visualisation tools. For example, the use of a three-dimensional force-directed network graph view is common for visualising scale-free networks [16, 19, 6, 3]; the causal message tree view is similar to that employed by Causeway [27]; and the time-line is also a standard technique for showing parallelism and message passing within systems [18, 29, 14] .

However, although the individual visualisations used by *replay* are not new, the *combination* of all three views plus an event model and plugin system make *replay* unique and interesting. We are not aware of any other tool which combines such disparate visualisations in such a coherent manner to provide a comprehensive system for understanding how networks change through time.

5 Future work

While Annex provided the initial motivation for the development of *replay*, the current and future directions for the project lay in applying the general information visualisation abilities of *replay* to a wider range of systems. The existing list of plug-ins already developed shows the ability of *replay* as a visualisation tool for general object-capability / object-oriented programming systems, and for general parallel, message passing systems.

Although *replay* has proven effective in visualising numerous software systems, we believe it would also be apt in visualising a wide range of existing real-world interconnected systems such as WWW hyperlink networks and computer networks including real-time data flows within such networks. *replay* could also be useful when applied within the field of forensic analysis of computer systems to visualise communication networks of suspects, and we believe *replay* would also be well suited to visualising real world social networks. The network graph view of *replay* is quite similar to existing social network graph visualisations [26, 16], and so is well suited to visualising the structure of such networks. We also believe the timeline view showing interactions through time, as well as the causal view showing the relationships between communications would provide valuable insight in understanding these networks which the previously cited tools do not provide. We also believe a similar approach could be used to visualise the transmission of email or instant messages, to determine the structure and behaviour of such communications.

While the manual filtering already provided by *replay* allows easy analysis by removing extraneous information, it does not yet apply to the causal message tree view, but this could be done in the future. It would also be useful to investigate the utility of applying automatic filtering and grouping mechanisms via the plug-in system, as well as implmenting various graph and performance analysis algorithms.

Finally, extensions to the plug-in system enabling other visualisations of the existing data structures, such as different layout algorithms for the network graph view, could also be developed.

6 Conclusion

In this paper we have presented *replay*, a novel tool for the visualisation of concurrent networked systems. We have shown the unique aspects of *replay* including its programmable event model and its three synchronised and related visualisations. The plug-in system, which allows *replay* to be applied to a wide variety of applications has also been presented and a number of existing uses of *replay* have been described, demonstrating its clear utility. Finally, we have contrasted *replay* against existing tools and presented possible future directions for this work. We believe that the use of a generic, programmable event model, the combination of the three different but consistent views of these events and an extensible plug-in system make *replay* a unique tool with both a high degree of usability and utility for the visualisation and analysis of networked, parallel systems.

References

[1] Debugging a Waterken application. Available online - http://waterken.sourceforge.net/debug/.

[2] OKTECH Profiler. Available online - http://code.google.com/p/oktech-profiler/.

[3] Walrus - graph visualization tool. Available online - http://www.caida.org/tools/visualization/walrus/.

[4] Waterken server documentation. Available online - http://waterken.sourceforge.net/.

[5] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286, 1999.

[6] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In *Proceedings of the International AAAI Conference on Weblogs and Social Media*, 2009.

[7] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization: Using Vision to Think.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[8] G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-laws in a large object-oriented software system. *IEEE Trans. Softw. Eng.*, 33(10):687–708, 2007.

[9] C. Exton and M. Kölling. Concurrency, objects and visualisation. In *ACSE '00: Proceedings of the Australasian conference on Computing education*, pages 109–115, New York, NY, USA, 2000. ACM.

[10] Formal Systems (Europe) Limited. *Failures Divergences Refinement - FDR2 User Manual*, 2009. Available online - http://www.fsel.com/documentation/fdr2/html/.

[11] E. R. Gansner and S. C. North. An open graph vizualisation system and its applications to software engineering. *Software - Practice and Experience*, 30:1203–1233, 1999.

[12] GNU. *GLOB(7) - glob - globbing pathnames*, August 2003. Linux Programmer's Manual 'man' page - http://www.kernel.org/doc/man-pages/.

[13] D. A. Grove, T. C. Murray, C. A. Owen, C. J. North, J. A. Jones, M. R. Beaumont, and B. D. Hopkins. An overview of the Annex system. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 341–352, December 2007.

[14] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.

[15] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[16] Y. Jia, J. Hoberock, M. Garland, and J. Hart. On the visualization of social and other scale-free networks. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1285–1292, 2008.

[17] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[18] E. Kraemer and J. T. Stasko. Creating an accurate portrayal of concurrent executions. *IEEE Concurrency*, 6(1):36–46, 1998.

[19] G. Kumar and M. Garland. Visual exploration of complex time-varying graphs. *IEEE transactions on visualization and computer graphics*, 12(5):805, 2006.

[20] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[21] M. S. Miller and J. S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *Proceedings of the 8th Asian Computing Science Conference (ASIAN03)*, pages 224–242, December 2003.

[22] T. Murray. Analysing object-capability security. In *Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08)*, 2008.

[23] C. Owen, D. Grove, T. Newby, A. Murray, C. North, and M. Pope. PRISM: Program Replication and Integration for Seamless MILS. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 281–296, Washington, DC, USA, 2011. IEEE Computer Society.

[24] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in OO programs. *Commun. ACM*, 48(5):99–103, 2005.

[25] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.

[26] L. Shi, N. Cao, S. Liu, W. Qian, L. Tan, G. Wang, J. Sun, and C. Lin. HiMap: Adaptive visualization of large-scale online social networks. In *Proceedings of the 2009 IEEE Pacific Visualization Symposium-Volume 00*, pages 41–48. IEEE Computer Society, 2009.

[27] T. Stanley, T. Close, and M. S. Miller. Causeway: A message-oriented distributed debugger. Technical report, HP Laboratories, 2009.

[28] M. Q. Wang Baldonado, A. Woodruff, and A. Kuchinsky. Guidelines for using multiple views in information visualization. In *AVI '00: Proceedings of the working conference on Advanced visual interfaces*, pages 110–119, New York, NY, USA, 2000. ACM.

[29] J. Yan, S. Sarukkai, and P. Mehra. Performance measurement, visualization, and modeling of parallel and distributed programs using the aims toolkit. *Software Practice and Experience*, 25(8):429–461, 1995.

[30] Y. Yao, S. Huang, Z. ping Ren, and X. ming Liu. Scale-free property in large scale object-oriented software and its significance on software engineering. *Information and Computing Science, International Conference on*, 3:401–404, 2009.

On Connected Two Communities

V. Estivill-Castro[1] **Mahdi Parsa**[1]

[1] School of Information and Communication Technology
Griffith University,
Nathan, Qld, 4111, Australia.

Abstract

We say that there is a community structure in a graph when the nodes of the graph can be partitioned into groups (communities) such that each group is internally more densely connected than with the rest of the graph. However, the challenge is to specify what is to be *dense*, and what is *relatively more connected* (there seems to exist an analogous situation to what is a cluster in unsupervized learning). Recently, Olsen (2012) provided a general definition that seemed to be significantly more generic that others. We make two observations regarding such definition. (1) First, we show that finding a community structure with two equal size communities is *NP*-complete (Uniform 2-Communities). The first implication of this is that finding a large community seems intractable. The second implication is that, since this is a hardness result for $k = 2$, the Uniform k-Communities problem is not fixed-parameter tractable when k is the parameter. (2) The second observation is that communities are not required to be connected in Olsen (2012)'s definition. However, we indicate that our result holds as well as the results by Olsen (2012) when we require communities to be connected, and we show examples where using connected communities seems more natural.

Keywords: Community detection, graph partitioning, complexity, parameterized complexity

1 Introduction

Researchers are now focusing on analyzing the community structure (Boccaletti et al. 2006, Lancichinetti et al. 2010) of graphs and finding so called *communities* or modules (intuitively these are groups of nodes that are more densely connected to each other than with the rest of the graph). Exploring communities in graphs is important (Lancichinetti et al. 2010) because 1) communities uncover the graph at a coarse level, for example, formulating realistic mechanisms for its genesis and evolution 2) communities provide a new aspect for understanding dynamic processes occurring in the graph and 3) communities reveal relationships among the nodes that are not apparent when inspecting the graph as a whole.

Recently, there has been a large research focus on community structures in graphs (Condon & Karp 2001, Fortunato 2010, Gargi et al. 2011, Kevin J. Lang et al. 2009). However, the main problem is how to define communities in the first place. This is the essential issue tackled by most papers on the topic which have appeared in the literature (Fortunato 2010, references therein). Here we consider the most recent definition of community structure introduced by Olsen (2012). This definition is inspired by the planted l-partition model, and the hierarchical random graph model introduced by Condon & Karp (2001). Olsen (2012) was able to justify why this becomes a more suitable (and formal) definition of community and initiated the study of the complexity of finding communities by showing that it is *NP*-complete to decide if a group of nodes can be extended to a community in some community structure.

We introduce this generic notion of community using the following notation. Let Π be a partition of the vertices V of a graph $G = (V, E)$ ($\Pi = \{C_1, C_2, \ldots, C_k\}$, with $\emptyset \neq C_j \subset V$ for $j = 1, \ldots, k$ and $\bigcup_{j=1}^{k} C_j = V$, and $C_j \cap C_{j'} = \emptyset$ for $j \neq j'$). If $i \in V$, then we denote the part vertex i belongs to by Π_i. Let $i \in V$ be a vertex and $S \subset V$, then $N_i(S)$ is the number of vertices in S that are neighbors (adjacent) to the vertex i (a vertex is never considered adjacent to itself).

Definition 1.1 *A community structure for an undirected connected graph $G = (V, E)$ is a partition Π of V such that*

1. $|\Pi| \geq 2$ *(we have at least 2 communities),*
2. $|C| \geq 2$ *for all* $C \in \Pi$ *(every community has at least 2 members) and*
3. $\forall i \in V, \quad \forall C \in \Pi$ *the following holds*

$$\frac{N_i(\Pi_i)}{|\Pi_i| - 1} \geq \frac{N_i(C)}{|C|}. \tag{1}$$

Each set of the partition is called a community.

Olsen (2012) also showed that finding a community structure in a graph that does not contain S_n (the stars of n vertices), for $n \geq 3$ can be done in polynomial time. However, nothing could be said about the community structure, like if large communities could be found. Also, it was left open any claim whether finding community structures with few communities is tractable or not. Thus, we investigate here the question of finding a community structure with two communities. That ensures one community is large as it must include at least half of the vertices. It turns out that this investigation reveals one more aspect regarding Definition 1.1. We direct the reader to the observation that communities are not required to be connected. That is, each part C is not required to be connected. Why we suggest communities be connected? Because it is hard not to consider the connected components of a disconnected "community" more naturally as communities in themselves. In fact, the lack of links (vs links to other parts of the graph) suggest the connected components are not to be placed together. We also consider uniform community structure, that is, all communities have the same size. The uniform community structure has gained importance due to its application for clustering and detection of cliques in social, pathological and biological networks (Patkar & Narayanan 2003).

We start with a discussion on the complexity of finding 2-Communities. Why we look at the problem of two communities rather than the problem with k communities? Because by showing the problem with 2 communities is hard, we are showing the problem with k communities is also hard. Why we look at equal size communities? Because this forces the communities to be large. It seems in practice, the larger a community, the more interesting. We prove that when we require the communities to have equal size the problem is *NP*-complete. This result suggests that other lines of attack may be required. For example, a very successful avenue of attack has recently been the application of parameterized complexity theory. Such approach can lead to polynomial-time algorithms on the size of the input (at the cost of exponential-time complexity on the parameter, which can be small in practical settings). A first natural parameter is the number k of communities. That is, to consider the question whether, for a given graph G, there exists a community structure with exactly k communities. We call this problem k-Communities. Because we will show that for $k = 2$, the problem Uniform k-Communities (where communities are all of the same size) is *NP*-complete, the problem Uniform k-Communities is not fixed-parameter tractable when k is the parameter. In other words, it is unlikely to have an algorithm for this problem with $f(k) \cdot poly(|G|)$ time requirements, for some computable function f.

2 Uniform Two-Communities is hard

In this section, we formally define our problem and then show our main hardness result (Theorem 2.1). Our proof is inspired by a hardness result for a graph partitioning problem (Bazgan et al. 2010). We prove this results in several steps.

Uniform k-Communities
Instance: A graph $G = (V, E)$.
Parameter: An integer $k > 1$.
Question: Does a community structure $\Pi = \{C_1, C_2, \ldots, C_k\}$ exist such that $|C_i| = |C_j|$ for $i, j = 1, \ldots, k$?

Theorem 2.1 Uniform 2-Communities *is* NP-*complete.*

Uniform 2-Communities belongs to the class *NP*. Because, we can verify, in polynomial time, whether a partition of size two constitutes (with equal parts) a community structure. For the hardness part of the theorem, we give a polynomial reduction from a variant of the Clique problem to the Uniform 2-Communities problem. The version of the Clique problem that asks, for a given non-complete graph G of size n (n is even), whether there exists a complete subgraph of size at least $n/2$. This version of the Clique problem is also *NP*-complete (Garey & Johnson 1979), and it is not hard to see that the version we will use (whether a graph has a clique of size $n/2$) is also *NP*-complete. Now we construct our reduction and we will show that every Yes-instance of the Clique problem maps to a Yes-instance of the Uniform 2-Communities problem and vice versa.

Construction 1 *Let $G= (V, E)$ be an instance of the* Clique *problem with $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots e_m\}$ (with $|E| = m > 0$). Let p be the number of non-edges in G, that is $p = n \times (n-1)/2 - m$. The value of p is at least one, as the graph G is a non-complete graph. Suppose we label the non-edges in G by $ne_1, \ldots, ne_p$. We construct an instance $G'' = (V'', E'')$ of the* 2-Communities *problem as follows. The vertex set V'' consists of four disjoint sets, F, T, V and V'. That is, $V'' = F \cup T \cup V \cup V'$. The set V is the original set of vertices in the instance of the* Clique *problem; the set $V' = \{v'_1, \ldots, v'_n\}$ consists of as many mirror vertices as in the original set V of vertices. The set $F = \{f_1, \ldots, f_{2p+1}\}$, has two vertices f_{2l}, f_{2l+1} for each non-edge ne_l with $l = 1, \ldots, p$ and f_1 is an additional vertex. The set $T = \{t_1, \ldots, t_{2p+1}\}$ also has two vertices t_{2l}, t_{2l+1} for each non-edge in the original instance of the* Clique *problem, and also t_1 is an additional vertex.*

We now describe the set of edges E''. The set E of original edges among vertices in V is in E''; that is $E \subset E''$. In the new instance, F and T are two cliques of size $2p + 1$ (that is, in E'', all vertices of F are connected among themselves and also in E'', all vertices of T are connected among themselves). For $j = 1, \ldots, n$, (v'_j, v_j) is in E''. The edge set E'' contains some additional edges as follows:

- *Each vertex $t \in T$ connects to all vertices of V.*
- *Each vertex $f \in F$ connects to all vertices of V, unless*
  - *f is of the form f_{2l} or f_{2l+1}*
  - *and $ne_l = (v_i, v_j)$ is the missing edge (with $i < j$) in G corresponding to the pair (f_{2l}, f_{2l+1}).*

  In this case, the vertex f_{2l} connects to every vertex in $V \setminus \{v_j\}$, and f_{2l+1} connects to every vertex in $V \setminus \{v_i\}$.

Finally, the edge (f_1, t_1) is in E''.

Note the following about this construction. First, the degree of all vertices in V' is one, as these vertices are only connected to their mirror vertices. Second, the degree of every vertex $t \neq t_1$ in T is $|V| + |T| - 1 = n + 2p$, and the degree of t_1 is $|V| + |T| = n + 2p + 1$. This is because t is in clique T (degree $|T| - 1$) and it is connected to each vertex in V and t_1 is additionally connected to f_1. Third, the degree of every vertex $f \in F$ is at least $|F| - 1$ as it belongs to the clique F. The vertex f_1 has degree $|F| + |V|$, but the other vertices in F have degree $|F| + |V| - 2$, as each of these vertices looses one connection to one vertex in V that is an endpoint of a non-edge.

Figure 1 provides a more specific example of the reduction. Clearly, this construction can be performed in polynomial time. We only need to show that a Yes-instance of the first problem maps to a Yes-instance of the second problem and vice versa.

Proposition 2.2 *A* Yes-*instance of the* Clique *problem maps to a* Yes-*instance of the* Uniform 2-Communities *problem.*

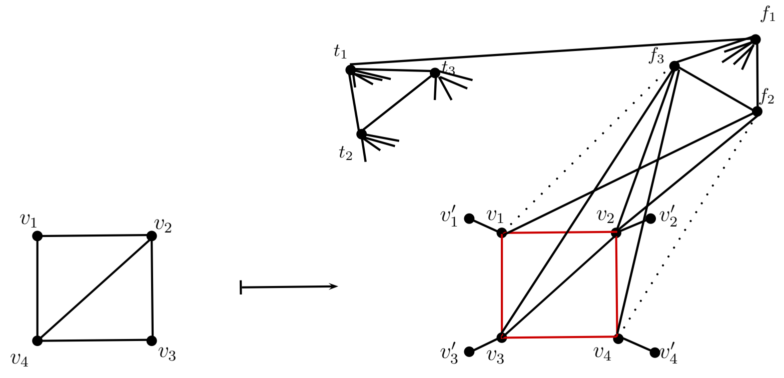


Figure 1: The dotted lines mean that there is no edge between the two end points of the line. A branch of four edges at f_1 and each vertex of T mean those vertices connect to all vertices of V.

Proof: First, assume that the graph G has a clique C of size $n/2$. We define a partition of size two of the graph G'' by considering the first set as $\Pi_1 = F \cup C \cup C'$ where $C' = \{v'_i : v_i \in C\}$ and the second set as $\Pi_2 = T \cup \bar{C} \cup \bar{C}'$, where $\bar{C} = V - C$ and $\bar{C}' = \{v'_i : v_i \in \bar{C}\}$. We show these two sets constitute a community structure of size two.

We must verify Inequality (1) for the three types of vertices that appear in $\Pi_1 = F \cup C \cup C'$ and also for the three types of vertices in $\Pi_2 = T \cup \bar{C} \cup \bar{C}'$. We start with Π_1, and in particular with vertices in C. Then, vertices in F (this will require three cases) and then C'. When dealing with Π_2, we sill start with $\bar{C}$, then T and finally $\bar{C}'$.

Let c be a vertex in the clique C. We consider $x_c = n - 1 - N_c(V)$, that is the number of non-edges in the graph G with one end-point in the vertex c. Then, by construction, the vertex c is not linked to x_c vertices of the clique F. Since C is a clique of size $n/2$ in V, then it involves at least half of the vertices of V, that is $|C| \geq |\bar{C}|$. Also, by construction we have $|F| = |T|$. We can then see that $N_c(\Pi_1)$ equals $|F| + |C| - x_c$, because the vertex c connects to $|C| - 1$ vertices of the clique C, its mirror c' and $|F| - x_c$ vertices in F. Therefore, we have

$$\begin{aligned}\frac{N_c(\Pi_1)}{|\Pi_1| - 1} &= \frac{|F| + |C| - x_c}{|\Pi_1| - 1} \\ &\geq \frac{|T| + |\bar{C}| - x_c}{|\Pi_1| - 1} \\ &> \frac{|T| + |\bar{C}| - x_c}{|\Pi_1|} = \frac{|T| + |\bar{C}| - x_c}{|\Pi_2|}.\end{aligned}$$

Now, we compute $N_c(\Pi_2)$. The vertex c in the clique C connects to every vertex in T and to every vertex v on $\bar{C}$ unless (c, v) is a non edge. Moreover, all non-edges with an endpoint in c must have an endpoint in $\bar{C}$ as C is a clique. Therefore, we have

$$\frac{N_c(\Pi_2)}{|\Pi_2|} = \frac{|T| + |\bar{C}| - x_c}{|\Pi_2|}.$$

This implies that the vertex c satisfies Inequality (1).

Now we need to show that every vertex in F also satisfies Inequality (1) since the second type of vertex in Π_1 are the vertices in F.

Consider $f \in F$. According to our construction the size of the clique F is at least three ($|F| \geq 3$) and we will face the following cases.

Case 1: $f \neq f_1$, and f connects to every vertex in C.

In this case $N_f(\Pi_1)$ equals $|F| - 1 + |C|$, since the vertex f connects to every vertex in C. Also, we have $|\bar{C}| \geq N_f(\Pi_2)$, therefore

$$\begin{aligned}\frac{N_f(\Pi_1)}{|\Pi_1| - 1} &= \frac{|F| - 1 + |C|}{|\Pi_1| - 1} \\ &\geq \frac{|F| - 1 + |\bar{C}|}{|\Pi_1| - 1} \\ &> \frac{|F| - 1 + |\bar{C}|}{|\Pi_1|} \\ &= \frac{|F| - 1 + |\bar{C}|}{|\Pi_2|} \\ &> \frac{|\bar{C}|}{|\Pi_2|} \geq \frac{N_f(\Pi_2)}{|\Pi_2|}.\end{aligned}$$

Case 2: The vertex f connects to every vertex in C except one.

We recall that the degree of every vertex in F that is not f_1 is $|F| + |V| - 2$. Since $|F| \geq 3$, then we have

$$\begin{aligned}\frac{N_f(\Pi_1)}{|\Pi_1| - 1} &= \frac{|F| - 1 + |C| - 1}{|\Pi_1| - 1} \\ &\geq \frac{|F| - 1 + |\bar{C}| - 1}{|\Pi_1| - 1} \\ &> \frac{|F| - 2 + |\bar{C}|}{|\Pi_1|} = \frac{|F| - 2 + |\bar{C}|}{|\Pi_2|} \\ &\geq \frac{|\bar{C}|}{|\Pi_2|} = \frac{N_f(\Pi_2)}{|\Pi_2|}.\end{aligned}$$

Case 3: $f = f_1$.

According to the construction, f connects to every vertex in C and also connects to t_1. Hence, we have

$$\begin{aligned}\frac{N_f(\Pi_1)}{|\Pi_1| - 1} &= \frac{|F| - 1 + |C|}{|\Pi_1| - 1} \\ &\geq \frac{|F| - 1 + |\bar{C}|}{|\Pi_1| - 1} \\ &> \frac{|F| - 1 + |\bar{C}|}{|\Pi_1|} \\ &= \frac{|F| - 1 + |\bar{C}|}{|\Pi_2|} \\ &\geq \frac{1 + |\bar{C}|}{|\Pi_2|} \geq \frac{N_f(\Pi_2)}{|\Pi_2|}.\end{aligned}$$

The last type of vertex in Π_1 that we check for Inequality (1) belongs to C', but these vertices have degree 1 in Π_1 and degree zero in Π_2, so this is immediate.

To complete the proof that we have a YES-instance of 2-COMMUNITIES we need to establish Inequality (1) for the vertices in Π_2. We start by showing that Inequality (1) holds for every vertex c in $\bar{C}$.

First, $N_c(\Pi_2) = |T| + 1 + N_c(\bar{C})$, since c connects to all vertices in T, all its neighbors in $\bar{C}$ and also connects to its mirror c'. Second, assume x_c is the number of non-edges in $\bar{C}$ with endpoint in c, then we have $N_c(\Pi_2) = |T| + 1 + |\bar{C}| - 1 - x_c = |T| + |\bar{C}| - x_c$. Third, if there exists a missing edge (e, v) with $v \in \bar{C}$, corresponding to this missing edge, there exists exactly a missing edge between c and a vertex $f \in F$. Therefore, $N_c(\Pi_1)$ equals $|F| - x_c + N_c(C)$. Since $|\bar{C}| = |C|$ and $|\bar{C}| \geq N_c(C)$, then we have

$$\begin{aligned} \frac{N_c(\Pi_2)}{|\Pi_2|-1} &= \frac{|T| + |\bar{C}| - x_c}{|\Pi_2|-1} \\ &\geq \frac{|F| + N_c(C) - x_c}{|\Pi_1|-1} \\ &> \frac{|F| + |C| - x_c}{|\Pi_1|} \\ &= \frac{N_c(\Pi_1)}{|\Pi_1|}. \end{aligned}$$

We now argue for the second type of vertices in Π_2. We show that every vertex t in T satisfies Inequality (1). Since $|\Pi_1| = |\Pi_2|$, $|T| \geq 3$ and $|C| = |\bar{C}|$ we have for every $t \neq t_1$

$$\begin{aligned} \frac{N_t(\Pi_2)}{|\Pi_2|-1} &= \frac{|T| + |\bar{C}| - 1}{|\Pi_2|-1} \\ &= \frac{|T| + |C| - 1}{|\Pi_1|-1} \\ &> \frac{|T| + |C| - 1}{|\Pi_1|} \\ &> \frac{|C|}{|\Pi_1|} \\ &\geq \frac{N_t(\Pi_1)}{|\Pi_1|}. \end{aligned}$$

Similarly, for $t = t_1$ we have

$$\begin{aligned} \frac{N_t(\Pi_2)}{|\Pi_2|-1} &= \frac{|T| + |\bar{C}| - 1}{|\Pi_2|-1} \\ &= \frac{|T| + |C| - 1}{|\Pi_1|-1} \\ &> \frac{|T| + |C| - 1}{|\Pi_1|} \\ &\geq \frac{|C|+1}{|\Pi_1|} \\ &\geq \frac{N_t(\Pi_1)}{|\Pi_1|}. \end{aligned}$$

And to complete all vertices of Π_2 we consider the mirror vertices in $\bar{C}'$, but again these vertices have degree one to their community Π_2 and zero to the other part Π_1, so trivially they satisfy Inequality (1).

Therefore, a YES-instance of the CLIQUE problem maps to a YES-instance of the UNIFORM 2-COMMUNITIES problem. □

Now we show that the reverse is true.

Proposition 2.3 *A* YES-*instance of the* UNIFORM 2-COMMUNITIES *problem maps to a* YES-*instance of the* CLIQUE *problem.*

Suppose $I = (G'', \Pi_1, \Pi_2)$ is a YES-instance of the UNIFORM 2-COMMUNITIES problem. We justify the following observations to show the pre-image of I is a YES-instance of the CLIQUE problem.

Observation 2.4 *(about mirror vertices): In each* YES-*instance of* 2-COMMUNITIES, *the mirror vertices v'_j must be in the same community as v_j, with $j = 1, \ldots, n$.*

Proof: If a mirror vertex v' is in community Π_1, and its corresponding vertex v is in community $\Pi_2 \neq \Pi_1$, then $N_{v'}(\Pi_1) = 0$, while $N_{v'}(\Pi_2) > 0$. This contradicts that the vertex v' must satisfy Inequality (1). □

Observation 2.5 *The set T can not be cut by the community structure.*

Proof: (by contradiction) Suppose T is divided in (T_1, T_2) with $T_i \subseteq \Pi_i$ and $i = 1, 2$. Also assume that the set V is cut in (V_1, V_2) with with $V_i \subseteq \Pi_i$ and $i = 1, 2$, where V_1 and V_2 could be empty. Moreover, assume that F is divided in (F_1, F_2) with $F_i \subseteq \Pi_i$ and $i = 1, 2$. We will face the following cases where each one leads to a contradiction.

Case 1: $T_1 = \{t_1\}$ and $F_1 = \{f_1\}$.

In this case $|T_2|$ is $|T| - 1$ and $|F_2| = |F| - 1$. Therefore, $|T_2|$ is equal to $|F_2|$ and both are equal to $|T| - 1$ because $|T| = |F|$. Since Π is a community structure, then every $t \in T_2$ must satisfy Inequality (1). But,

$$\begin{aligned} \frac{N_t(\Pi_2)}{|\Pi_2|-1} &= \frac{|T_2| - 1 + |V_2|}{|F_2| + 2|V_2| + |T_2| - 1} \\ &= \frac{(|T|-1) - 1 + |V_2|}{(|F|-1) + 2|V_2| + (|T|-1) - 1} \\ &= \frac{|T| + |V_2| - 2}{2|T| + 2|V_2| - 3} < \frac{1}{2}, \end{aligned}$$

and
$$\frac{N_t(\Pi_1)}{|\Pi_1|} = \frac{1 + |V_1|}{2|V_1| + 2} = \frac{1}{2}.$$

This statement contradicts the vertex t must satisfy Inequality (1).

Case 2: $T_1 = \{t_1\}$ and $f_1 \in F_2$.

The neighbors of the vertex t_1 in Π_1 are all vertices in V_1 as $T_1 = \{t_1\}$. Also, the neighbors of the vertex t_1 in Π_2 are all vertices in T_2, with also all vertices in V_2 and f_1. Therefore, we have

$$\begin{aligned} \frac{N_{t_1}(\Pi_1)}{|\Pi_1|-1} &= \frac{|T_1| - 1 + |V_1|}{|F_1| + 2|V_1| + |T_1| - 1} \\ &= \frac{1 - 1 + |V_1|}{|F_1| + 2|V_1| + 1 - 1} \\ &= \frac{|V_1|}{|F_1| + 2|V_1|} \\ &\leq \frac{1}{2}. \end{aligned}$$

Since $|T_1| = 1$, $|T_2| = |T| - 1$ and $|T| + 1 > |F| \geq |F_2|$, then we have

$$\frac{N_{t_1}(\Pi_2)}{|\Pi_2|} = \frac{|T_2| + |V_2| + 1}{|T_2| + 2|V_2| + |F_2|} > \frac{1}{2}.$$

This statement contradicts the fact that the vertex t_1 must satisfy Inequality (1).

Case 3: $T_1 = \{t_1\}$ and $|F_1| \geq 2$.

Case 1 and Case 2 resulted in if $T_1 = \{t_1\}$, then the vertex f_1 must be in F_1 and $|F_1| \geq 2$. Now we consider the vertex t_1 to show that it violates Inequality (1). First, the neighbors of the vertex t_1 in Π_1 are all vertices in V_1 and f_1. Second, the size of $|F_1| \geq 2$, therefore we have

$$\begin{aligned} \frac{N_{t_1}(\Pi_1)}{|\Pi_1| - 1} &= \frac{|V_1| + 1}{|F_1| + 2|V_1| + |T_1| - 1} \\ &= \frac{|V_1| + 1}{|F_1| + 2|V_1|} \\ &\leq \frac{1}{2}. \end{aligned}$$

On the other hand, $|F_1| \geq 2$ implies that $|F_1| + |F_2| \geq |F_2| + 2$. The latter inequality implies that $|T| = |F| > |F_2| + 1$. Then, we have

$$\begin{aligned} &|T| > |F_2| + 1 \\ &\quad \Rightarrow \quad |T| - 1 > |F_2| \\ &\quad \Rightarrow \quad |T_2| > |F_2| \\ &\quad \Rightarrow \quad 2|T_2| + 2|V_2| > |T_2| + 2|V_2| + |F_2|. \end{aligned}$$

The most right inequality implies that

$$\frac{N_{t_1}(\Pi_2)}{|\Pi_2|} = \frac{|V_2| + |T_2|}{|T_2| + 2|V_2| + |F_2|} > \frac{1}{2}.$$

This statement shows that the vertex t_1 violates Inequality (1).

Case 4: $\{t_1, t\} \subseteq T_1$ where $t \neq t_1$.

The above cases imply that the set T_1 must contain another vertex $t \neq t_1$. Since the vertex $t \in T_1$ must satisfy Inequality (1), then we have

$$\begin{aligned} \frac{N_t(\Pi_1)}{|\Pi_1| - 1} &= \frac{|T_1| + |V_1| - 1}{|\Pi_1| - 1} \\ &\geq \frac{N_t(\Pi_2)}{|\Pi_2|} \\ &= \frac{|T_2| + |V_2|}{|\Pi_2|}. \end{aligned} \qquad (2)$$

Also, each vertex t' in T_2 must satisfy Inequality (1), therefore,

$$\begin{aligned} \frac{N_{t'}(\Pi_2)}{|\Pi_2| - 1} &= \frac{|T_2| + |V_2| - 1}{|\Pi_2| - 1} \\ &\geq \frac{N_{t'}(\Pi_1)}{|\Pi_1|} \\ &= \frac{|T_1| + |V_1|}{|\Pi_1|}. \end{aligned} \qquad (3)$$

From Inequality (2) we get

$$\begin{aligned} &|\Pi_2|(|T_1| + |V_1|) \\ &\quad \geq \quad (|\Pi_1| - 1)(|T_2| + |V_2|) + |\Pi_2|. \end{aligned} \qquad (4)$$

From Inequality (3) we get

$$|\Pi_1|(|T_2| + |V_2| - 1) \geq (|\Pi_2| - 1)(|T_1| + |V_1|),$$

or equivalently

$$\begin{aligned} &|\Pi_1|(|T_2| + |V_2| - 1) \\ &\quad \geq \quad |\Pi_2|(|T_1| + |V_1|) - (|T_1| + |V_1|). \end{aligned} \qquad (5)$$

Combining Inequality (4) and Inequality (5) we arrive at

$$|\Pi_1| + |\Pi_2| \leq (|T_1| + |T_2|) + (|V_1| + |V_2|).$$

This inequality contradicts to the fact that F is not empty. $\square$

Observation 2.6 *The set F can not be cut by the community structure.*

Proof: Assume that F is cut into (F_1, F_2) where $F_1 \subseteq \Pi_1$, $F_2 \subseteq \Pi_2$ and $F_1 \neq \emptyset$. Also assume that the original set V is cut into (V_1, V_2) with $Vi \subseteq \Pi_i$ and $i = 1, 2$, where V_1 and V_2 could be empty. Moreover, since T can not be split, without loss of generality we can assume that $\Pi_1 = V_1 \cup V_1' \cup F_1$, $\Pi_2 = V_2 \cup V_2' \cup T \cup F_2$. We show that F_2 is empty or we have a contradiction.

Assume that F_2 is not empty and let $f \in F_2$. Then we will face the following cases.

Case 1: $f = f_1$.

The neighbors of vertex f_1 in Π_2 are all vertices in V_2, plus all vertices in $F_2 - \{f_1\}$ and the vertex t_1. Therefore, we have

$$\frac{N_{f_1}(\Pi_2)}{|\Pi_2| - 1} = \frac{|F_2| + |V_2|}{|F_2| + 2|V_2| + |T| - 1} \leq \frac{1}{2}.$$

Similarly, the neighbors of the vertex f_1 in Π_1 are all vertices in V_1, plus all vertices in F_1, therefore,

$$\frac{N_{f_1}(\Pi_1)}{|\Pi_1|} = \frac{|F_1| + |V_1|}{|F_1| + 2|V_1|} > \frac{1}{2}.$$

This statement shows that the vertex f_1 violates Inequality (1), so it is a contradiction.

Case 2: $f \neq f_1$ and f does not connect to a vertex of V_2.

The neighbors of the vertex f in Π_2 are all vertices in V_2 except one, plus all vertices in $F_2 - \{f\}$, hence,

$$\frac{N_f(\Pi_2)}{|\Pi_2| - 1} = \frac{|F_2| - 1 + |V_2| - 1}{|F_2| + 2|V_2| + |T| - 1} < \frac{1}{2}.$$

Similarly, the neighbors of the vertex f in Π_1 are all vertices in V_1, plus all vertices in F_1, therefore,

$$\frac{N_f(\Pi_1)}{|\Pi_1|} = \frac{|F_1| + |V_1|}{|F_1| + 2|V_1|} > \frac{1}{2}.$$

This contradicts the fact that the vertex f must satisfy Inequality (1).

Case 3: $f \neq f_1$, f does not connect to a vertex of V_1 and $|F_1| \geq 2$.

The neighbors of the vertex f in Π_2 are all vertices in V_2, plus all vertices in $F_2 - \{f\}$, hence,

$$\frac{N_f(\Pi_2)}{|\Pi_2| - 1} = \frac{|F_2| - 1 + |V_2|}{|F_2| + 2|V_2| + |T| - 1} < \frac{1}{2}$$

Similarly, the neighbors of the vertex f in Π_1 are all vertices in V_1 except one, plus all vertices in F_1. Moreover, the size of $|F_1| \geq 2$, therefore,

$$\frac{N_f(\Pi_1)}{|\Pi_1|} = \frac{|F_1| + |V_1| - 1}{|F_1| + 2|V_1|} \geq \frac{1}{2}.$$

Similar to Case 1 above, we have a contradiction that the vertex f violates Inequality (1).

Case 4: If $f \neq f_1$, f does not connect to a vertex of V_1 and $|F_1| < 2$.

Since F_1 is not empty, we must have $|F_1| = 1$, and by Case 1, $F_1 = \{f_1\}$, while $|F_2| = |F| - 1$. Moreover, the vertex f_1 must satisfy Inequality (1). Therefore, we have

$$\frac{N_{f_1}(\Pi_1)}{|\Pi_1| - 1} = \frac{|V_1|}{1 + 2|V_1| - 1} = \frac{1}{2}.$$

Now, to find the value of $N_{f_1}(\Pi_2)/|\Pi_2|$, we note that f_1 is adjacent to all the vertices in F_2, all the vertices in V_2 and t_1. Moreover, $|F_2| = |T| - 1$. Thus,

$$\begin{aligned}\frac{N_{f_1}(\Pi_2)}{|\Pi_2|} &= \frac{|V_2| + |F_2| + 1}{2|V_2| + |F_2| + |T|} \\ &= \frac{|V_2| + |T|}{2|V_2| + 2|T| - 1} \\ &> \frac{|V_2| + |T|}{2|V_2| + 2|T|} \\ &= \frac{1}{2}.\end{aligned}$$

This is a contradiction since the vertex f_1 must satisfy Inequality (1) for a 2-community. □

Observation 2.7 *The set F and and the set T do not belong to a same community.*

Proof: (by contradiction) Assume V is cut in (V_1, V_2). Also, assume $\Pi_1 = F \cup T \cup V_1 \cup V_1'$ and $\Pi_2 = V_2 \cup V_2'$. Consider a vertex $t \neq t_1$ in T. The neighbors of the vertex t in Π_1 are all vertices in V_1, plus all vertices in $T - \{t\}$. Similarly, the neighbors of the vertex t in Π_2 are only all vertices in V_2. Since (Π_1, Π_2) is a community structure, then the vertex t must satisfy Inequality (1). Therefore, we have

$$\frac{N_t(\Pi_1)}{|\Pi_1| - 1} = \frac{|V_1| + |T| - 1}{|F| + 2|V_1| + |T| - 1} \geq \frac{N_t(\Pi_2)}{|\Pi_2|} = \frac{|V_2|}{2|V_2|}.$$

By simplifying the the above inequality we arrive at

$$\frac{|V_1| + |T| - 1}{|F| + 2|V_1| + |T| - 1} \geq 1/2.$$

Now the above inequality implies that

$$2 \cdot (|V_1| + |T| - 1) \geq |F| + 2|V_1| + |T| - 1,$$

and hence

$$2 \cdot |V_1| + 2 \cdot |T| - 2 \geq |F| + 2|V_1| + |T| - 1.$$

Since $|T| = |F|$, the last inequality implies that $-2 \geq -1$, which is a contradiction. Therefore, T and F are not in a same community. □

Observation 2.8 *If (V_1, V_2) is a cut of V based on community structure (Π_1, Π_2), then $\Pi_1 = F \bigcup V_1 \bigcup V_1'$, $\Pi_2 = T \bigcup V_2 \bigcup V_2'$ and V_1 is a clique.*

Proof: (by contradiction) Assume V_1 is not a clique. Therefore, there exist a missing edge between two vertices of V_1. Suppose $v \in V_1$ is one of the end points of the mentioned missing edge. Assume x_v is the number of missing edge in V_1 with one end in v. Clearly $x_v \geq 1$. Also assume that y_v is the number of missing edge in V_2 with one end in v.

Since (Π_1, Π_2) is a community structure, the vertex v must satisfy Inequality (1), therefore we have

$$\begin{aligned}\frac{N_v(\Pi_1)}{|\Pi_1| - 1} &= \frac{(|V_1| - 1) - x_v + |F| - (x_v + y_v)}{|\Pi_1| - 1} \\ &\geq \frac{N_v(\Pi_2)}{|\Pi_2|} \\ &= \frac{|V_2| - y_v + |T|}{|\Pi_2|}. \qquad (6)\end{aligned}$$

Since $|\Pi_1| = |\Pi_2|$, therefore $|V_1| = |V_2|$. Now we simplify Inequality (6) as follows.

$$\begin{aligned}&|\Pi_2|((|V_1| - 1) - x_v + |F| - (x_v + y_v)) \\ &\quad \geq \ (|\Pi_1| - 1)(|V_2| - y_v + |T|).\end{aligned}$$

Now we substitute $|\Pi_1|$ with $|\Pi_2|$, $|F|$ with $|T|$ and $|V_1|$ with $|V_2|$ as they are equal to each other. Therefore, we get

$$\begin{aligned}&|\Pi_2|((|V_2| - 1) - x_v + |T| - (x_v + y_v)) \\ &\quad \geq \ (|\Pi_2| - 1)(|V_2| - y_v + |T|).\end{aligned}$$

After canceling equal values from the both sides of the inequality and simplifying it, then we arrive at

$$|V_2| + |T| \quad \geq \quad |\Pi_2| + 2 \cdot x_v|\Pi_2|.$$

But, the latter inequality represents a contradiction since $x_v \geq 1$ and the value of the left side of the above inequality is in fact less than $|\Pi_2|$. Therefore V_1 is a clique. □

Observation 2.9 *The size of V_1 is at least $n/2$.*

Proof: Observation 2.8 shows that V_1 is a clique. Also we know that $|\Pi_1| = |\Pi_2|$, therefore, $|V_1| = |V_2|$. Hence, the size of $|V_1| = n/2$. □

3 Some observations on the definition of community structure

As we alluded in the introduction, our aim was to investigate when can we find a large community within a community structure. Thus, we focused on the 2-COMMUNITIES problem since this ensures one community is large as it must include half of the vertices of the underling graph. However, we discovered that requesting connectivity for each community changes the problem. According to Definition 1.1, communities are not required to to be connected. That is, each community C in the community structure is not required to be connected.

Observation 3.1 *There are graphs that do not have a 2-community structure, if we demand that each community must be connected; but have a 2-community structure under Definition 1.1.*

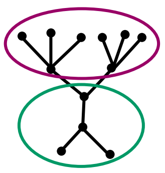


Figure 2: This graph has a 2-community structure, nodes in the purple oval is one community and nodes in the green are the other. But the purple community is disconnected. There is no 2-community structure if we require the communities to be connected.

Proof: For example, the graph in Figure 2 has a non-connected community in a 2-community structure, but it does not have a 2-community structure where both communities are connected. It does have a community structure with three communities. □

One can examine Olsen (2012)'s original proof about whether there exists a community structure where a given set S of vertices is in one community. We discovered that the proof also shows the problem to be NP-complete when we add the condition that each community shall be connected. Also, Olsen's algorithm (Olsen 2012, Theorem 2) for computing a sample community structure always returns connected communities in the structure.

We find it more natural that communities should be connected. And thus, propose that Definition 1.1 should require that each community be connected.

Olsen's algorithm (Olsen 2012, Theorem 2) also has the unfortunate circumstance that it may produce very small (and thus a large number) of communities. The algorithm uses a polynomial number of local-search improvements among certain partitions of the input graph G. Each step requires polynomial time and the climb on the values of the objective function finishes with a community structure. The output of his algorithm depends to the initial state and, for example, if we consider the graph in Figure 3 the algorithm finishes with many communities, each of size three; although the graph accepts a 2-community (with connected communities). That is, if we apply his algorithm to this graph by considering the edge $\{v, w\}$ and the edge $\{v', w'\}$ as an initial stage, then it will produce a community structure with many small communities. Therefore, this algorithm may not produce a community structure which has far more communities ($O(n)$) when the graph actually accepts a constant number. Thus, it is not a good algorithm to approximate within a constant ratio the largest community or the smallest number of communities.

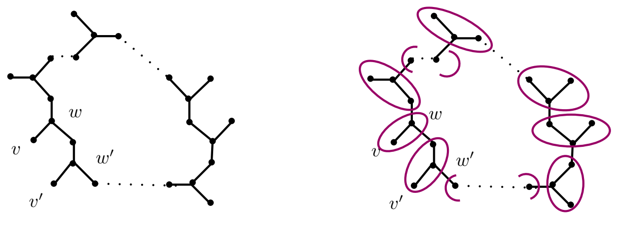


Figure 3: The left side illustrates the schema of the input graph; many S_3s (stars of three vertices, arranged in a cycle). The right side illustrates a community structure found by applying Olsen's algorithm (Olsen 2012) with an initial state consisting of the edge $\{v, w\}$ and the edge $\{v', w'\}$.

4 Classes of graphs with 2-communities

The example of Figure 3 enables us to reflect on what graphs accept 2-communities. In particular, since a community is a concept close to a cluster or a region of high density, a community structure with 2-communities must imply some low density between the communities. We can establish a relation between the notion of a cut in a graph and the notion of a 2-community structure. A cut in a graph $G=(V, E)$ is a partition (Π_1, Π_2) of vertices of G, and is called balanced if $|\Pi_1| = |\Pi_2|$. The set of edges whose end points are in different subsets of the partition is called a cut set. A min-cut is a cut with the smallest cut-set size (and can be found in polynomial time, although it might not be balanced). Figure 4 illustrates a min-cut of size two.

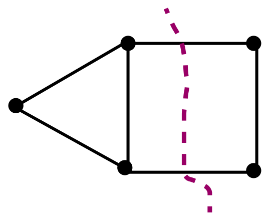


Figure 4: A min cut of size two.

We show that a balanced min-cut of a graph G constitutes a 2-community structure.

Observation 4.1 *If (Π_1, Π_2) is a cut of size two of a graph G with cut-set S, then every vertex that is not an endpoint of an edge in S satisfies Inequality (1).*

This is immediate. Every vertex $v \in \Pi_i$, that is not an endpoint of an edge in the cut-set S, has no connections to the other side. Thus, the value of $N_v(\Pi_j)$ with $j \neq i$ is zero.

Observation 4.2 *If (Π_1, Π_2) is a minimum cut of graph G with $|\Pi_1| = |\Pi_2|$, then (Π_1, Π_2) forms a 2-community structure.*

Proof: Based on Observation 4.1, we only need to show that every vertex in the cut-set satisfies Inequality (1). Assume a vertex $v \in \Pi_1$ is an endpoint of an edge in the cut-set S. The number of neighbors of vertex v in the set Π_1 is equal or greater that the number of neighbors of vertex v in the set Π_2. Otherwise, we can make an smaller cut-set by moving vertex v to the set Π_2 (contradicting the fact that the size of S is minimum among all cut-sets). Therefore,

$$\frac{N_v(\Pi_1)}{|\Pi_1| - 1} \geq \frac{N_v(\Pi_2)}{|\Pi_2|},$$

since the size of Π_1 is equal to the size of Π_2. That is the vertex v satisfies Inequality (1). □

Corollary 4.3 *Paths and cycles with even number of vertices have a 2-communities structure.*

The above corollary can be extended to the paths and cycles with odd number of vertices.

Lemma 4.4 *The* 2-COMMUNITIES *problem for graphs with maximum degree two and $|V| \geq 3$ can be solved in polynomial time.*

Proof: Let $G=(V, E)$ be a graph with maximum degree two. If G is not a connected graph, then consider any connected component Π_1 as one community and $\Pi_2 = V - \Pi_1$ as the second community. The partition (Π_1, Π_2) forms a 2-community structure since there is no edge between the two sets. Thus, based on Observation 4.1, all vertices satisfy Inequality (1).

Assume now that G is a connected graph. Since the maximum degree is at most two and the graph is connected, the graph G is a path or a cycle. We can construct a two communities as follows.

Case 1: The graph G is a path. We pick a vertex v of degree one and add all vertices in a path of length $\lceil |V|/2 \rceil$ from v into a set Π_1. The rest of vertices is placed in a set Π_2. It is not hard to see that all vertices in Π_1, and Π_2 satisfy Inequality (1). Hence (Π_1, Π_2) is a 2-community structure.

Case 2: G is a cycle. We pick a vertex v of the cycle and add all vertices in a path of length $\lceil |V|/2 \rceil$ from v into a set Π_1. Again, the rest of vertices is placed in a set Π_2. A similar argument to Case 1 shows that (Π_1, Π_2) is a 2-community structure.

□

5 Conclusion and open problems

We studied the computational complexity of the uniform k-Communities problem. We showed that this problem is NP-complete even for $k = 2$. The complexity of the problem is not known if we drop the uniformity (size of all communities are equal) condition as in the k-Communities problem. This leads to observations for detecting a community structure of size two. We also showed that the known algorithm (Olsen 2012) for finding a community structure may find a solution that is very far from an optimal solution to the 2-Communities problem. Moreover, we observed that there may exist graphs where some communities are not connected. Since requiring all communities to be connected is consistent with previous work, we suggest the definition should incorporate this requirement.

Our work here leads to several interesting open problems for finding a community structure with a specific property. We list some of them.

Problem 1: Determine the computational complexity of the uniform k-Communities problem on different classes of graphs, such as planar graphs and regular graphs.

Problem 2: Determine the computational complexity of the k-Communities problem.

Problem 3: Determine the computational complexity of finding a community structure with one community of size at least k.

Another interesting connection of the k-Communities problem seems to be a relatively similar problem in the literature which is called the Sparsest Cuts problem. A sparest cut of a graph $G = (V, E)$ is a partition $(V_1, V \setminus V_1)$ having the minimum density

$$|\text{cut-set}(V_1)|/|V_1||V \setminus V_1|$$

among all partitions in the graph, where

$$\text{cut-set}(V_1) = \{e = \{u, v\} \in E \mid u \in V_1 \text{ and } v \notin V_1\}.$$

The Sparsest Cuts problem is NP-hard; however, it can be solved in polynomial time on trees and planar triconnected graphs (Matula & Shahrokhi 1990). It is not hard to see that in paths and in cycles a sparsest cut is also a 2-community structure and vice versa. However, it would be interesting to know on what graph classes the concept of 2-community structure and of sparest-cut are identical.

References

Bazgan, C., Tuza, Z. & Vanderpooten, D. (2010), 'Satisfactory graph partition, variants, and generalizations', *European Journal of Operational Research* **206**(2), 271–280.

Boccaletti, S., Latora, V., Moreno, Y., Chavez, M. & Hwang, D. U. (2006), 'Complex networks: Structure and dynamics', *Physics Reports* **424**(4-5), 175 – 308.

Condon, A. & Karp, R. M. (2001), 'Algorithms for graph partitioning on the planted partition model', *Random Structures & Algorithms* **18**(2), 116–140.

Fortunato, S. (2010), 'Community detection in graphs', *Physics Reports* **486**(3-5), 75 – 174.

Garey, M. R. & Johnson, D. S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, USA.

Gargi, U., Lu, W., Mirrokni, S. V. & Yoon, S. (2011), Large-scale community detection on youtube for topic discovery and exploration, *in* L. A. Adamic, R. A. Baeza-Yates & S. Counts, eds, 'ICWSM11, Proceedings of the Fifth International Conference on Weblogs and Social Media, Barcelona, Catalonia, Spain, July 17-21', The AAAI Press.

Kevin J. Lang, K. J., Mahoney, M. W. & Orecchia, L. (2009), Empirical evaluation of graph partitioning using spectral embeddings and flow, *in* J. Vahrenhold, ed., 'SEA09, Proceedings of 8th International Symposium on Experimental Algorithms, Dortmund, Germany, June 4-6', Vol. 5526 of *Lecture Notes in Computer Science*, Springer, pp. 197–208.

Lancichinetti, A., Kivela, M., Saramaki, J. & Fortunato, S. (2010), 'Characterizing the community structure of complex networks', *CoRR* **abs/1005.4376**.

Matula, D. W. & Shahrokhi, F. (1990), 'Sparsest cuts and bottlenecks in graphs', *Discrete Applied Mathematics* **27**(1-2), 113–123.

Olsen, M. (2012), On defining and computing communities, *in* J. Mestre, ed., 'Computing: The Australasian Theory Symposium (CATS 2012)', Vol. 128 of *CRPIT*, ACS, Melbourne, Australia, pp. 97–102.

Patkar, S. B. & Narayanan, H. (2003), An efficient practical heuristic for good ratio-cut partitioning, *in* 'Proceedings of 16th International Conference on VLSI Design', pp. 64 – 69.

Workflow Resource Pattern Modelling and Visualization

Hanwen Guo, Ross Brown, Rune Rasmussen
Information Systems School, Science and Engineering Faculty
Queensland University of Technology
GPO 2343, Brisbane 4001, QLD

hanwen.guo@student.qut.edu.au, r.brown@qut.edu.au, r.rasmussen@qut.edu.au

Abstract
Workflow patterns have been recognized as the theoretical basis to modeling recurring problems in workflow systems. A form of workflow patterns, known as the resource patterns, characterise the behaviour of resources in workflow systems. Despite the fact that many resource patterns have been discovered, people still preclude them from many workflow system implementations. One of reasons could be obscurity in the behaviour of and interaction between resources and a workflow management system. Thus, we provide a modelling and visualization approach for the resource patterns, enabling a resource behaviour modeller to intuitively see the specific resource patterns involved in the lifecycle of a workitem. We believe this research can be extended to benefit not only workflow modelling, but also other applications, such as model validation, human resource behaviour modelling, and workflow model visualization.

Keywords: Workflow Resource Patterns, Modelling, Visualization.

1 Introduction
Presently, people often use workflow modeling languages to describe their business environment (van der Aalst and Hofstede 2005). Conventionally, a workflow system can be understood from the control, resource and data perspective (van der Aalst, Hee et al. 1994). The resource perspective represents responsibilities, behaviour and the organizational structure of workflow resources within a business environment.

Human resource behaviour is one important component in the resource perspective, since it can affect the efficiency of an organization (Moore 2002; zur Muehlen 2004). Presently, people have already indentified patterns to describe the human resource behaviours, and have used these patterns to solve human resource behaviour related problems (Russell, van der Aalst et al. 2005).

Russell et al. have defined a group of resource patterns (Russell, van der Aalst et al. 2005), describing various human resource task allocation, execution manner and interaction mechanisms between human resources.

A modelling approach that can represent this resource-task logic or resource patterns is quite necessary. This is because such a modelling approach can provide people with a clear view about the relationship between resources and tasks.

In the modelling domain, there are various modelling approaches, such as Petri nets (Pesic and van der Aalst 2007) and BPEL4PEOPLE (an extension of BPEL) (Russell and van der Aalst 2008) that can be used in representing such resource logic. These modelling approaches usually have visual representations in 2D conceptual shapes, such as circles, arrows and rectangles.

Indeed, these visual representations can impede the communication between business analysts and stakeholders (Shannon and Weaver 1963; Sadiq, Indulska et al. 2007; Moody 2009). This is because stakeholders usually don't hold necessary knowledge about modelling grammars (V. Khatri, I. Vessey et al. 2006), and some empirical studies show that such a simple representation can reduce the cognitive load required for understanding in the human brain, especially for naïve stakeholders who are not quite familiar with specialist visual grammars (J. Parsons and Cole 2005; Burton-Jones, Wand et al. 2009).

According to relevant research, communication between business analysts and stakeholder has been recognized as a key critical success factor in success of business process management projects (Nah, Lau et al. 2001; Trkman 2010). This implies that an ineffective communication approach may result in the failure of a process modeling and improvement exercise, as a workflow management solution may be implemented sub optimally, resulting in an inefficient organization.

Herein, we can say that a well established resource behaviour modelling approach with an easily understood visual representation can not only enable managers to understand the relationships between workflow resources and tasks in the workflow system and facilitate workflow management system development, but it can also benefit communication between business analysts and stakeholders, improving implementation outcomes.

Therefore, in this paper we propose a resource behaviour oriented modelling and visualization approach for resource patterns (Russell, van der Aalst et al. 2005). The modelling approach is based on an automated planning technique, known an Hierarchical Task Network (HTN) (Erol, Hendler et al. 1994). Such a modelling approach supports a decomposition mechanism, whereby some simple resource patterns can be automatically assembled up to represent some complex resource patterns, enabling a "many workitem to many resources" relationship to be modelled. We employ the virtual world as our visualization approach. Virtual worlds are a synthetic replication of the real world (Burdea and Coiffet 2003). With a mapping mechanism between HTN modelling results and a virtual world, the modelling results can be represented as an intuitive, easy to understand animation. Such an intuitive visualization enables people to recall and cognate about conceptual and concrete content in the business process system, facilitating the communication process with regards to analysing, modelling and validating human resource behaviours, see a simple example in Figure 1, below.

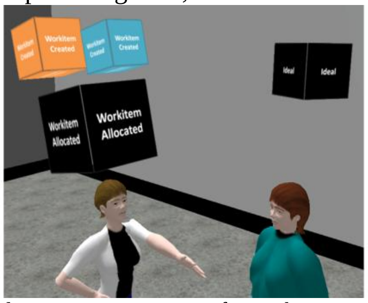


Figure 1 visualization representation of a push resource pattern. The cubes above the avatars' heads indicate the current states of workitems, and the allocation task is illustrated with the body movement of the avatar, indicating a state transition of a workitem.

This paper is organized as follows: Section 2 discusses related work within the control, resource and data perspectives. Section 3 discusses our HTN based resource pattern modelling and visualization approach. Section 4 utilizes a health care scenario to demonstrate the modelling and visualization ability of our approach with multiple resources and workitems. Finally, Section 5 discusses our further work.

2 Related Work

2.1 Control Perspective

Regarding the control perspective, some researchers (Lu, Bernstein et al. 2006; Rasmussen and Brown 2012) model tasks in the workflow system as an operator with the form $op = <p, q, v>$. Item v is a list of parameters, while items p and q are two assertions indicating the execution of operator op must satisfy p (pre-conditions) and can establish the post-conditions q, invoking the state transition from the state containing p to the state containing q in the workflow system. In their approach, search algorithms are used to continually select a suitable operator whose pre-conditions are compatible with the current state. With several iterations, the search algorithms will find a serial list of sequenced operators, and the execution of operators can lead the workflow system to transition from an initial state to a goal state. Modellers can use these sequenced tasks to represent the workflow model.

2.2 Data Perspective

A similar approach has been used within the data perspective. Some researchers (Nigam and Caswell 2003; Wang and Kumar 2005; Bhattacharya, Gerede et al. 2007) utilize business artefacts or documents in the workflow system to automatically construct workflow models. These document centric approaches recognize that tasks in the workflow system are services requiring input data from the task executors which generate related output data. These inputs and outputs are regarded as clues of task execution records, being used to derive the temporal and logic ordering of tasks. For example, an online shopping activity may contain an order list, a confirmation letter, an invoice and a confirmation slip of reception. Based on their occurrence, the ordering logics of item selection, payment and delivery can be derived. With these relations, the occurrence ordering and dependency of these tasks can be derived, and linkages between these tasks can be used to represent a workflow model.

2.3 Resource Perspective

In a resource perspective, most of the research work focuses on resource modelling and resource utilization issues. Zur Muehlen (zur Muehlen 1999) states that a resource model usually contains two parts: assignment policies and resource details. He points out that most modelling approaches do not consider that the resource details should facilitate the assignment policies on the one hand, and ignore the importance of non-human resources in the workflow system on the other hand. Therefore, he proposes a generic meta-model that can not only represent any resources in the workflow activity, but also facilitates the assignment policy implementation and execution.

Pesic and van der Aalst (Pesic and van der Aalst 2007) focus on task distribution issues in the workflow system. They proposed a basic model which contains the work distribution and work list model. These two modules can interact with each other to simulate the process of workitem distribution, and the internal mechanisms of these two modules are modelled using Petri nets.

It can be concluded that these works (Nigam and Caswell 2003; Wang and Kumar 2005; Lu, Bernstein et al. 2006; Bhattacharya, Gerede et al. 2007; Rasmussen and Brown 2012) focus on the automated model construction mechanism in the workflow system from different perspectives. Zur Muehlen (zur Muehlen 1999) deals with the static structural description of resource properties, while Pesic and van der Aalst (Pesic and van der Aalst 2007) describe the dynamics aspect of work distribution.

In this paper, we intend to automate the execution of a single workitem, rather than the entire workflow model construction approaches which have been proposed in the papers (Nigam and Caswell 2003; Wang and Kumar 2005; Lu, Bernstein et al. 2006; Bhattacharya, Gerede et al. 2007; Rasmussen and Brown 2012). Similar with the work done by Pesic and van der Aalst (Pesic and van der Aalst 2007), we focus on the dynamics aspects of a workitem, describing resource interactions around an allocated workitem. The difference between our approach and others (Pesic and van der Aalst 2007) is that we employ an HTN to automatically model interaction mechanisms involved in a workitem execution. The benefit of utilizing an HTN as a modelling approach is that an HTN can automatically generate rational interactions between human resources, if pre-conditions and post-conditions of each interaction are provided. In addition, we address the visualization issues that have not been addressed in these papers (zur Muehlen 1999; Nigam and Caswell 2003; Wang and Kumar 2005; Lu, Bernstein et al. 2006; Bhattacharya, Gerede et al. 2007; Pesic and van der Aalst 2007). These papers only consider modelling aspects, rather than communication aspects of the modelling approach. Their graphical representations are in a highly abstracted 2D diagram, which will likely puzzle naïve stakeholders who have less professional knowledge about specific modelling languages. In contrast, we provide a "hands on" representation manner for model readers, by defining a mapping system between modelling results and a virtual world. Such a manner is expected to be intuitive and easily understood by stakeholders.

3 Resource Pattern Modelling and Visualization

We are going to describe an HTN based modelling and visualization approach for resource patterns. We briefly introduce resource patterns in Section 3.1, and discuses how to use the Hierarchical Task Network (HTN) to model resource patterns in Section 3.2. We then continue with a brief introduction of virtual worlds in Section 3.3, followed by the mapping mechanism between the HTN modelling results and a virtual world. Lastly, we discuss implementation issues with this approach in Section 3.4.

3.1 Resource Pattern as State-transitions

Resource patterns (Russell, van der Aalst et al. 2005) can be categorized into seven groups, namely: Creation, Push, Pull, Detour, Auto-start, Visibility and Multiple Pattern, respectively, see brief description in Table 1.

Pattern Category	*Brief Description*
Creation Pattern	Workitem creation mechanism in a workflow management system
Push Pattern	Workitem allocation mechanism in workflow management system
Pull Pattern	Workitem acquisition mechanism in workflow management system
Detour Pattern	How a workitem is related to another resource
Auto-start Pattern	How one workitem can trigger the execution of other workitems
Visibility Pattern	Visibility of committed workitems with respect to other resources
Multiple resource Patterns	Coordination mechanism between multiple resource execution

Table 1 The brief description of pattern category.

These patterns can characterize the behaviour of workflow management systems and workflow resources in the lifecycle of a workitem, in Russell et al. these patterns belong to two relationship groups, viz., “single workitem to single resource” and “many workitems to many resources”. Some evidence (Pesic and van der Aalst 2007) utilizing Petri-net to model resource patterns shows that resource patterns are state-transitions. Here, we also consider resource patterns as state-transitions, but we will use different mechanism to model them.

The “single workitem to single resource” relationship involves the Creation Pattern, Push Pattern, Pull Pattern, Detour Pattern, and Auto-start Pattern. The selected visualizations of these patterns are available in Figure 2, and readers who are interest in resource pattern visualizations are suggested to read their original paper (Russell, Hofstede et al. 2004). Within these five resource pattern categories, the lifecycle of a workitem begins at the created state and ends at a failed or completed state. For example, the push patterns can be represented via three state transitions, which are state transitions from created state to offered state to single resource, created state to allocated state to a single resource, and created state to offered to multiple resources (see the three red dash lines with arrows in Figure 2). It can be said that the essence of resource patterns are actual state transitions in the lifecycle of a workitem. The life cycle of a workitem can be viewed as a sequence of resource patterns, transiting a workitem from the created state to completed or failed state (see the dash rectangle in Figure 2, where a creation pattern, push pattern and pull pattern occur, consequently). Therefore, patterns in the “single workitem to single resource” relationship can be modelled as state-transitions.

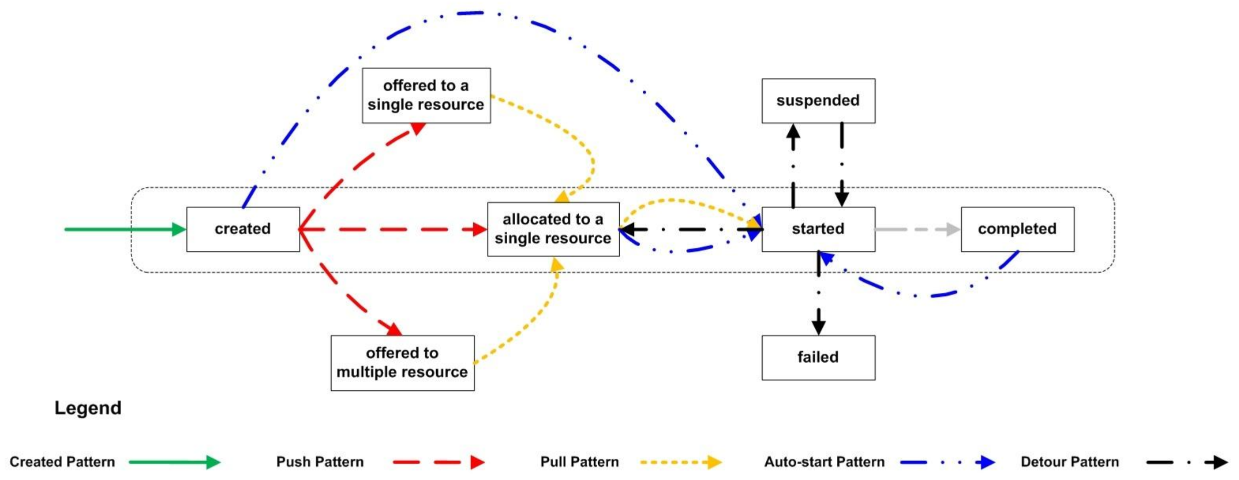


Figure 2 is a visual representation of the resource pattern. The rectangles are used for representing the states, and arrows are used for the transitions. Some of the resource patterns have been omitted for clarity.

Similar to patterns in the “single workitem to single resource”, we consider resource patterns in the “many workitems to many resources” relationship can be modelled as state-transitions. Russel et al. (Russell, Hofstede et al. 2004) discuses two multiple resource patterns, Additional Resources Pattern (Pattern R-AR) and Simultaneous Execution Pattern (Pattern R-SE). The Pattern R-AR describes the behaviour of a resource requiring assistance from additional resources when this resource is dealing with a workitem, while Pattern R-SE describes the behaviour of several resources processing the same workitem at the same.

We utilize these two patterns placing them into Scenario 1, and then use this scenario to demonstrate the applicability of using a state transition mechanism to model the Multiple Resource Pattern.

Scenario 1 *There is a workitem W_o. It is created and allocated to a resource R_a by a workflow engine. Resource R_a started the* *workitem, and then divided Wo as three sub-workitems W_a, W_b and W_c to himself and two subordinates R_a, R_b and R_c, respectively. The execution of sub-workitem W_a is dependent on the results of W_b and W_c. R_a allocates the W_b to R_b without negotiation, R_b has to executed W_b immediately. R_a allocates the W_c to R_c with negotiation, R_c can select an appropriate time to execute it. As these two sub-workitems have been completed and reported back to R_a, R_a can start to execute the W_a. When W_a is finished, the original workitem W_o can be accomplished and checked back to workflow engine.*

Scenario 1 describes four nested workitems. Their state transitions are different from flat ones we discussed previously, that is, these state transitions are in a hierarchical structure.

At a very high level, the lifecycle of this workitem W_o can be understood as two states (initial and final) with an execution phase. The execution phase also involves several states. That is, workitem W_o is created and allocated to the resource R_a, and then it is the resource R_a and the other two additional resources R_b and R_c that jointly complete it. The state transition transiting the workitem W_c from created state to the completed state can be further investigated. According to the description, workitem W_o can be divided as three sub-workitems W_a, W_b and W_c. The life cycles of these sub-workitems consist of different state-transitions or resource patterns. For example, the life cycle of workitem W_c involves two state-transitions or two resource patterns, that is, a creation pattern transiting workitem W_c from created state to started state, and an auto-state pattern transiting workitem W_c from started state to completed state. This will be true when considering lifecycles of W_a and W_b. The execution of W_a and W_b can be started simultaneously. In particular, the execution of W_a and W_b can be the Pattern R-SE, if we recognize R_a and R_b are the same resource.

We illustrate these state-transitions in a top to bottom view, see Figure 4. It can say that a decomposition mechanism enables us to analyse the state-transitions of nested workitems. In other words, patterns in the Multiple Resource Pattern category can be represented as state-transitions in a hierarchical structure.

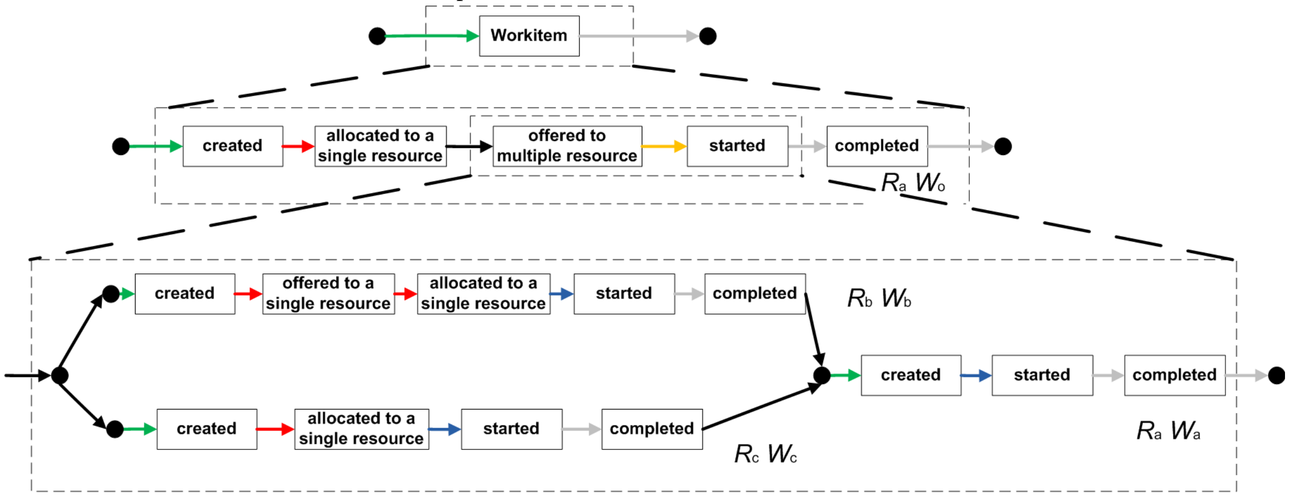

Workitem
created
allocated to a single resource
offered to multiple resource
started
completed
Ra Wo
created
offered to a single resource
allocated to a single resource
started
completed
Rb Wb
created
allocated to a single resource
started
completed
Rc Wc
created
started
completed
Ra Wa

Figure 3 a top-bottom view of the state transitions in the workitem W_o.

3.2 HTN Modelling Approach

Conventionally, it is believed that Erol et al. (Erol, Hendler et al. 1994) first provided a clear theoretical framework for an HTN. There are two types of tasks in their HTN framework, namely complex and primitive tasks. The execution of a primitive task or a complex task can lead the system transit from a state to another, but the execution mechanism of these two types of tasks are different. In practice, the executions ordering of primitive and complex task are constrained by a task network, and a decomposition mechanism may involve substantial computational effort. We believe these two types of tasks, task network and decomposition mechanism can be used to represent resource patterns. Table 2 indicates a mapping mechanism between resource patterns and an HTN.

Resource Pattern	***HTN framework***
"single resource to single workitem" resource patterns	Primitive task
"multiple resources to multiple workitems" resource patterns	Complex task
Workitem life cycle	Task network

Table 2 mapping mechanism between Resource Pattern and HTN framework.

In the following we select some necessary concepts for introduction. Readers who are interest in the full syntax and semantics of HTNs are suggested to read the original paper (Erol, Hendler et al. 1994).

A primitive task is a task that can be directly solved by the task execution. It can be modelled with the form $op = <p, q, v>$. The satisfaction of pre-conditions p enables operator execution, and the operator execution enables the establishment of post-conditions q. This means the operator execution enables a state transition from the state containing pre-conditions p to the state containing post-conditions q.

A complex task can be recognized as the aggregation of primitive tasks. Such a complex task cannot be solved by task execution directly, but by requiring decomposition before execution. That is, using a set of primitive tasks to represent this complex task, the execution of the complex task is equivalent to execution of all primitive tasks. The state transition triggered by the execution of a complex task is equivalent to the aggregation of state transitions of selected primitive tasks. For example, a complex task *ct* is a complex task, being composited by three primitive tasks pt_1, pt_2 and pt_3. The execution of the *ct* is the execution of pt_1, pt_2 and pt_3. The pre-conditions of the firstly executed primitive task or primitive tasks should not violate the state s_i before the execution of *ct*, and the state s_t after the execution of ct is dependent upon the post-conditions of finally executed primitives.

In short, the complex task needs to be resolved by a task network. The task network is an array where some states and task sets are alternatively placed. It can be modelled in a form $[tn\_name, [(t_1, tl_1) \dots \dots (t_n, tl_n)], \mu]$, where tn_name is the name of this task network, t_i and tl_i are the name and label of a task, μ is a formula defining the partial ordering of tasks and states. Usually, a task network has two functionalities, decomposing a complex task and defining logical ordering of tasks.

A complex task can be modelled in a form $me = < ct, tn, mp, mq >$, where ct is the name of the complex task, tn is the corresponding task network, and mp and mq are high-level pre- and post-conditions of the sequenced primitive tasks in the task network tn, respectively. That is, a method $me = < ct, tn, mp, mq >$ can be selected for the complex task ct, if and only if the current and target state contains the mp and mq, respectively.

3.3 Modelling Resource Patterns with HTN

We will enumerate a number of HTN modelled resource patterns to prove the possibility of using HTN as the modelling approach for resource patterns in this section.

Patterns in the Creation Pattern, Push Pattern and Pull Pattern are relatively simple. The common feature of these patterns is that they can transit one state to another without further decomposition. For example, pull patterns characterize the transition from the allocated or offered state to started state, characterising the proactive behaviour of resources selecting a suitable workitem to execute. Thus, we provide basic HTN modelling results for them, see Table 3.

Task Name	***Task Network Details***
basic_task	$[basic\_task\_network, [(t, tl)], \mu]$ $\mu = \{(S_I \prec tl \prec S_T)\}$
REMARKS	
The *basic_task* is a primitive task that can be used to represent the resource patterns with two states and one transition. The primitive task *t* in it can be implemented as requirements mentioned in the Creation Pattern, Push Pattern, and Pull Pattern.	

Table 3 the basic task and task network that can be used to model patterns in Creation Pattern, Push Pattern, Pull Pattern.

Most patterns, for example, the Detour Pattern and Auto-start Pattern, transit a workitem from one state to another. They can be modelled by the basic task network in Table 3. However, there are some patterns, in these two categories, requiring a decomposition mechanism. We have to model these patterns individually. These patterns are the Stateful Reallocation Pattern (Pattern R-PR) and Stateless Reallocation Pattern P-UR (Pattern P-UR) in the Detour Pattern group, Piled Execution (Pattern P-PR) in the Auto-start Pattern group, as well as the Simultaneous Execution Pattern (Pattern R-SE) and the Additional Resources Pattern (Pattern R-AR) in the Multiple Resources Pattern. These patterns should be modelled by complex tasks and decomposition mechanisms.

Pattern R-PR (Stateful Reallocation) and Pattern P-UR (Stateless Reallocation) are different in functionality. Pattern R-PR requires the state information of a workitem being kept when this workitem is reallocated to another resource, while Pattern P-UR doesn't have such a rule. However, their modelling result can be illustrated in a similar manner. Tasking Pattern R-PR (Stateful Reallocation) as an example, at the top level, a task *reall_task* is needed to transit workitem from the started state back to the allocated state. A task network for this task contains a primitive task *exe_task* and a complex task *next_task*. The primitive task *exe_task* enables the execution situation to be recorded, and the complex task *next_task* can be interpreted by a task network as *next_step_a* or *next_step_b*, see details in Table 4. In particular, modelling results in Table 4 can be used as a reference to model the Pattern P-UR (Stateless Reallocation) by implementing *exe_task* as a function that doesn't record the execution information.

Task Name	***Task Network Details***
reall_task	$[reall\_with\_state, [(exe\_task, tl_e), (next\_task, tl_n)], \mu]$ $\mu = \{(S_S \prec tl_e \prec S), (S \prec tl_n \prec S_A)\}$
next_task	$[next\_step\_a, [(exe\_task, tl_e), (next\_step, tl_n)], \mu]$ $\mu = \{(S_I \prec tl_e \prec S), (S \prec tl_n \prec S_A)\}$
next_task	$[next\_step\_b, [(all\_task, tl_a)], \mu]$ $\mu = \{(S \prec tl_a \prec S_A)\}$
REMARKS	
reall_task is a task network that can reallocate workitems from one one resource to another, involving one primitive task *exe_task* and a complex *next_task*. The *exe_task* is an executable function that can records the execution state information, while *next_step* can be interoperated by two different task networks, *next_step_a* and *next_step_b*. the *next_step_a* enables a resource to further execute the workitem, *next_step_b* enables a resource to reallocate the workitem to another resource.	

Table 4 the modelling result of Pattern R-PR (Stateful Reallocation).

Pattern P-PE (Piled Execution) in the Auto-start Pattern is a pattern that enables a resource to execute workitems in batch. HTN modelling results of Pattern P-PR is available in Table 5. In this modelling result, there are two tasks. The task *pile_all* enables a resource to recognize the incoming tasks, while the task *pile_cpl* enables the resource to start processing and complete these allocated workitem.

Task Name	***Task Network Details***
pile_task	$[pile\_network, [(pile\_all, tl_a), (pile\_cpl, tl_c)], \mu]$ $\mu = \{(S_I \prec tl_a \prec S), (S \prec tl_c \prec S_C)\}$
pile_all	$[pile\_all\_network, [(t_1, tl_1), (t_2, tl_2) \dots \dots (t_m, tl_m)], \mu]$ $\mu = \{(S_I \prec tl_i), (tl_i \prec S_S)\}$, where $i = 1,2 \cdots \cdots m$
pile_cpl	$[pile\_cpl\_network, [(t_{m+1}, tl_{m+1}) \dots \dots (t_{2m}, tl_{2m})], \mu]$ $\mu = \{(S_S \prec tl_i), (tl_i \prec S_C)\}$, where $i = 1,2 \cdots \cdots m$
REMARKS	
plie_task being interoperated by *pile_network* is the task that enables a resource to execute workitems in a batch. Such a task can be divided into two parts, namely *pile_all* and *pile_cpl*. The task *pile_all* enables all involved workitems transit from some state S_I to the started state S_S in a partial order, while *pile_cpl* enables all involved workitems in the started state S_S to the state S_C where all workitems are completed.	

Table 5 the modelling result of Pattern R-PE (Piled Execution).

Pattern R-SE (Simultaneous Execution) and Pattern R-AR (Additional Resources) are two patterns in the Multiple

Resource Pattern, characterizing the “many workitems to many resources” relationship.

Pattern R-SE (Simultaneous Execution) requires that one single resource can manipulate multiple workitems in a period. We believe the Pattern P-PE (Piled Execution) is a particular type of Pattern R-SE. This is because those two patterns require that one single resource can deal with multiple workitems at the same time. The difference is that Pattern P-RP constrains a resource to complete workitems in batch, while Pattern R-SE doesn’t have such a strong constraint. Thus, the modelling results of Pattern R-SE (Simultaneous Execution), as a simple version of Pattern R-PE, is available in Table 6 .

Task Name	***Task Network Details***
sim_task	$[sim\_network, [(t_1, tl_1), (t_2, tl_2) \dots\dots (t_m, tl_m)], \mu]$ $\mu = \{(S_I \prec tl_i \prec S_S)\}$, where $i = 1,2 \cdots\cdots m$
REMARKS	
sim_task is a complex task involving many workitems. The μ doesn’t put execution ordering in a strict manner. It puts every workitem t_i in a context that every task should be executed between states S_I and S_S.	

Table 6 the modelling result of Pattern R-SE (Simultaneous Execution).

Pattern-AR (Additional Resource) characterizes that one resource can request additional resources to assist in the process of a workitem. One possible solution is to divide the workitem into several sub workitems, and allocate these sub workitems to additional resources. Then, these additional resources can start to process sub workitems, individually. As all the sub workitems have been completed, then the original workitem is completed (van der Aalst and Kumar 2001). We model this pattern in Table 7 .

Task Name	***Task Network Details***
add_res_task	$[add\_network, [(div\_and\_dis, tl_d), (cpl, tl_c)], \mu]$ $\mu = \{(S_I \prec tl_d \prec S_S), (S_S \prec tl_c \prec S_C)\}$ where $i = 1,2 \cdots\cdots m$
div_and_dis_task	$[dd\_network, [(t_1, tl_1), (t_2, tl_2) \dots\dots (t_m, tl_m)], \mu]$ $\mu = \{(S_I \prec tl_i \prec S_S)\}$, where $i = 1,2 \cdots\cdots m$
REMARKS	
add_res_task is the task being interpreted by *add_network* containing complex task *div_and_dis_task* and primitive task *cpl*. The complex task *div_and_dis_task* can be used to decompose a workitem into a set of sub-workitems (task) t_i, and these sub-workitems should be completed before final *completion*, see the constrains $(tl_d \prec S_S \prec tl_c)$. The decomposition details about t_i are not shown in this modelling results, but can take the *basic_task_network* in Table 3 as reference.	

Table 7 the modelling result of Pattern R-AR (Additional Resource).

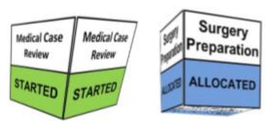
4 Resource Pattern Visualization in the Virtual World

A virtual world is a network-based, computer synthesized dynamic environment, where participants can observe and interact with computer-generated objects (Burdea and Coiffet 2003). The modelling results of the resource patterns previously detailed, basically involves two entities, state and transition. A state means a unique configuration of the system, indicating the static aspects of a workitem. A transition means a process where a system moves from one state to another, describing the dynamics aspects of the workitem. We believe that appropriate geometry and an animated avatar, as features of a virtual world, can be used to satisfied static and dynamic workitem aspect visualization.

Geometry in a virtual world can be shaped and decorated with different textures to represent different material. These representations are an integration of visual singles (structure and spectrum). According to cognitive theory, the working memory in human can distinguish the features of visual singles (Lohse 1997). In the context of our research, we can use these visual features to represent the different states of a workitem. For example, the green colour can suggest a workitem is in started state, while a red arrow can suggest a workitem is being handed over from one human resource to another.

Avatars, in general, are a representation of the self in a given environment, enabling its host to sense and react on events happening in the environment, and to change the given environment (Castronova 2003). In the context of a 3D virtual world, an avatar can be a humanoid 3D representation, driven by a virtual world participant (a human or an intelligent software agent). It can be used to replicate the behaviour of human resource in the workflow system. For example, the hand shaking of two avatars can be used to represent reallocation of a workitem, the keyboard tapping of an avatar can be used to represent a human resource is dealing with a workitem. Such an animated behaviour can intuitively suggest the transitions happening in the system (Tverskyand and Morrison 2002).

With the discussion above, if a resource pattern can be modelled and mapped into a virtual world appropriately, participants such as business analysts and stakeholders can observe resource patterns in an intuitive manner. We already demonstrated the modelling applicability of an HTN for resource patterns in Section 3, a mapping mechanism between modelled resource patterns and virtual world features, geometry and avatar, will be necessary for us to establish this visualization system. Therefore, we demonstrate the mapping between resource patterns and virtual world features in Table 8

Resource Pattern Element	**Visual Representation**	**Description**
State	 Medical Case Review STARTED Medical Case Review STARTED Surgery Preparation ALLOCATED	A cube with different textures can be used to represent the state of the workitem. The texture with words and colourful background can be used to indicate the name and statues of the workitem, respectively. By attaching the texture on the cube, it enables people to observe the state information of the workitem from different angles, and different colours can make it easy to distinguish in different states.

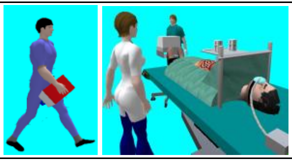
Transition		The animation and postures of avatars are used to represent the dynamic aspect of the workitem. The avatar taking a blood pack may indicate the blood transition is in progress.

Table 8 the visual representation of states and transitions in the resource patterns.

5 Detailed Medical Example

The previous section has established a mapping mechanism between resource patterns and a virtual world. Here, we use a medical example to illustrate our modelling and visualization approach in detail. Section 5.1 introduces the background of workflow applications in the medical domain, indicating a potential visualization needs in this field. Section 5.2 uses a fabricated scenario to demonstrate applicability of virtual world visualization.

5.1 Background

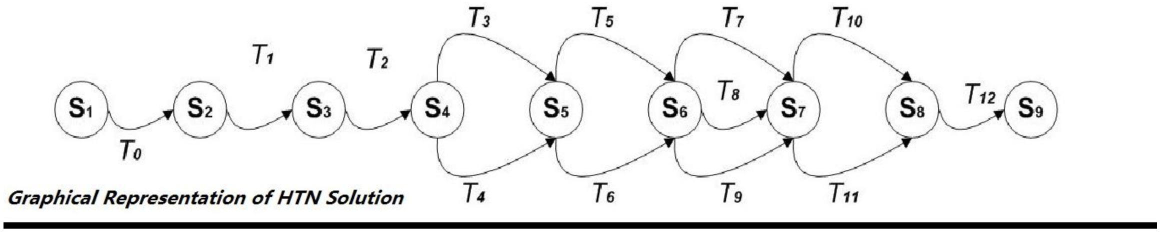
Treatment processes in the medical domain have been investigate by many workflow experts, and these experts (Mans, M.H.Schonenberg et al. 2008) recognized treatment processes as "careflows", which are a type of ad-hoc workflows. Workitems involved in such workflows require resources to be highly participative, interactive and collaborative, therefore it is evident that, numerous resource patterns occur in the lifecycle of one workitem in such scenarios. A resource modelling component would be useful to clarify the participation, interaction and collaboration mechanisms in these careflows (Richard and Manfred 2007). Animation has a strong ability to explain the dynamics aspect of a system (Tverskyand and Morrison 2002), therefore, a visualization component will be necessary to reduce the cognitive overhead in understanding underlying participation, interaction and collaboration mechanisms.

5.2 Resource Pattern Visualization Example

We adapt the Scenario 1 in Section 3.1 into a medical context for demonstrating modelling and visualization results. In an adapted scenario, four resources are involved in accomplishing a complex workitem, containing three primitive tasks. The example involves a creation pattern, pull pattern, push pattern, detour and an auto-start pattern, with the example itself as a multiple resource pattern representing the relationship "many workitems to many resources", see the details below:

> ***Scenario 2.*** *The trauma team lead R_1 is executing a workitem W_X, the "Medical Case Review". At that time, the workflow engine creates a workitem called "Surgery Preparation" W_0 for the resource R_1. Thus, resource R_1 reallocates workitem W_X to another R_X with current execution information of the workitem W_X. After the reallocation, resource R_1 accepts this workitem W_0 and starts to divide W_0 as three sub-workitems, "Retrieve Patient Information" workitem W_1, "Aesthetic Preparation" workitem W_2 and "Instrument Preparation" workitem W_3, which are going to be allocated to herself, and surgery assistants R_2 and R_3, respectively. R_2 should passively wait for the allocation, while R_3 can actively commit to the workitem. After resources R_2 and R_3 confirm the sub-workitems, resources R_1, R_2 and R_3 can execute these three workitems. The execution of sub-workitem W_1 should be started immediately after the accomplishment of W_2 and W_3. When W_1 is finished, the original workitem W_0 can be concluded and checked back to the workflow engine."*

The scenario above implicitly contains several resource patterns (Russell, Hofstede et al. 2004). For example, the detour pattern (Pattern R-PR,) between Resource R_1 and R_X, as they are dealing with the workitem W_X. The pull pattern (Pattern R-SA, see) between Resource R_1 and R_3, as resource R_3 is actively requiring the commitment of sub-workitem W_3. The lifecycle of workitem W_0 can be modelled by an HTN. We show the final modelling results in Figure 4, where the relationships between the tasks are represented, but we omit the task network construction and execution processes. With the mapping mechanism we defined in Section 4, the visualization results can be shown in Figure 4.

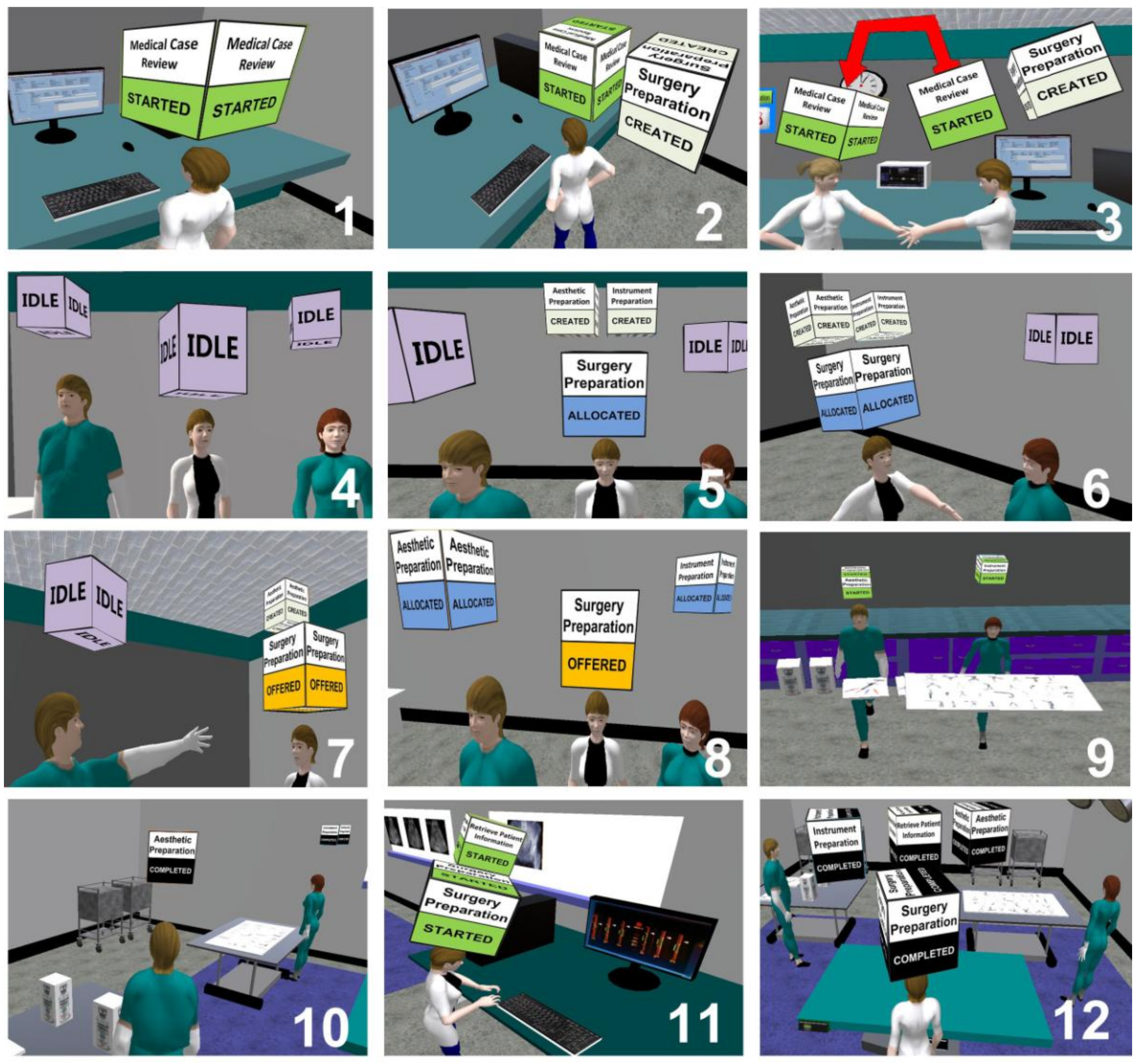


Annotation:

$S_1 = \{ W_X\_is\_started \}$
$S_2 = \{ W_X\_is\_started,\ W_0\_is\_created \}$
$S_3 = \{ W_X\_is\_reallocated\_to\_R_X,\ W_0\_is\_created \}$
$S_4 = \{ W_0\_is\_allocated\_to\_R_1 \}$
$S_5 = \{ W_0\_is\_allocated\_to\_R_1,\ W_2\_is\_created,\ W_3\_is\_created \}$
$S_6 = \{ W_0\_is\_offered\_to\_multiple\_resource,\ W_2\_is\_allocated\_to\_R_2,\ W_3\_is\_offered\_to\_R_3 \}$
$S_7 = \{ W_0\_is\_started\_by\_R_2\_R_3,\ W_2\_is\_started\_by\_R_2,\ W_3\_is\_started\_by\_R_3 \}$
$S_8 = \{ W_2\_is\_completed\_by\_R_2,\ W_3\_is\_completed\_by\_R_3,\ W_1\_is\_created\_and\_is\_started\_by\_R_1 \}$
$S_9 = \{ W_1\_is\_completed,\ W_0\_is\_completed \}$

T_0 : create_W_0
T_1 : reallocate_W_X_to_R_X
T_2 : allocate_W_0_to_R_0
T_3 : create_W_2
T_4 : create_W_3
T_5 : allocate_W_2_to_R_2
T_6 : offer_W_3_to_R_3
T_7 : start_W_2_with_R_2
T_8 : start_W_0_with_R_2_R_3
T_9 : start_W_3_with_R_3
T_{10} : execute_W_2_with_R_2
T_{11} : execute_W_3_with_R_3
T_{12} : execute_W_1_with_R_1

Figure 4 The HTN solution of surgery preparation, representing the lifecycle of the workitem "surgery". The circle is the representation of state, while the curved arrows is the representation of transition.



Picture ID	*HTN modelling result*	*Picture ID*	*HTN modelling result*	*Picture ID*	*HTN modelling result*
1	S_1	*2*	S_2	*3*	T_1
4	S_4	*5*	S_5	*6*	T_5
7	T_6	*8*	S_6	*9*	T_{10}, T_{11}
10	S_8	*11*	S_8 , T_{12}	*12*	S_9

Pattern Category	*Pattern Name*	*Picture ID*	*Remark*
Creation Pattern	Pattern R-DA	4,5	A resource is creating three sub-workitems, and going to allocate these sub-workitems.
Push Pattern	Pattern R-DBOS	6	A resource is trying to allocate a workitem to her subordinates.
Pull Pattern	Pattern R-SA	7	A resource is actively asking for workitem commitment.
Auto-Start Pattern	Pattern R-CC	10,11	As two resources completed their workitems in picture 10, a resource can start a workitem in picture 11.
Detour Pattern	Pattern R-PR	3	One resource is reallocating her workitem to another resource, the other resource can continue her work
Multiple Resource Pattern	Pattern R-AR	5-12	A resource needs two extra resources to assist her to accomplish surgery preparation.

Figure 5 The resource behaviour visualization in 9 pictures. These 9 pictures describe the responsibilities of different resources in the workitem surgery preparation. Such a workitem is divided into three sub-workitems that are allocated to three different resources. The combination of these 9 pictures reflects the relationship of many workitems to many resources, that is, the multiple resource pattern.

In Figure 5, we visualize six categories of resource patterns, except the Visibility Pattern. According to the statement in paper (Russell, van der Aalst et al. 2005), the Visibility Pattern mainly deals with relationship among the availability and commitment of workitems and attributes of resources. This is a problem of authorization rather than state transition. Thus, we don't visualize patterns belonging to this category. Despite the fact the visualization case does not involve visualized visibility patterns, we can still visualize them by modifying the property of cube hovering above heads of avatars. The cube is the indication of the state of a workitem being processed by an avatar. The solid, semi-transparent and fully-transparent appearance of the cube can be mapped to indicate that the workitem is in a different state.

6 Conclusion

Reviewing the state of the art of knowledge in the field of workflow, only a few researchers have started to explore the resource pattern modelling issue in the workflow domain. Few researchers have thought fully about how to utilize a virtual world to visualize the behaviour of human resources at an operational level.

With this in mind, we propose that an HTN can be used to model the resource patterns occurring in the lifecycle of a workitem. The major advantage of such a mathematical tool is that it can represent all resource patterns in detail, as we demonstrated. We hope the modelling approach we discussed in this paper can inspire more research works in the multiple resource pattern field.

In addition, we proposed a visual mapping mechanism between the resource patterns and a virtual world visualization. The conceptual resource pattern can be turned into an intuitive animation. This will be useful for naïve stakeholders who have less knowledge in conceptual modelling terminology, enabling them to more easily engage in resource model validation activities with business analysts.

Presently, our approach can translate resource pattern into an intuitive animation, however, subjective evaluation tests need to be performed to indicate its capacity as a visualization approach. To our best knowledge, less attention has been made to resource model visualisation as a research question. While some are investigating the perception and comprehension of 2D process models (Recker, Rosemann et al. 2009), no work has been performed in the validity of 3D process model representations.

7 References

Bhattacharya, K., C. E. Gerede, et al. (2007). Towards Formal Analysis of Artifact-Centric Business Process Models. Conference on Business Process Management.

Burdea, G. C. and P. Coiffet (2003). Virtual Reality, Wiley.

Burton-Jones, A., Y. Wand, et al. (2009). "Guidelines for Empirical Evaluations of Conceptual Modeling Grammars." Journal of the Association for Information Systems **10**(6): 495-532.

Castronova, E. (2003). Theory of the Avatar, Department of Telecommunications , Indiana University Bloomington.

Erol, K., J. Hendler, et al. (1994). Semantics for hierarchical task-network planning, University of Maryland at College Park**:** 28.

J. Parsons and L. Cole (2005). "What do the pictures mean? Guidelines for experimental evaluation of representation fidelity in diagrammatical conceptual modeling techniques." Data & Knowledge Engineering **55**(3): 327-342.

Lohse, G. L. (1997). "The Role of Working Memory in Graphical Information Processing." Behaviour and Information Technology **16**(6): 297-308.

Lu, S., A. Bernstein, et al. (2006). "Automatic workflow verification and generation." Theoretical Computer Science **353**: 71-92.

Mans, R. S., M.H.Schonenberg, et al. (2008). Application of Process Mining in Healthcare - A Case Study in Dutch Hospital. Biomedical Engineering Systems and Technologies International Joint Conference**:** 425-438.

Moody, D. L. (2009). "The "Physics" of Notations: Toward Scientific Basis for Constructing Visual Notations in Software Engineering." IEEE Transctions On Software Engineering **35**(6): 756-779.

Moore, C. (2002). Common mistakes in workflow implementations, Cambridge, MA: Giga Information Group.

Nah, F., J. Lau, et al. (2001). "Critical factors for successful implementation of enterprise systems." Business Process Management Journal **7**(3): 286-296.

Nigam, A. and N. S. Caswell (2003). "Business artifacts: An approach to operational specification." IBM Systems Journal **42**(3): 428-445.

Pesic, M. and W. M. P. van der Aalst (2007). "Modelling Work Distribution Mechanisms Using Colored Petri Nets." INTERNATIONAL JOURNAL ON SOFTWARE TOOLS FOR TECHNOLOGY TRANSFER **9**(3-4): 327-352.

Rasmussen, R. and R. Brown (2012). "A deductive system for proving workflow models from operational procedures." Future Generation Computer Systems **28**(5): 732-742.

Recker, J., M. Rosemann, et al. (2009). "Business Process Modeling: A Comparative Analysis." Journal of the Association for Information Systems **10**(4): 333-363.

Richard, L. and R. Manfred (2007). "IT support for healthcare processes– premises, challenges, perspectives." Data & Knowledge Engineering **61**: 39-58.

Russell, N., A. H. M. t. Hofstede, et al. (2004). Workflow Resource Patterns. BETA Working Paper Series. Eindhoven, Eindhoven University of Technology. **WP 127**.

Russell, N. and W. M. P. van der Aalst (2008). Work Distribution and Resource Management in BPEL4People: Capabilities and Opportunities. Advanced Information Systems Engineering.

Russell, N., W. M. P. van der Aalst, et al. (2005). Workflow Resource Patterns: Identification, Representation and Tool Support Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05).

Sadiq, S., M. Indulska, et al. (2007). Major Issues in Business Process Management: A Vendor Perspective. 11th Pacific-Asia Conference on Information Systems:Managing Diversity in Digital Enterprises**:** 40-47.

Shannon, C. E. and W. Weaver (1963). The Mathematical Theory of Communication, University of Illinois Press.

Trkman, P. (2010). "The critical success factors of business process management." International Journal of Information Management **30**(2): 125-134.

Tverskyand, B. and J. B. Morrison (2002). "Animation: can it facilitate?" Human-Computer Studies **57**: 247-262.

V. Khatri, I. Vessey, et al. (2006). "Understanding Conceptual Schemas: Exploring the Role of Application and IS Domain Knowledge." information Systems Research **17**(1): 81-99.

van der Aalst, W. M. P., K. M. v. Hee, et al. (1994). Modelling and Analyzing Workflow using a Petri-net based approach. Proceedings of Second Workshop on Computer-supported Cooperative Work, Petri nets related formalisms.

van der Aalst, W. M. P. and A. H. M. t. Hofstede (2005). "YAWL: Yet Another Workflow Language." Inofrmation Systems **30**(4): 245-275.

van der Aalst, W. M. P. and A. Kumar (2001). "A reference model for team-enabled workflow management systems." Data and Knowledge Engineering **38**(3): 335 - 363.

Wang, J. and A. Kumar (2005). A Framework for Document-Driven Workflow Systems. Business Process Management : 3rd International Conference.

zur Muehlen, M. (1999). Resource Modeling in Workflow Applications. Workflow Management Conference.

zur Muehlen, M. (2004). "Organizational Management in Workflow Applications – Issues and Perspectives." Information Technology and Management **5**: 271-291.

Inductive Definitions in Constraint Programming

Rehan Abdul Aziz **Peter J. Stuckey** **Zoltan Somogyi**

Department of Computing and Information Systems,
The University of Melbourne and National ICT Australia (NICTA)

Email: raziz@student.unimelb.edu.au, pjs@csse.unimelb.edu.au, zs@unimelb.edu.au

Abstract

Constraint programming (CP) and answer set programming (ASP) are two declarative paradigms used to solve combinatorial problems. Many modern solvers for both these paradigms rely on partial or complete Boolean representations of the problem to exploit the extremely efficient techniques that have been developed for solving propositional satisfiability problems. This convergence on a common representation makes it possible to incorporate useful features of CP into ASP and vice versa. There has been significant effort in recent years to integrate CP into ASP, primarily to overcome the *grounding* bottleneck in traditional ASP solvers that exists due to their inability to handle integer variables efficiently. On the other hand, ASP solvers are more efficient than CP systems on problems that involve inductive definitions, such as reachability in a graph. Besides efficiency, ASP syntax is more natural and closer to the mathematical definitions of such concepts. In this paper, we describe an approach that adds support for answer set rules to a CP system, namely the lazy clause generation solver chuffed. This integration also naturally avoids the grounding bottleneck of ASP since constraint solvers natively support finite domain variables. We demonstrate the usefulness of our approach by comparing our new system against two competitors: the state-of-the-art ASP solver clasp, and clingcon, a system that extends clasp with CP capabilities.

Keywords: Answer set programming, constraint programming, stable model semantics, inductive definitions.

1 Introduction

Constraint programming (Rossi et al. 2006) is a declarative programming paradigm that is used to solve a wide range of computationally difficult problems. It allows users to encode a problem as a concise mathematical model, and pass it to a constraint solver that computes solution(s) that satisfy the model. The goal of CP is that users should find writing models as natural and as easy as possible, which requires hiding away all the complexity involved in finding those solutions inside the constraint solvers, where only the solvers' implementers have to see it.

Motivated by the efficient engineering techniques developed in the domain of propositional satisfiability (SAT) solving (Mitchell 2005), a recent highly competitive constraint solving approach, *lazy clause generation* (Ohrimenko et al. 2009, Feydy & Stuckey 2009) builds an on-the-fly Boolean representation of the problem during execution. This keeps the size of the representation small. The propagators record their results of failed searches as Boolean clauses (*nogoods*) so that the solver can later use SAT unit propagation on those clauses to find other instances of that failure elsewhere in the search tree much more quickly.

Constraint modelling languages allow users to succinctly define many natural notions, particularly when using *global constraints*, which can capture the entirety of some substructure of the problem. Global constraints also allow solvers to use efficient specialized reasoning about these substructures. But constraint modelling cannot naturally capture some important constructs, such as transitive closure. (Propositional) definite logic programs do allow the modelling of transitive closure efficiently, because they rely on a least model semantics, which ensures that *positive recursion* in the rules among a set of atoms, i.e. circular support between these atoms to establish each other's truth, is not sufficient to cause an atom to be true. When we extend logic programs to normal logic programs, which allow negative literals in the body, we can extend the least model approach in at least two ways which still maintain this property: the *stable model* (Gelfond & Lifschitz 1988) and the *well-founded model* (Van Gelder et al. 1988).

Answer set programming (ASP) (Baral 2003), based on stable model semantics, is another form of declarative programming. Answer set solvers take as input a normal logic program, usually modelling a combinatorial problem, and calculate its stable models, each of which corresponds to one of the problem's solutions. The incorporation of some of the engineering techniques originally developed for SAT solvers (such as nogood learning) in answer set solvers has resulted in excellent performance (Gebser et al. 2007). The implementation of these techniques relies on translating the normal logic program back to propositional formulas, an approach which was first proposed by Lin & Zhao (2004). Representing the program as its Clark's completion introduces practically no overhead, but detecting *unfounded sets* (Van Gelder et al. 1988), sets of atoms that are supported only by each other but have no *external* support, is far from straightforward. The main reason for this is that a program can potentially have a very large number of unfounded sets (Lifschitz & Razborov 2006). To tackle this, Gebser et al. (2007) use a principle similar to lazy clause generation: they calculate and record unfounded sets *lazily* during the search, as the need arises.

The effectiveness of the above approach motivates us to incorporate stable model semantics into our modelling language MiniZinc (Nethercote et al. 2007) in order to broaden the scope of problems that we can model; in particular, we are thinking about problems such as reachability, whose mathematical descriptions require induction. Unless specialized graph constraints are used (Dooms et al. 2005, Viegas & Azevedo 2007), these problems cannot be efficiently handled by existing constraint solvers. In this paper, we propose the use of inductive definitions in MiniZinc and empirically demonstrate its usefulness. We describe two implementations of unfounded set calculation as propagators for the lazy clause generator chuffed. These implementations are based on the *source pointer* technique (Simons et al. 2002) combined with either of the unfounded set algorithms described by Anger et al. (2006) and Gebser et al. (2012).

The rest of the paper is organized as follows. Section 2 lays out the theoretical background required for this paper. Section 3 describes, with the help of a running example, how recursive definitions under propositional semantics lack the ability to model certain problems correctly and efficiently, and presents an extension of the MiniZinc modelling language as a solution. Section 4 explains how this extension may be implemented. Section 5 describes one of our two implementations in detail. We evaluate both implementations experimentally in Section 6. We then discuss related work by other authors in Section 7.

2 Background

Constraints and propagators

We consider constraints over a set of variables $\mathcal{V}$. We divide $\mathcal{V}$ into two disjoint sets, namely integer variables $\mathcal{I}_\mathcal{V}$ and Boolean atoms $\mathcal{A}_\mathcal{V}$. A literal is an atom or its negation. A domain D is a mapping: from $\mathcal{I}_\mathcal{V}$ to fixed finite sets of integers, and from $\mathcal{A}_\mathcal{V}$ to sets over $\{\top, \bot\}$. A domain D_1 is *stronger* than a domain D_2, written $D_1 \sqsubseteq D_2$, if $D_1(x) \subseteq D_2(x)$ for all $x \in \mathcal{V}$. A valuation θ is a mapping of variables to a single value in their domains, written $\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\}$. Let $vars$ be a function that returns the set of variables that appear in any expression. We say that a valuation θ is an element of the domain D, written $\theta \in D$, if $\theta(x) \in D(x)$ for all $x \in vars(\theta)$. A valuation θ is *partial* if $vars(\theta) \subset \mathcal{V}$ and *complete* if $vars(\theta) = \mathcal{V}$.

A constraint c is a restriction on the values that a set of variables, represented by $vars(c)$, can be simultaneously assigned. In our setting, a constraint c is associated with one or more *propagators* that operate on $vars(c)$. Propagators for a constraint work by narrowing down the values that the variables of the constraint can take. More formally, a propagator f is a monotonically decreasing function from domains to domains, that is, $f(D) \sqsubseteq D$, and $f(D_1) \sqsubseteq f(D_2)$ if $D_1 \sqsubseteq D_2$. A propagator f for a constraint c is *correct* iff for all possible domains D, and for all solutions θ to c, if $\theta \in D$, then $\theta \in f(D)$.

A CP problem is a pair (C, D) consisting of a set of constraints C and a domain D. A constraint programming solver solves (C, D) by interleaving propagation with choice. It applies all propagators F for constraints C to the current domain D, and it does so repeatedly until no propagator makes a change (i.e. until a fixpoint is reached). If the final domain D' represents failure ($D(x) = \emptyset$ for some x) then it backtracks to try another choice. If all variables have at least one element in their domains, but some have two or more, then the solver needs to make a choice by splitting the domain of one of these variables into two parts. This *labelling step* results in two subproblems (C, D'') and (C, D'''), which the solver then solves recursively. If all variables have exactly one value in their domains, then there are no choices left to be made, and the domain is actually a valuation. Whether that valuation satisfies all the constraints can be trivially checked, although this check is unnecessary if the propagators of all constraints are guaranteed to find any failures of those constraints. In practice, solvers use event-driven scheduling of propagators and priority mechanisms to try to reach fixpoints as quickly as possible (Schulte & Stuckey 2008).

(Propositional) Normal logic programs

In our proposed system, we divide the set of atomic variables $\mathcal{A}_\mathcal{V}$ into two disjoint subsets: the set of *default* variables $\mathcal{D}_\mathcal{V}$, and the set of non-default variables $\mathcal{N}_\mathcal{V}$. A *normal rule* r has the form:

$$a \leftarrow p_1, \dots, p_j, {\sim} n_1, \dots, {\sim} n_k$$

where $a \in \mathcal{D}_\mathcal{V}$ and $\{p_1, \dots, p_j, n_1, \dots, n_k\} \subseteq \mathcal{A}_\mathcal{V}$. We say that a is the *head* of r, written r_H, and $\{p_1, \dots, p_j, {\sim} n_1, \dots, {\sim} n_k\}$ is the *body* of r, written r_B. To allow us to represent the truth or falsity of each rule body, we have the *bodyRep* function, which maps each rule body to a new *body atom* $b \in \mathcal{N}_\mathcal{V}$. We also have functions that return the positive and negative atoms in each rule body: if $bodyRep(r_B) = b$, then $pos(b) = \{p_1, \dots, p_j\}$ and $neg(b) = \{n_1, \dots, n_k\}$. We call the set of positive default literals of the body $b^+ = \{p \mid p \in pos(b), p \in \mathcal{D}_\mathcal{V}\}$.

A normal logic program (NLP) is a set of normal rules. We consider an NLP $\mathcal{P}$ as a constraint. We define the following functions for a default atom a and a body atom b. The set of all body atoms in the program is $bodies(\mathcal{P}) = \{bodyRep(r_B) \mid r \in \mathcal{P}\}$; the set of bodies of the rules whose head is a is $body(a) = \{bodyRep(r_B) \mid r \in \mathcal{P}, r_H = a\}$; the set of heads supported by b is $supHead(b) = \{r_H \mid r \in \mathcal{P}, bodyRep(r_B) = b\}$; and the set of body atoms in whose positive parts $a \in \mathcal{D}_\mathcal{V}$ appears is $posInBody(a) = \{c \mid c \in bodies(\mathcal{P}), a \in c^+\}$.

We use the concept of *positive body-head dependency graph* as defined in (Gebser & Schaub 2005). It is a directed graph $(\mathcal{D}_\mathcal{V} \cup bodies(\mathcal{P}), E(\mathcal{P}))$ where $E(\mathcal{P}) = \{(a, b) \mid a \in \mathcal{D}_\mathcal{V}, b \in posInBody(a)\} \cup \{(b, a) \mid b \in bodies(\mathcal{P}), a \in supHead(b)\}$.

We associate each strongly connected component of the graph with a number, and we map every atom and body literal in the component to this number through a function *scc*.

Our implementation of stable model semantics relies on the translation of logic programs into propositional theories. Given a default atom $a \in \mathcal{D}_\mathcal{V}$, the Clark completion (Clark 1978) of its definition, $Comp(a)$, is the formula $a \leftrightarrow \vee_{b \in body(a)} b$. The completion of $\mathcal{P}$ is the formula:

$$Comp(\mathcal{P}) = \bigwedge_{x \in \mathcal{D}_\mathcal{V}} Comp(x)$$

The following formula ensures that all body literals are equal to the conjunction of their literals:

$$Body(\mathcal{P}) = \bigwedge_{b \in bodies(\mathcal{P})} (b \leftrightarrow \bigwedge_{p \in pos(b)} p \wedge \bigwedge_{n \in neg(b)} \neg n)$$

We refer to the conjunctive normal form (CNF) of the formula $Comp(P) \wedge Body(P)$ as $Clauses(P)$.

Given a valuation θ, a set $U \subseteq \mathcal{D}_{\mathcal{V}}$ is *unfounded* with respect to θ iff for every rule $r \in \mathcal{P}$ and $bodyRep(r_B) = b$:

1. $r_H \notin U$ or
2. $\theta(p) = \bot$ for some $p \in pos(b)$ or $\theta(n) = \top$ for some $n \in neg(b)$ or
3. $b^+ \cap U \neq \emptyset$.

Basically, a set is unfounded if every default variable in it depends on some other variable in it being true, but none of them have *external support*, i.e. none of them can be proven true without depending on other default variables in the set. This is expressed most directly by the third alternative. The previous alternatives cover two different uninteresting cases: rules whose heads are not in the potentially unfounded set we are testing, and rules whose bodies are known to be false.

Finally, we say that a complete valuation θ satisfies P, or that θ is a *constraint stable model* of P iff $\theta \models Clauses(P)$ and there is no $U \subseteq \mathcal{D}_{\mathcal{V}}$ such that U is an unfounded set with respect to θ.

Constraint stable models can also be defined through use of *constraint reducts* (Gebser et al. 2009). Given a valuation θ such that $vars(\theta) \supseteq \mathcal{N}_{\mathcal{V}}$ and $vars(\theta) \cap \mathcal{D}_{\mathcal{V}} = \emptyset$, the constraint reduct of $\mathcal{P}$ with respect to θ, written $\mathcal{P}^{\theta}$, is a version of $\mathcal{P}$ that has no non-default variables. The reduct is computed by first removing the rules whose bodies have one or more literals that are not satisfied by θ, and then removing satisfied non-default atoms from the rule bodies. A complete valuation θ' that extends θ is a constraint stable model of $\mathcal{P}$ if θ' is a stable model of the reduced program $\mathcal{P}^{\theta}$.

3 Inductive definitions

The goal of this section is to demonstrate the usefulness of inductive definitions in constraint modelling languages with the help of an example. Consider the *Connected Dominating Set* problem.[1] Given a graph `G` defined by `n` nodes numbered `1..n` and a (symmetric) adjacency 2D array `edge` where `edge[i,j] = edge[j,i] = true` iff the nodes `i` and `j` are adjacent. A connected dominating set `D` is a subset of `1..n` such that for each node `i`, either `i` or one of its neighbours is in `D`, and the set `D` is connected (each node in `D` can reach other nodes in `D` by a path of edges both of whose endpoints are in `D`). The problem is to find a dominating set with at most `k` nodes for a given graph.

A MiniZinc (Nethercote et al. 2007) model for this problem excluding the connectedness condition is:

```
int: k;                        % size limit
int: n;                        % nodes in G
set of int: N = 1..n;          % node set
array[N,N] of bool: edge;      % edges in G

array[N] of var bool: d;       % is member of D?

constraint sum(i in N)(bool2int(d[i])) <= k;
constraint forall(i in N)
  d[i] \/ exists(j in N where
    edge[i,j])(d[j]);
```

[1]See http://dtai.cs.kuleuven.be/events/ASP-competition/Benchmarks/ConnectedDSet.shtml

But modelling the connectedness condition in CP is very difficult. We may imagine we can model this by saying each node in `D` is adjacent to another node in `D`.

```
constraint forall(i in N)
  (d[i] -> exists(j in N, j != i)
    (d[j] /\ edge[i,j]);
```

This model is incorrect, for example for the symmetric completion of the asymmetric graph with edges `{(1,2),(2,3),(3,4),(4,5),(4,6),(5,6)}` and limit 4, it has a solution `D = {1,2,4,5}` which is dominating but not connected.

We need to define a base case for connectedness, and reason about reachability from there in terms of distance: `reach[n,s]` means we can reach node `n` from base node `min_idx` (the node in `D` with least index) in `s` or fewer steps. (The `var N` indicates that `min_idx` must belong to the previously defined set `N`.)

```
var N: min_idx = min(i in N)
  (i + bool2int(not d[i])*n);
array[N,0..n-1] of var bool: reach;
constraint forall(i in N)
  (reach[i,0] <-> i == min_idx);
constraint forall(i in N)
  (forall(s in 0..n-2)
    (reach[i,s+1] <-> (reach[i,s] \/
       exists(j in N where edge[i,j])
         (d[i] /\ reach[j,s]))));
constraint forall(i in N)
  (d[i] -> reach[i,n-1]);
```

The model defines only `min_idx` as reachable with 0 steps, and node `i` is reachable in `s+1` steps (`reach[i,s+1]`) if it was reachable previously or if it is in `d` and there is an adjacent node `j` reachable in `s` steps. The model is correct giving two answers `D = {2,3,4,5}` and `D = {2,3,4,6}`. This model is very expensive, requiring `n*n` Boolean variables to define the final connected set.

Lets consider the ASP model for the same problem, shown here in `gringo` syntax:

```
% select the dominating set
{ dom(U) : vtx(U) }.

% dominating set condition
in(V) :- edge(U,V), dom(U).
in(V) :- dom(V).
:- vtx(U), not in(U).

% connectivity constraints
reach(U) :- dom(U),
  not dom(V) : vtx(V) : V < U.
reach(V) :- reach(U), dom(V), edge(U,V).
:- dom(U), not reach(U).

% size bound
:- not { dom(U) : vtx(U) } K, bound(K).
```

where the input is `vtx(i)` where `i` is in `1..n`, `edge(i,j)` whenever `edge(i,j)`, and `bound(k)` for limit `k`. The size constraint is expressed in negation, while the dominating set is expressed more obscurely. The base case that relies on defining minimum index computation is arguably more transparent. The biggest difference is the reachability condition which is much more succinct and much more efficient.

The advantage of the ASP model is that it makes use of the *inductive* interpretation of the rules for transitive closure. The solution `D = {1,2,4,5}` is not generated because the transitive closure computation cannot generate reach(4) or reach(5) from reach(1).

In order to incorporate this succinct modelling, and to take advantage of the efficient solving approaches for this, we extend MiniZinc with *inductively defined predicates*.

For the running example we use an *inductive definition* of the predicate reach as follows

```
idpredicate reach(N: i) = i == min_idx \/
  exists(j in N where edge[i,j])
     (d[i] /\ reach(j));
```

and add the constraint

```
constraint forall(i in N)(d[i] -> reach(i));
```

Inductively defined predicates are allowed to have only fixed arguments; that is, their arguments cannot be decision variables. They can use arbitrary MiniZinc in the bodies with the restriction that they cannot introduce new decision variables, and inductively defined literals cannot appear inside non-Boolean constraint expressions. The example above uses the existing decision variables min_idx and d in the body.

In MiniZinc we assume that inductively defined predicates have an (extended) stable model semantics (Gelfond & Lifschitz 1988). At the moment, we have not defined the stable models of inductive definitions. Instead, we talk about the constraint stable models of an equivalent set of normal rules. What the extended stable models of the inductive definitions should be, and how the translation of the inductive predicate into a set of normal rules should work, are both directions for our future research. We comment further on this topic in Section 8.

4 Mapping inductive definitions to FlatZinc

The first step in solving a MiniZinc model is having the translator program mzn2fzn map it to FlatZinc, a lower level language that is easier for solvers to understand and implement. In this section, we show what this translation has to do for MiniZinc models that contain inductive definitions.

The first task of the translation is identifying the default variables used by inductively defined predicates, and defining them properly. The FlatZinc we want to generate for a default variable named dv is:

```
var bool: dv;
default_variable(dv);
```

The first line defines dv as a Boolean variable, while the second tells the solver that this variable is a default variable, and not an ordinary Boolean variable. This line defines a predicate we added to FlatZinc:

```
predicate default_variable(var bool: v);
```

Each use of this predicate declares a default Boolean variable. For our running example from previous section, we want to generate this FlatZinc:

```
array[N] of var bool: reach;
default_variable(reach[1]);
...
default_variable(reach[n]);
```

The second task of the translator is replacing the inductively defined predicate for each default variable with a set of normal logic rules that have the same constraint stable models as that predicate.

To make this possible, we have extended FlatZinc with a predicate that represents normal rule definitions:

```
predicate normal_rule(var bool: head,
  array[int] of var bool: pos_atoms,
  array[int] of var bool: neg_atoms);
```

As the names suggest, pos_atoms and neg_atoms contain the atoms appearing in positive and negative literals in the rule respectively, so the generic normal rule r shown in Section 2 would be represented as

$$\texttt{normal\_rule}(a, [p_1, \ldots, p_j], [n_1, \ldots, n_k])$$

The head must be a default variable, but the positive and negative literals have no such restriction.

MiniZinc supports quantifiers, and the inductive definition of reach in the previous section included a quantifier. FlatZinc does not allow quantifiers, so we must eliminate them during translation. This requires knowing the data over which such predicates operate. For reach, this data is the edge predicate. If the edge predicate contains the three facts {(1,2),(2,1),(2,2)}, representing a small graph with two nodes and three edges, we want to translate reach into these constraints:

```
normal_rule(reach[1],[min_idx==1],[]);
normal_rule(reach[2],[min_idx==2],[]);
normal_rule(reach[1],[d[1],reach[2]],[]);
normal_rule(reach[2],[d[2],reach[1]],[]);
normal_rule(reach[2],[d[2],reach[2]],[]);
```

The first step in this translation is the replacement of existential quantifiers in predicate bodies with disjunctions. The second step is the replacement of disjunctions in bodies with two or more rules. This particular translation is essentially a form of the Lloyd-Topor transformation (Lloyd & Topor 1984). While that is the translation we want to do, we have not yet implemented either the translation, or the recognition of idpredicate definitions in MiniZinc. Yet to test the effectiveness of our system, we need *some* way to generate FlatZinc code that implements inductive definitions. To do this, we have exploited mzn2fzn's existing ability to expand out quantifications. We have simply added the normal_rule predicate to MiniZinc as well as FlatZinc.

To generate the FlatZinc that we would want generated from

```
idpredicate reach(N: i) = i == min_idx \/
  exists(j in N where edge[i,j])
    (d[i] /\ reach(j));
```

a MiniZinc user can now write these constraints:

```
constraint forall(i in N)
  (normal_rule(reach[i],[i==min_idx],[]));

constraint forall(i,j in N where edge[i,j])
  (normal_rule(reach[i],[d[i],reach[j]],[]));
```

Basically, until we implement our full translation, we require users to expand out existential quantifiers and disjunctions for themselves, although as we have shown above, this can be conveniently done with the MiniZinc generator forall.

5 Implementation

Before we describe our implementation in detail, let us sketch briefly how propagation works in chuffed. A propagator can subscribe to an event e, written *Subscribe*(e). When the event e takes place, the *WakeUp* function of the propagator is called. At this point, the propagator can *Queue* up for propagation.

Since a single event can wake up more than one propagator, each propagator has a priority; any woken propagators are added to the queue in priority order. The *Propagate* function of a propagator is called after all the higher priority propagators have finished. The code of the *Propagate* function can choose to *Requeue* itself after it has done some work, if it wants higher priority propagators to run before it does some more work.

For unfounded set calculation, we have implemented two approaches taken from existing literature. The first one is based on the approach outlined by Gebser et al. (2007), which is the combination of smodels' source pointer technique (Simons et al. 2002) with the unfounded set computation algorithm described by Anger et al. (2006). The second approach follows Gebser et al. (2012) in combining the source pointer technique with a different unfounded set computation algorithm (described in that paper). We call our implementation of the first approach anger, and the second one gebser, after the authors of their unfounded set algorithms.

Computing unfounded sets is inherently more expensive than most other propagators. We therefore want to invoke the unfounded set propagator as rarely as possible, which requires its priority to be low. This low priority is required for another reason as well: the algorithms used by the unfounded set propagator need to run *after* unit propagation has finished, so that they have access to a consistent valuation of all the Boolean solver variables. If they do not, then the work that they do is likely to turn out to be wasted.

In the rest of this section, we will describe the first approach in detail, and then briefly outline the second approach. For all the algorithms used in this section, we assume that they access the valuation θ.

Initial calculations

When the solver is initialized, prior to any propagation and search, we calculate $Clauses(\mathcal{P})$ and record its clauses. We could also optimize $Clauses(\mathcal{P})$ by exploiting any equivalences present in it (Gebser et al. 2008), but we do not (yet) do so. For example, if one atom a is known to be exactly equivalent to a body b, because a is defined by only one clause represented by b, we can consistently use one constraint variable for both a and b. Doing so reduces both the number of variables the solver needs to manage, and eliminates the propagation steps that would otherwise be needed to keep them consistent.

We then calculate the strongly connected components of the body-head graph implicit in $\mathcal{P}$. We number each component, and assign each default atom and body the number of the component it is in. The only atoms of interest are those whose components contain more than one atom, since only they can ever participate in an unfounded set. The calculation of the SCCs allows us to distinguish between these *cyclic* atoms, and all other atoms, which are *acyclic.*

Establishing source pointers

Both our unfounded set detection algorithms are based on the idea of source pointers. Each cyclic default atom has a *source*, which is a non-false body b such that atoms in b^+ are not unfounded. As long as the source of an atom is non-false, the atom has evidence of not being unfounded. If the source of an atom becomes false, then we must look for another source for it; if we cannot find one, then the atom is part of an unfounded set.

Algorithm 1 *EstablishSourcePointers*()

```
1: for each atom $a$ do $source(a) \leftarrow \bot$
2: for each body $b$ do
3:     if $\theta(b) \neq \bot$ then $ct(b) \leftarrow |b^+|$ else $ct(b) \leftarrow \infty$
4:     if $ct(b) = 0$ then
5:         if $\theta(b) = \top$ then
6:             $MustBeQ.add(b)$
7:         else
8:             $MayBeQ.add(b)$
9: while $MustBeQ \neq \emptyset$ do
10:     $b \leftarrow MustBeQ.pop()$
11:     for $a \in supHead(b) : source(a) = \bot$ do
12:         $source(a) \leftarrow \top$
13:         for $c \in posInBody(a)$ do
14:             $ct(c) \leftarrow ct(c) - 1$
15:             if $ct(c) = 0$ then
16:                 if $\theta(c) = \top$ then
17:                     $MustBeQ.add(c)$
18:                 else
19:                     $MayBeQ.add(c)$
20: while $MayBeQ \neq \emptyset$ do
21:     $b \leftarrow MayBeQ.pop()$
22:     for $a \in supHead(b) : source(a) = \bot$ do
23:         $source(a) \leftarrow b$, $Subscribe(b = \bot)$
24:         for $c \in posInBody(a)$ do
25:             $ct(c) \leftarrow ct(c) - 1$
26:             if $ct(c) = 0$ then $MayBeQ.add(c)$
27: for each atom $a : source(a) = \bot$ do $\theta(a) = \bot$
```

We initialize the source pointers of default variables before beginning search. Our initialization, shown in Algorithm 1, partitions the set of default atoms into three disjoint sets: *MustBeTrue*, the set of atoms that are true in every constraint stable model of $\mathcal{P}$; *MayBeTrue*, the atoms that can be true in some constraint stable model; and *CantBeTrue*, atoms that cannot be true in any constraint stable model. Atoms in *MustBeTrue* cannot be part of any unfounded set, and the unfounded atoms in *CantBeTrue* can be set to false at this early stage. Only atoms in *MayBeTrue* actually require source pointers; we record the source "pointers" of atoms in *MustBeTrue* as $\top$, and the source pointers of atoms in *CantBeTrue* as $\bot$.

Algorithm 1 describes a bottom-up calculation which is similar to the Dowling-Gallier algorithm (Dowling & Gallier 1984). The algorithm keeps two queues of bodies, *MustBeQ* and *MayBeQ*, that we use to incrementally build *MustBeTrue* and *MayBeTrue* respectively. If for some $b \in bodies(\mathcal{P})$, $\theta(b) = \top$ and $b^+ \subseteq MustBeTrue$, then we add body b to *MustBeQ*. Otherwise, if $\theta(b) \neq \bot$ and $b^+ \subseteq MustBeTrue \cup MayBeTrue$, then we add b to *MayBeQ*. Since the heads of a body b in *MayBeQ* can become true due to b, we set their sources to b. Whenever we assign the id of a solver variable b to be the source of another variable a, we make the propagator subscribe to the event $b = \bot$, since if b becomes false, the propagator must determine a new source for a (or construct an unfounded set from a if one exists).

The algorithm works by keeping a count $ct(b)$ for each body b. Before the end of the first while loop, $ct(b)$ represents the number of atoms in b^+ that we need to find in *MustBeTrue* before we can put the heads supported by b into *MustBeTrue*. After the end of the first while loop, when there is no possibility left of finding any atoms that must be true, $ct(b)$ represents the number of atoms in b^+ that we need to find in *MayBeTrue* before we can put the heads supported by b into *MayBeTrue*.

At the end, we set all the atoms that do not have a source to false, since these atoms that cannot be true in any constraint stable model of the program.

Algorithm 2 *WakeUp*($b = \bot$)

```
1: if first wakeup on current search tree branch then
2:     $U \leftarrow \emptyset$, $P \leftarrow \emptyset$
3: for each atom $a$ : $source(a) = b$ do
4:     if $\theta(a) \neq \bot$ then
5:         $P.add(a)$
6:         $Queue()$
```

Algorithm 3 *Propagate*()

```
1: $U \leftarrow U \setminus \{a \in \mathcal{D}_\mathcal{V} \mid \theta(a) = \bot\}$
2: $P \leftarrow P \setminus \{a \in \mathcal{D}_\mathcal{V} \mid \theta(a) = \bot\}$
3: while $U = \emptyset$ do
4:     if $P = \emptyset$ then return
5:     $a \leftarrow P.pop()$
6:     if $\theta(a) \neq \bot$ then
7:         if $\exists b \in body(a) : \theta(b) \neq \bot$ and $scc(a) \neq scc(b)$ then
8:             $source(a) \leftarrow b$, $Subscribe(b = \bot)$
9:         else
10:            $UnfoundedSet(a)$
11: $a \leftarrow U.remove()$
12: if $\theta(a) \neq \top$ then $Requeue()$
13: $\theta(a) = \bot$
14: add $loop\_nogood(U, a)$
```

The *WakeUp* and *Propagate* functions

EstablishSourcePointers() subscribes our propagator to events that record the source of a default variable becoming false. When such events happen, the subscription system will call the *WakeUp* function in Algorithm 2, which delegates most of its work to the *Propagate* function in Algorithm 3. These two functions jointly manage two global variables: U, which contains atoms that form an unfounded set, and P, which contains atoms that are pending an unfounded check. These variables are global because they must retain their values across all the propagation invocations in a propagation step between two consecutive labelling steps. We set both variables to be empty the first time we get control after a labelling step.

When an invocation of *WakeUp* tells us that a body b is false, we add the atoms supported by b to the pending queue P unless they are already known to be false. However, we do not process the pending queue immediately; we let higher priority propagators run first, to allow them to tighten the current valuation as much as possible before we process the pending queue in our own low priority *Propagate* function.

The *Propagate* function starts by removing all the atoms that have become false from both U and P. (Other propagators with higher priorities can set an atom in P to false *after* the *WakeUp* that put that atom in P.)

If U is not empty, we remove an atom a from U, set it to false, add its *loop nogood* (see below) to the set of learned constraints, and requeue the propagator to allow propagators of higher priority efficiently propagate the effects of setting a to false. This may or may not fix the values of all the atoms in the updated U. While it does not, each invocation of *Propagate* will set another unfounded atom to false.

For a given set of default atoms $U \subseteq \mathcal{D}_\mathcal{V}$, we denote the set of *external bodies* as $EB(U) = \{b \mid b \in bodies(\mathcal{P}), supHead(b) \cap U \neq \emptyset, b^+ \cap U = \emptyset\}$. The loop nogood of a set U with respect to an atom $a \in U$ is $loop\_nogood(U, a) = (\neg a \vee b_1 \vee \ldots \vee b_n)$ where $EB(U) = \{b_1, \ldots, b_n\}$ (Gebser et al. 2007). This captures the idea that an atom in a set cannot be true unless one of the external bodies of the set is true.

When there are no more known-to-be-unfounded atoms left, we look for atoms in the pending queue that can be part of an unfounded set. If the pending queue is empty, then there cannot be any more unfounded sets, and we are done. If there is an atom a in P, we test whether it is supported by an external body, a body b in a lower SCC. If it is, then a is not unfounded. If it isn't, then it is possible that a is part of an unfounded set, and we invoke *UnfoundedSet* to check if a can be extended to an unfounded set. If the call fails and leaves $U = \emptyset$, we try again with a different member of P. If it succeeds, we handle the newly-made unfounded set the same way as we handle unfounded sets that already exist when *Propagate* is invoked.

Unfounded set calculation

Algorithm 4 is based on the unfounded set algorithm by Anger et al. (2006). Its key local data structure is *Unexp*, which contains the bodies that may contain external support for some of the atoms in U. A body variable b can support an atom a only if it represents the body of one of the rules of a, it is not false, and it does not contain any atoms that are themselves unfounded. If b is in a lower SCC than a, then we take its valuation as a given; if it is not false, then it supports a and therefore a cannot be declared unfounded; if it is false, then it does not support a. If it is in the same SCC as a, then b may or may not support a; we need to find out which. That is why we put into *Unexp* the set of bodies that may support the atoms in U. To help us to do this, we define the function $maysupport(a)$ as $\{b \mid b \in body(a), \theta(b) \neq \bot, b^+ \cap U = \emptyset, scc(b) = scc(a)\}$. The algorithm uses two other data structures, *SAtoms* and *SBodies* (S is for *supported* in this context): *SAtoms* contains atoms that have been proven to be not unfounded, while *SBodies* contains externally supported bodies. A non-false body b is externally supported if every atom in b^+ is either in *SAtoms*, or belongs to a different component.

The algorithm processes the unexplored bodies in *Unexp* one by one. It looks at the positive default atoms in each such body. Those that are in *SAtoms* or in a lower SCC are known to support b; the others are not. We compute $nks(b, curscc, SAtoms)$ as the set of not-known-to-be-supporting atoms in b: $nks(b, curscc, SAtoms) = \{p \mid p \in b^+ : p \notin SAtoms$ and $scc(p) = curscc\}$

If this is not empty, then then we need to test the atoms in it to see whether or not they actually do support b. If the test on Line 10 succeeds for an atom p, then we have found a source for p. The algorithm records this source. It then removes p from P and adds it to the set of supported atoms. If the test on Line 10 fails, then the algorithm makes p part of the unfounded set U. The definition of *Unexp* says that bodies whose positive atoms are in U must not be in it; on line 17 we remove from it the bodies that would now violate that invariant. To allow later iterations of the outermost loop to check whether p can be supported via other bodies, we then add those possible bodies to *Unexp*. All these changes may have reduced the set of not-known-to-be-supporting atoms to the empty set, which is why we compute that set again.

If the set of not-known-to-be-supporting atoms is empty, either originally or after being recomputed, then we know b is externally supported (Line 20). This means that all atoms in U that have a rule whose body is represented by b are now supported by b. We compute R as the set of these atoms, and we record b as their source. We also remove them from U and P, and add them to *SAtoms*. Adding them to

Algorithm 4 *UnfoundedSet*(a)

```
1: $cursc \leftarrow scc(a)$
2: $U \leftarrow \{a\}$
3: $Unexp \leftarrow maysupport(a)$
4: $SAtoms \leftarrow \emptyset$, $SBodies \leftarrow \emptyset$
5: while $Unexp \neq \emptyset$ do
6:   $b \leftarrow Unexp.pop()$
7:   if $nks(b, curscc, SAtoms) \neq \emptyset$ then
8:     [$b$ is not externally supported]
9:     for $p \in nks(b, curscc, SAtoms)$ do
10:      if $\exists c \in body(p) : \theta(c) \neq \bot$ and
         $(scc(c) \neq curscc$ or $c \in SBodies)$ then
11:        if $scc(source(p)) = curscc$ then
12:          $source(p) \leftarrow c$, $Subscribe(c = \bot)$
13:        if $P.contains(p)$ then $P.remove(p)$
14:        $SAtoms.add(p)$
15:      else
16:        $U.add(p)$
17:        $Unexp \leftarrow Unexp \setminus \{d \mid d \in Unexp, p \in d^+\}$
18:        $Unexp \leftarrow Unexp \cup maysupport(p)$
19:  if $nks(b, curscc, SAtoms) = \emptyset$ then
20:    [$b$ is externally supported]
21:    $SBodies.add(b)$
22:    $R \leftarrow \{r \mid r \in U, b \in body(r)\}$
23:    for $r \in R$ do
24:      $source(r) = b$
25:    $Subscribe(b = \bot)$
26:    while $R \neq \emptyset$ do
27:      $r \leftarrow R.pop()$
28:      $U.remove(r)$
29:      $P.remove(r)$
30:      $SAtoms.add(r)$
31:      for $j \in posInBody(r) : \theta(j) \neq \bot$ and
         $\forall t \in j^+, (t \in SAtoms$ or $scc(t) \neq curscc)$ do
32:        $SBodies.add(j)$
33:        for $a \in supHead(j) \cap U$ do
34:          $source(a) \leftarrow j$, $Subscribe(j = \bot)$
35:          $R.add(a)$
36:    $Unexp \leftarrow \bigcup_{p \in U} maysupport(p)$
```

SAtoms may make more bodies qualify for membership of *SBodies*, which in turn may provide external support for more atoms. We put any such atoms into R as well, and we keep going until everything in R has been processed. Once we have removed as many atoms as possible from U and have reached a fixpoint, we reinitialize *Unexp* based on the final value of U.

Second approach

Our second implementation, gebser, differs from our first, anger, only in its use of a different unfounded set algorithm. We have taken that algorithm directly from (Gebser et al. 2012), so here we just give its outline. The algorithm uses the concept of a *scope*, which is an upper bound on U. The algorithm computes the scope by starting with P, and extending it through a fixpoint algorithm. It then computes U by restricting the scope to a single SCC.

6 Experiments

We benchmarked our implementations anger and gebser against two competing systems. The first is a combination of clasp (version `2.0.6`) and gringo (version `3.0.4`), which we call cl+gr in our tables for brevity. The second is clingcon (Gebser et al. 2009) (version `2.0.0-beta`), which is an extension of clasp with CP capabilities. We ran all the benchmarks on a Lenovo model 3000 G530 notebook with a 2.1 GHz Core 2 Duo T6500 CPU and 3 GB of memory running Ubuntu 12.04. We repeated each experiment with a timeout five times, and each experiment without a timeout twice; the results we present are their averages.

		Solved	Opt	AvgPct
RoutingMin	cl+gr	N/A	N/A	N/A
	clingcon	19	3	69.1
	gebser	19	8	33.5
	anger	18	9	33.9
RoutingMax	cl+gr	N/A	N/A	N/A
	clingcon	22	0	100.0
	gebser	27	0	42.2
	anger	27	0	37.4

Table 1: Results for routing on 34 instances

We ran two sets of benchmarks. The first set consists of different instances of two routing problems, which are slightly modified versions of the models used in the experiments by Liu et al. (2012). Our reason for selecting these two problems is that they involve not just reachability, but also variables with large finite domains. The two problems differ only in their objective; they use the same data representation and impose the same set of constraints. Each instance of these problems is specified by

- a weighted directed graph (V, E, w) where $w : E \mapsto \mathbb{N}$,
- a source node $s \in V$,
- a set of destination nodes $D \subseteq V \setminus \{s\}$, and
- a deadline for each destination $f : D \mapsto \mathbb{N}$.

Their solutions consist of two parts:

- a cycle-free route $(r_0, r_1, \dots, r_k)$ where $r_0 = s$, $(r_i, r_{i+1}) \in E$ for all $i \in \{1, ..., k-1\}$, and for each $d \in D$, $d = r_i$ for some $i \in \{1, ..., k\}$, and
- a time assignment $t : V \mapsto \mathbb{N}$ such that $t(r_0) = 0$, $t(r_{i+1}) \geq t(r_i) + w(r_i, r_{i+1})$ for all $i \in \{1, ..., k-1\}$, and for each $d \in D$, $t(d) \leq f(d)$.

For the *RoutingMin* problem, the objective is minimizing the total delay $\sum_{d \in D}(f(d) - t(d))$. For the *RoutingMax* problem, the objective is maximizing the total delay.

Table 1 presents our results on 34 instances each of RoutingMin and RoutingMax. The sizes of the graphs in those instances range from 21 to 87 nodes. The Solved column gives the number of instances for which the named solver computed a result (which may or may not be optimal) within the timeout period, which was one minute. The Opt column gives the number of these instances for which the solver not only computed the optimal result, but also proved it to be optimal.

Since the sizes of the instances vary significantly, the minimum and maximum values of the total delay differ greatly as well. Averages of the delays are therefore not an appropriate representation of the overall quality of the solutions from a solver. Therefore we express the quality of each solution as a percentage of the maximum delay computed by any solver on the relevant problem instance. If all the solvers compute a delay of 0, we score all solvers as 0% for RoutingMin and as 100% for RoutingMax. The AvgPct column shows the average of these percentages for the

	cl+gr	flat	anger		gebser	
			o/a	prop	o/a	prop
WR (S/8)	894.49	10.22	166.74	95.39	456.71	7.58
WR (U/7)	24.14	8.89	41.20	28.44	53.06	1.29
GP (S/7)	9.69	2.88	656.69	4.13	659.27	2.86
GP (U/6)	43.02	0.95	221.38	9.76	243.08	3.54
CDS (S/7)	26.80	0.39	74.98	44.53	27.24	0.50
CDS (U/8)	1618.22	0.64	566.47	318.65	395.84	4.86
MG (S/15)	2.56	32.95	43.04	42.14	0.86	0.03

Table 2: ASP problems, geometric restart, no timeout

	cl+gr		flat	anger		gebser	
WR (S/8)	148.21	(5/1)	9.86	149.12	(4/1)	96.74	(5/1)
WR (U/7)	101.68	(5/1)	11.68	36.68	(0/0)	49.10	(0/0)
GP (S/7)	39.24	(0/0)	2.58	21.36	(0/0)	46.52	(0/0)
GP (U/6)	169.22	(5/1)	0.91	123.96	(3/1)	121.72	(0/0)
CDS (S/7)	185.09	(10/2)	0.41	102.79	(5/1)	127.88	(5/1)
CDS (U/8)	274.77	(10/2)	0.69	321.20	(20/4)	314.56	(20/4)
MG (S/15)	5.07	(0/0)	52.86	49.95	(5/1)	1.37	(0/0)

Table 3: ASP problems, Luby restart, with timeout

instances for which *all* the solvers get a solution. All the solvers were run with a slow restart strategy that used a Luby sequence (Luby et al. 1993) with a restart base of 400.

Due to the large domains involved, grounding is very inefficient, which causes cl+gr to run out of memory on all our test instances, even the smallest. For RoutingMin, all the solvers solve roughly the same number of instances, but anger and gebser get optimal solutions on almost three times as many instances, and the average quality of their solutions is also better by about a factor of 2. (For minimization, better performance is represented by smaller percentages, while for maximization, it is represented by larger ones.) For the instances of RoutingMax that all the solvers can solve, clingcon invariably generates the best solutions. However, clingcon generates solutions for substantially fewer instances than anger and gebser, showing that it is not as robust.

Our second set of benchmarks is a selection of problems taken from the second ASP competition [2]: Wire Routing (WR), Graph Partitioning (GP), Connected Dominating Set (CDS), and Maze Generation (MG). For each of these problems, we report on their satisfiable (S) and unsatisfiable (U) instances in separate rows of both Tables 2 and 3; the numbers after the forward slashes give the number of instances in each category. Table 2 shows results for a setup in which all the solvers were run using the default restart strategy of clasp (geometric restart, with the restart threshold starting at 100 conflicts, multiplied by 1.5 after each restart) and without time limits, while Table 3 shows the results when all the solvers were run with a Luby sequence restart strategy with restart base 10, and with a timeout of 10 minutes.

The numbers in the slots of Table 2 all represent an average runtime, in seconds, over all the problem instances represented by the row. The cl+gr column gives the average time taken by cl+gr to solve those instances. The flat column gives the average time taken to flatten the MiniZinc model to FlatZinc. The anger and gebser overall (o/a) columns give the average time taken to solve the resulting FlatZinc models using the anger and gebser variants of our implementation. The anger and gebser prop columns give the average times taken by our propagator within those overall solution times.

The numbers in time slots of Table 3 also represent average execution times; the execution time will be the timeout time (600 seconds) if the solver does not complete before then. Table 3 omits propagation times to make room for the numbers in parentheses after each average execution time. These represent respectively the number of runs on which the given solver failed to produce a solution before the timeout, and the number of instances to which those runs belong. (For example, 4/1 means that of the five runs on a problem instance, one produced a solution, but four did not.) Neither table has a column for clingcon, since the purpose of Tables 2 and 3 is to compare gebser and anger with a native ASP solver on pure Boolean problems. The results of running these problems on clingcon would be the same as the results of cl+gr, since these problems have nothing to do with the difference between cl+gr and clingcon, namely the finite domain extension present in clingcon.

The overall results in Table 2 are mixed. On WR/U, GP/S, GP/U and (due to flattening) MG/S, cl+gr clearly beats both anger and gebser. On WR/S and CDS/U, both anger and gebser clearly beat cl+gr. On CDS/S, cl+gr clearly beats anger, but edges out gebser by just a whisker. The overall winner on these tests is clearly cl+gr.

However, this picture changes when we switch our attention to Table 3. Even after including flattening time, gebser is faster than cl+gr on four of the seven problem sets (WR/S, WR/U, GP/U, CDS/S), and it is slower on only three (GP/S, CDS/U and MG/S). It is also more robust, failing to find a solution on only six problem instances, compared to seven for cl+gr. Our other system anger is less robust, failing to find solutions on eight problem instances, though for two of these, it did solve them on *some* runs. However, to compensate for this, anger is the fastest system on three problem sets (WR/U, GP/S, CDS/S).

The propagation time columns in Table 2 show that anger spends a lot more time on unfounded set propagation than gebser. In some cases, such as WR/S, this pays off handsomely, in the form of more effective pruning of the search space. In some other cases, such as CDS/U, anger spends less time outside the propagator than gebser, so anger seems to get better pruning, but not enough to pay back the extra cost of the propagator itself. And on most problems in Table 2, the extra cost of its propagator does not even help anger get better pruning. This suggests that we should investigate whether one can blend the two unfounded set calculation algorithms in order to achieve the pruning power of anger (Anger et al. 2006), or something close to it, at an efficiency closer to that of gebser (Gebser et al. 2012).

7 Related work

The closest modelling system to our approach is the IDP system (Wittocx et al. 2008*b*) which extends classical logic with the use of inductive definitions. Like our proposed extension of MiniZinc, IDP allows arbitrary first order formulae in the rule bodies of its inductive definitions while most ASP systems allow only normal rules. Unlike ASP solvers which apply the closed world assumption (an atom that cannot be derived is assumed to be false) to entire programs, our system and IDP can localize it to only a certain part of the program: default atoms for us, and *definitional atoms* for IDP. IDP handles constraints by

[2]See `http://dtai.cs.kuleuven.be/events/ASP-competition/SubmittedBenchmarks.shtml`

grounding, just as a traditional ASP system.

There has been significantly more effort in recent years to integrate CP into ASP systems than to integrate ASP into CP systems. The principal advantage of ASP over CP systems is the ability to use recursive definitions, particularly to model transitive closure. On the other hand, pure answer set solvers have a serious efficiency problem when dealing with problems that involve finite domain variables. This is why most research in this area has focused on integrating efficient finite domain handling in ASP systems, resulting in a new domain of research called *constraint answer set solving* (Drescher 2010).

The systems described by Baselice et al. (2005), Mellarkod & Gelfond (2008), Mellarkod et al. (2008) view both answer set and constraint solvers as black boxes, and their frameworks do not allow the incorporation of modern engineering techniques such as nogood learning and advanced backjumping. The clingcon system (Gebser et al. 2009), while it implements nogood learning and backjumping, still treats the CP solver as an *oracle* that does not explain its propagation to the ASP solver, and works around this shortcoming by using an indirect method to record nogoods generated by propagation done by the CP solver. All these systems have to incur some overheads for communication between the ASP and CP solvers.

The approach described by Drescher & Walsh (2010) avoids this overhead by translating the CP part of the problem into ASP rules, and achieves its efficiency through unit propagation on these rules; that paper also gives their translation of the *alldifferent* global constraint. One shortcoming of this approach is its reliance on an *a priori* ASP decomposition of global constraints; the example of the performance gains achieved by techniques such as lazy clause generation strongly suggests that such decompositions should be done lazily. Their more recent paper (Drescher & Walsh 2012) overcomes this shortcoming by allowing lazy nogood learning from external propagators. The resulting system is close to what we have implemented, and shows promising performance in comparison with clingcon. The mingo system described by Liu et al. (2012) does translation in the other direction: it translates an ASP program (extended with integer and real variables) to a mixed integer program.

The unique feature of constraint languages and solvers that distinguishes them from other declarative systems like ASP, SMT, and SAT is the use of global constraints, and the existence of extremely efficient propagators to solve these constraints. Other solvers usually rely on a single propagation method such as unit propagation. Specialized propagation techniques for global constraints, such as the one given by Schutt et al. (2011) for the *cumulative* constraint, allow much stronger and more efficient propagation than approaches using decomposition and unit propagation.

8 Conclusion

We have shown a method for adding answer set programming capabilities to the general purpose constraint programming language MiniZinc. The resulting system is much better at solving combined ASP/CP problems than existing systems, and we hope that our examples have convinced readers that such problems can be expressed more *naturally* in the syntax of MiniZinc than in the syntax of ASP languages. MiniZinc is also more flexible: it can express constraints on non-Boolean variables (such as integers, floats and sets); it can express complex Boolean expressions more naturally, *and* (with the exception of disjunctions in heads) it can express all ASP extensions, including weight constraints, choice rules, cardinality constraints, integrity constraints, and aggregates such as sum, count, min and max.

We have shown two implementations of our extensions to MiniZinc. Our benchmark results show that both these systems can solve combined ASP/CP problems that cannot be solved by cl+gr, even though its ASP component, clasp, won the last two competitions for pure ASP solvers. We have also shown that our system is competitive with clingcon, an extension of clasp, on such problems, being better on some tasks, worse on others.

We have several directions for future work. We will start by implementing our proposal for the inductive predicate syntax in MiniZinc, which should allow programmers to model problems more naturally, without manually grounding normal rules. We intend to investigate different ways to map these predicates to FlatZinc. We will look at the approaches used by the grounder of the IDP system (Wittocx et al. 2008*a*) and the transformation to ASP rules described by Mariën et al. (2004). Adopting some of these approaches may require moving from the stable model semantics to the well-founded semantics; the jury is still out on which users find more natural. We plan to investigate moving the grounding phase entirely to runtime, as proposed by the lazy grounding scheme of Palù et al. (2009) and the lazy model expansion scheme of De Cat et al. (2012). We also intend to look into incorporating other efficient features of ASP, such as preprocessing (Gebser et al. 2008). Finally, we intend to see whether we can construct an unfounded set algorithm that combines the efficiency of Gebser et al. (2012) with the more effective pruning of Anger et al. (2006).

Acknowledgments: NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy, and by the Australian Research Council. We are grateful to Geoffrey Chu for helping us to understand chuffed, to Guohua Liu for providing us with clingcon encodings of routing problems, and to the Potassco mailing list users for their timely replies to our queries about clingcon.

References

Anger, C., Gebser, M. & Schaub, T. (2006), Approaching the core of unfounded sets, *in* ‘Proceedings of the International Workshop on Nonmonotonic Reasoning’, pp. 58–66.

Baral, C. (2003), *Knowledge representation, reasoning and declarative problem solving*, Cambridge University Press.

Baselice, S., Bonatti, P. A. & Gelfond, M. (2005), Towards an integration of answer set and constraint solving, *in* ‘Proceedings of the 21st International Conference on Logic Programming’, Sitges, Spain, pp. 52–66.

Clark, K. L. (1978), Negation as failure, *in* H. Gallaire & J. Minker, eds, ‘Logic and Data Bases’, Plenum Press, NY, pp. 127–138.

De Cat, B., Denecker, M. & Stuckey, P. (2012), Lazy model expansion by incremental grounding, *in* ‘Technical Communications of the 28th International Conference on Logic Programming’, Budapest, Hungary.

Dooms, G., Deville, Y. & Dupont, P. (2005), CP(graph): introducing a graph computation domain in constraint programming, *in* ‘In proceedings of the 11th International Conference on Principles and Practice of Constraint Programming’, Sitges, Spain, pp. 211–225.

Dowling, W. & Gallier, J. (1984), ‘Linear time algorithms for testing the satisfiability of propositional Horn formulae’, *Journal of Logic Programming* **1**, 267–284.

Drescher, C. (2010), Constraint answer set programming systems, *in* ‘Technical Communications of the 26th International Conference on Logic Programming’, Dagstuhl, Germany, pp. 255–264.

Drescher, C. & Walsh, T. (2010), ‘A translational approach to constraint answer set solving’, *CoRR* **abs/1007.4114**.

Drescher, C. & Walsh, T. (2012), Answer set solving with lazy nogood neneration, *in* ‘Technical Communications of the 28th International Conference on Logic Programming’, Budapest, Hungary.

Feydy, T. & Stuckey, P. J. (2009), Lazy clause generation reengineered, *in* ‘Proceedings of the 15th International Conference on the Principles and Practice of Constraint Programming’, Lisbon, Portugal, pp. 352–366.

Gebser, M., Kaufmann, B., Neumann, A. & Schaub, T. (2007), Conflict-driven answer set solving, *in* ‘Proceedings of the 20th International Joint Conference on Artificial Intelligence’, Hyderabad, India, p. 386.

Gebser, M., Kaufmann, B., Neumann, A. & Schaub, T. (2008), Advanced preprocessing for answer set solving, *in* ‘Proceedings of the 18th European Conference on Artificial Intelligence’, Patras, Greece, pp. 15–19.

Gebser, M., Kaufmann, B. & Schaub, T. (2012), ‘Conflict-driven answer set solving: From theory to practice’, *Artif. Intell* **187**, 52–89.

Gebser, M., Ostrowski, M. & Schaub, T. (2009), Constraint answer set solving, *in* ‘Proceedings of the 25th 25th International Conference on Logic Programming’, Pasadena, CA, pp. 235–249.

Gebser, M. & Schaub, T. (2005), Loops: Relevant or redundant?, *in* ‘LPNMR’, pp. 53–65.

Gelfond, M. & Lifschitz, V. (1988), The stable model semantics for logic programming, *in* ‘ICLP/SLP’, pp. 1070–1080.

Lifschitz & Razborov (2006), ‘Why are there so many loop formulas?’, *ACM Transactions on Computational Logic* **7**(2), 261–268.

Lin, F. & Zhao, Y. (2004), ‘ASSAT: computing answer sets of a logic program by SAT solvers’, *Artificial Intelligence* **157**(1-2), 115–137.

Liu, G., Janhunen, T. & Niemela, I. (2012), Answer set programming via mixed integer programming, *in* ‘Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning’, pp. 32–42.

Lloyd, J. W. & Topor, R. W. (1984), ‘Making Prolog more expressive’, *Journal of Logic Programming* **1**(3), 225–240.

Luby, M., Sinclair, A. & Zuckerman, D. (1993), ‘Optimal speedup of Las Vegas algorithms’, *Information Processing Letters* **47**, 173–180.

Mariën, M., Gilis, D. & Denecker, M. (2004), On the relation between ID-logic and answer set programming, *in* ‘JELIA’, pp. 108–120.

Mellarkod, V. S. & Gelfond, M. (2008), Integrating answer set reasoning with constraint solving techniques, *in* ‘Proceedings of the 9th International Symposium on Functional and Logic Programming’, Ise, Japan, pp. 15–31.

Mellarkod, V. S., Gelfond, M. & Zhang, Y. (2008), ‘Integrating answer set programming and constraint logic programming’, *Ann. Math. Artif. Intell* **53**(1-4), 251–287.

Mitchell, D. G. (2005), ‘A SAT solver primer’, *Bulletin of the EATCS* **85**, 112–132.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J. & Tack, G. (2007), MiniZinc: Towards a standard CP modelling language, *in* ‘Proceedings of the 13th International Conference on the Principles and Practice of Constraint Programming’, Providence, RI, pp. 529–543.

Ohrimenko, O., Stuckey, P. J. & Codish, M. (2009), ‘Propagation via lazy clause generation’, *Constraints* **14**(3), 357–391.

Palù, A. D., Dovier, A., Pontelli, E. & Rossi, G. (2009), Answer set programming with constraints using lazy grounding, *in* ‘ICLP’, pp. 115–129.

Rossi, F., Beek, P. v. & Walsh, T. (2006), *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, Elsevier Science, New York, NY.

Schulte, C. & Stuckey, P. (2008), ‘Efficient constraint propagation engines’, *ACM Transactions on Programming Languages and Systems* **31**(1), Article No. 2.

Schutt, A., Feydy, T., Stuckey, P. J. & Wallace, M. G. (2011), ‘Explaining the cumulative propagator’, *Constraints* **16**(3), 250–282.

Simons, P., Niemelä, I. & Soininen, T. (2002), ‘Extending and implementing the stable model semantics’, *Artificial Intelligence* **138**(1–2), 181–234.

Van Gelder, A., Ross, K. A. & Schlipf, J. S. (1988), Unfounded sets and well-founded semantics for general logic programs, *in* ‘Proceedings of the ACM Symposium on Principles of Database Systems’, pp. 221–230.

Viegas, R. D. & Azevedo, F. (2007), GRASPER, *in* ‘Proceedings of the 13th Portuguese Conference on Artificial Intelligence’, Guimarães, Portugal, pp. 633–644.

Wittocx, J., Mariën, M. & Denecker, M. (2008*a*), GIDL: A Grounder for FO, *in* ‘Proceedings of the Twelfth International Workshop on Non-Monotonic Reasoning’, pp. 189–198.

Wittocx, J., Mariën, M. & Denecker, M. (2008*b*), The IDP system: a model expansion system for an extension of classical logic, *in* ‘LaSh’, pp. 153–165.

Practical Compression for Multi-Alignment Genomic Files

Rodrigo Cánovas **Alistair Moffat**

NICTA Victoria Research Laboratory
Department of Computing and Information Systems
The University of Melbourne
Victoria 3010

Abstract

Genomic sequence data is being generated in massive quantities, and must be stored in compressed form. Here we examine the combined challenge of storing such data compactly, yet providing bioinformatics researchers with the ability to extract particular regions of interest without needing to fully decompress multi-gigabyte data collections. We focus on data produced in SAM format, which is particularly voluminous in nature, and describe storage techniques that have the desired blend of attributes.

Keywords: Genomic data, lossless compression, lossy compression, SAM format.

1 Introduction

Next generation sequencing machines produce vast amounts of genomic data (Ansorge, 2009). This data is valuable for the insights it allows now into the health of individuals and whole populations, and will continue to be of benefit into the future as medical knowledge grows. But for genomic data to be long-term useful, it must be stored. And with output files in the gigabyte range now being generated within an hour or less of technician time, and at a cost of just a few hundred dollars, the mechanics of storing them – and retrieving information out of them when it is required – is a challenge. Bioinformatics researchers are increasingly regarding big data storage facilities as being fundamentally necessary to their operations.

In this paper we consider data stored in SAM (*S*equence *A*lignment *M*ap) format files (Li et al., 2009). These files can contain millions of *reads*, each produced as a continuous fragment of data extracted from the processing of a single genome, represented as a string of *bases*, letters that indicate the fundamental molecules of DNA. A number of meta-data fields are associated with each read to form an *alignment read*, and some of these fields are as expensive to store as the sequence of bases. Because of the multiplicity of alignment reads extracted from each genome, the repetition in the meta-data elements, and the fact that they are stored as printable ASCII text, there is considerable redundancy in SAM files. Our purpose in this project is to identify and exploit that redundancy, and develop a new compressed representation for SAM-style genomic data that is both economical of space and readily queryable, so that data about particular alignments can be extracted in isolation, without requiring whole files to be decompressed. The latter option is of considerable benefit to researchers working with SAM data, who rarely wish to fully decompress archived data – and indeed, may not have the time or space resources required to do so.

The next section provides a general overview of compression methodologies, including a mode we refer to as being *information preserving* that sits between the conventional lossless and lossy approaches. Section 3 then introduces the SAM format that is used to store multi-alignment genomic data. Compression mechanisms suitable for the various SAM fields are examined in Section 4, including measurement of their effectiveness on several typical files. The issue of querying SAM files is then examined in Section 5. Section 6 presents related work, and then Section 7 concludes our presentations.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

2 Compression technologies

This section summarizes several issues relevant to the design of compression techniques. For detailed coverage of these topics, see, for example, Bell et al. (1990), Moffat and Turpin (2002) and Navarro and Mäkinen (2007).

Lossless and lossy compression

Compression techniques can be categorized as belonging to one of two distinct classes: *lossless*, or exact compression; and *lossy* compression. Lossy compression methods are typically applied to data originally sampled from continuous domains, and are based on the recognition that the process of turning that data into digital form can, within limits, be further approximated to save space. For example, digital cameras take images that can be stored in either `.jpg` form (lossy compression) or as `.raw` files (larger uncompressed files). But even the `.raw` file is a quantized approximation of the original scene, and its attractiveness to photography purists is not that it contains *no* loss of fidelity, but only that it contains no *additional* loss of fidelity. In most applications and environments a viewer will not perceive any difference between the two formats.

On the other hand, data which is fundamentally discrete and non-continuous, such as ASCII text, is almost always represented using lossless methods (although it is also worth noting that from time to time the observation has been made that human readers can still make sense of some lossy representations of text).

Information-preserving approaches

There is a second way in which lossy compression concepts might be useful for some data sources, which we

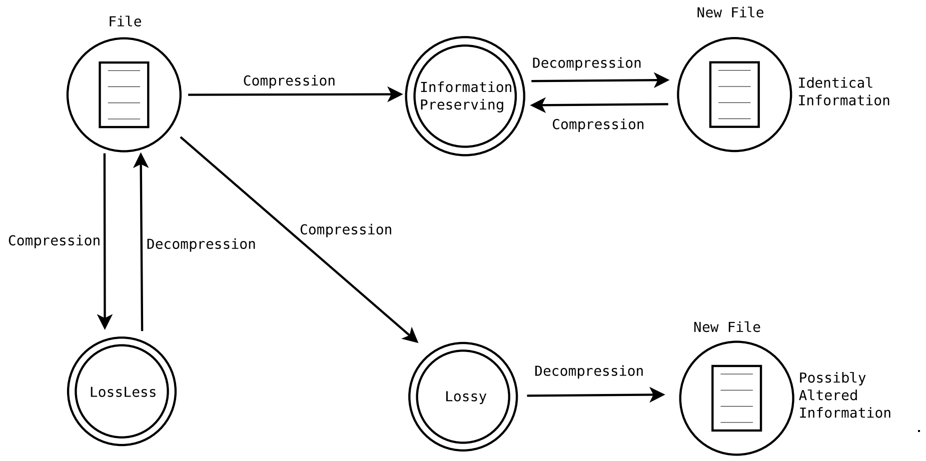


Figure 1: Three alternative compression modalities.

introduce by way of an example. Suppose a list of n numbers each in the range $0 \ldots (m-1)$ is generated by some process, in no particular order. These numbers might be stored in ASCII in a file of at most $n(\lceil \log_{10} m) \rceil + 1)$ bytes (allowing $+1$ for a newline character after each value), or, if a minimal binary representation is used, in a file of $n \lceil \log_2 m \rceil$ bits. If the purpose to which the data will be put is unknown, and no further indication is provided in regard to the distribution of the numbers within the specified range, then the binary form is an efficient one. But suppose the downstream application that uses the set of numbers places no importance on the order in which the numbers are received. If so, then the set of numbers can be sorted, differences between those numbers taken, and the differences coded using $n(2 + \lceil \log_2((n+m)/n) \rceil)$ bits (Moffat and Turpin, 2002), which might be a non-trivial saving. That is, if we regard the *order* in which the data is presented as being of no importance, then shifting the input into a particular arrangement might allow better compression. In such cases, the decompressed output will also be in that order, and so the original input file cannot be exactly regenerated, in the sense that the Unix `diff` and `cmp` commands will certainly report a discrepancy. Nevertheless, there might also be a sense in which all of the actual information embedded in the source data has been preserved, even if the physical representation has not. We call such representations *information preserving*, and regard them as being a third possible compression modality, neither lossless nor lossy.

Figure 1 illustrates these notions. A lossless compression mechanism must exactly recreate the input file, in all syntactic detail, as indicated by the double arrow. An information preserving regime will not be able to reproduce the original file (denoted by a single arrow), but once it has been decompressed a first time into its new form, it can be recompressed and decompressed a second time without further change taking place (the double arrow at the top of the diagram). That is, a representational fixed-point is reached after one compress/decompress cycle. A lossy scheme cannot reproduce the input file; nor is there even any guarantee that a second iteration of compression and decompression will achieve the same file.

Information preserving techniques have also been proposed in other application areas. For example, consider the area of program source code compression. In a syntax-aware compressor for a language such as C or Pascal, white-space tokens can be normalized and other changes made that do not alter the compilability or correctness of the program, and only affect how it looks in an editor.

It may also possible to augment an information preserving compression regime with additional information so as to adjust its output to recreate the exact input. That is, losslessness might be an optional enhancement, at the cost of storing further data. In the case of program source code compression, the auxiliary information would specify the exact whitespace token to be inserted at each location from which it had been stripped. In the case of the numeric example used at the beginning of this discussion, a permutation index costing $n \lceil \log_2 n \rceil$ bits would be required, which is a relatively high cost. In the SAM-format compression proposed in the next section, the items being permuted are long lines of text, and the overhead cost of storing the required permutation vector is small.

Modeling and coding

It is also recognized that compression should be thought of as consisting of two complementary activities. *Modeling* is the process of inferring structure from the data that is presented, and, for each type of symbol or type of context, estimating (either explicitly or implicitly) a probability distribution that covers the set of options that might occur next. Those probabilities are then – again, either explicitly or implicitly – used to drive a *coding* step, in which the actual symbol that appears is represented into the output bitstream, taking into account the probability estimates generated by the model. One early example of such a structure is given the blend of model and coder sometimes referred to as *Huffman coding*, in which symbol probabilities are estimated using a zero-order Markov model counting their occurrence frequencies in the text in question; and then the symbols in the sequence are coded via a bit-aligned minimum-redundancy code.

Static, semi-static, and adaptive

A third characterization of compression techniques is whether they make use of *static*, *semi-static*, or *adaptive* probability estimations (and hence coders). In a static regime, probability estimates are independent of any particular input file, and are constant. In the semi-static approach, the probability estimates are based on the data file being represented, and are established in a preliminary pass through the data and then sent as the first component of the compressed message. In an adaptive system the probability estimates are constructed on-the-fly, based on the part of the message that has already been processed, and if necessary, allowing for the possibility of previously unencountered symbols to be added as they arise.

Static and semi-static codes tend to allow faster decoding, because there is no need for the decoder to track the

Field	Type	Description
QNAME	string	Query template name
FLAG	int	Bitwise flags (values between 1 and 1160) that give properties of the alignment, including if the sequence is a reverse complement
RNAME	string	Reference sequence name
POS	int	Leftmost mapping position of the first matching base
MAPQ	int	Mapping quality: $-10 \log_{10} Pr(\text{mapping position is wrong})$
CIGAR	string	The CIGAR string
RNEXT	string	Reference sequence name of the next segment in the template
PNEXT	int	Position of the next segment in the template
TLEN	int	Signed observed template length
SEQ	string	Sequence of nucleotides bases of the read used in the alignment
QUAL	string	Estimated error probability of each base: $-10 \log_{10} Pr(\text{base is wrong}) + 33$
OTHER	string	Optional fields of the form TAG:TYPE:VALUE

Table 1: The twelve fields recorded for each read in a SAM file. The first eleven are required, but may be replaced with `*` for strings and `0` for numeric fields if data is not known or is not being stored. Further details are provided by Li et al. (2009).

Letter	Value	Probability
`(`	40	20%
`7`	55	0.6%
`F`	70	0.02%
`U`	85	0.0006%
`d`	100	0.00002%

Table 2: Examples of values stored in the QUAL field. The ASCII letters represent probabilities of error in the corresponding base according the relationship $value = -10 \log_{10} Pr(\text{error}) + 33$. The error probabilities are computed by the sequencing hardware.

probability changes. Static and semi-static methods also make it easier to provide random-access into the compressed file. In particular, if a bit pointer is provided into the compressed package, coding can be resumed from that location provided the context is understood.

3 Genomic data formats

Genomes are typically described by (usually long) sequences of identifying letters, one per base-pair of the original. In simplest form, the letters are the four acronyms of the fundamental bases, `A`, `C`, `G`, and `T`. Some formats (including SAM) add other letters, such as `N`, for unknown bases; and some formats further extend the alphabet to include specific identifying letters for other proteins that might be present. The common thread in all formats is the small alphabet that is employed (between four and around twenty symbols), and the dominance of the four key symbols.

SAM format

In the SAM format, each sequence of bases is accompanied by eleven other fields that add considerably to the total stored size (Li et al., 2009). These fields are shown in Table 1. All of them are required, in the sense that they cannot be omitted; but it is also common for them to be stored as place-holder `0` and/or `*` values.

```
                                    1
              1 2 3 4 5 6 7 8     9 0 1 2 3 4 5 6
Sequence 1:   T T A G A T A A * * G A T A G C T G

Sequence 2:   T T A G A T A A A G G A T A * C T G

     CIGAR:   8M 2I 4M 1D 3M
```

Figure 3: Example of CIGAR analysis. The positions marked with `*` are indicative only, and not present in either of the two sequences.

The field labeled SEQ is the sequence of bases corresponding to this read; the other critical field from a data storage point of view is QUAL, which is the same length as the SEQ field, and also contains an ASCII letter for each sequence position. The value stored in each QUAL field is an estimate of the correctness of the corresponding SEQ field. The mathematical relationship between estimated probability of error and value stored in QUAL is shown in Table 1; and Table 2 provides some examples. The QUAL sequence can be thought of as a quantization of a underlying phenomena that is numeric and continuous, and hence a candidate for possible use of lossy compression. Figure 2 shows a sample read containing 25 bases in the SEQ string, with the accompanying QUAL string indicating that each base after the first has an error probability of well under 0.05%.

Each of the reads may be referenced against an external resource described by the RNAME field, which can be thought of as a reference identifier indicating the external location of a related resource. If an alignment has been computed for this read relative to that resource, then the POS field indicates the offset within the resource at which the alignment commences.

Overall, a SAM file consists of a header block describing attributes of the sample as a whole, such as meta-data describing the experimental environment and regime; followed by thousands or millions of relatively short reads – each perhaps 30 to 120 bases long – derived from a single experimental run. Hence, it is not unusual for the same RNAME to turn up many times in the SAM file, and nor is it in any way unusual for the reads to overlap, in the sense of the identified alignment for one of them being within the range of the identified alignment of another.

In some cases, the read represented by the SEQ string has not only been aligned against the RNAME string, it is also represented relative to it as a sequence of edit instructions. If so, the corresponding CIGAR field (*C*ompact *I*diosyncratic *G*apped *A*lignment *R*eport) is non-empty. If it is present, the CIGAR string consists of a sequence of instructions: `M` atch the next ℓ characters; `D` elete the next ℓ characters; or `I` nsert a group of ℓ character. Figure 3 gives an example showing two similar reads, and a CIGAR string that describes their (relative to each other) structure.

Fields that are absent are represented by `*` and/or `0` characters. The file itself is tab-delimited between fields, and newline delimited between read alignments.

BAM format

The SAMtools software suite[1] provides other storage options. A second standard representation is known as BAM format, in which blocks of text from a SAM file are stored compressed using a modified `zlib` library. Compared to the original SAM file, a BAM compressed version can be expected to occupy around 30% of the original size, with an auxiliary index that allows limited random access to the reads in order to support queries. The BAM representation uses the BGZF (Blocked GNU Zip Format) compression

[1] See `http://samtools.sourceforge.net/`.

```
SEQ   :   C  T  G  A  A  C  T  T  A  G  G  C  T  C  A  G  C  C  T  C  A  G  T  A  A

QUAL  :   <  F  E  F  G  E  I  G  G  H  H  H  J  H  I  I  I  I  J  H  J  I  G  H  H

Value :  60 70 69 70 71 69 73 71 71 72 72 72 74 72 73 73 73 73 74 72 74 73 71 72 72
```

Figure 2: Example of the SEQ and QUAL components for one read within a SAM file.

	HG00113	HG00559	Local
Alignments	5,630,211	2,515,117	8,095,516
Av. read (bases)	90	108	100
Size (MB)	2,220.5	1,121.6	1,965.7
BAM (MB)	457.8	285.1	762.9
Gzip (MB)	429.2	260.7	748.2

Table 3: Statistics for three SAM files. File HG00113 refers to HG00113.chrom11.ILLUMINA.bwa.GBR.exome.20111114; File HG00559 is HG00559.chrom20.ILLUMINA.bwa.CHS.lowcoverage.20111114; and file Local was supplied by a local researcher based on their own bioinformatics work.

Field	Raw		Gzip	
	MB	%	MB	%
QUAL	488.6	22.0%	239.6	66.5%
SEQ	488.6	22.0%	38.2	10.6%
OTHER	933.5	42.0%	27.0	7.5%
QNAME	100.9	4.5%	20.1	5.6%
PNEXT	49.0	2.2%	12.2	3.4%
TLEN	24.0	1.1%	8.7	2.4%
POS	49.0	2.2%	8.2	2.3%
Total	2220.5	100.0%	360.4	100.0%

Table 4: Percentage required by various SAM fields of HG00113, before and after compression using gzip in a striped manner, ordered by decreasing compressed contribution to the total, and with six smaller fields omitted. The striped and compressed representation totals 360.4 MB.

format, which adds an access structure on top of the standard gzip file format.

As is shown in Table 3, BAM conversion results in a stored file that is a little larger than can be attained by gzip alone. The difference is largely caused by the additional BAM index, which links positions in the reference sequence with reads in the blocks of the BAM file.

4 Compressing SAM components

To evaluate alternative compression methodologies, three SAM files are used. Some attributes of the three sample files are summarized in Table 3. In the remainder of the text they are referred to by their abbreviated names HG00113, HG00559, and Local.

Striping

One well-known mechanism for compressing data stored in structured formats like SAM is to *stripe* the data into separate streams, and then use a general-purpose compressor – such as gzip – on the concatenated contents of each stream. Decompression regenerates the various streams, and they can be re-interleaved to recreate the original file. Striping is effective if the fields are distinctive in nature, and/or contain vertical repetitions. Table 4 shows the raw cost of some of the striped components of the test file HG00113, as a percentage of the file size, before and after the components are compressed by gzip.

Notable in the table is that the QUAL and SEQ fields are equal in size prior to application of a standard compression regime, but that the SEQ field is far more compressible than the QUAL field, and that the latter dominates the compressed representation. The high relative compressibility of the SEQ field is a consequence of the fact that it is over a very small alphabet; and that overlapping reads are likely to contain common subsequences that can be identified by the string match-based gzip compression mechanism. The OTHER field is also highly compressible, and moves from being the dominant cost in the uncompressed version of the file, to being less than 8% of the striped compressed file. Note that the percentages in Table 4 are relative to the sizes of the two original files. Overall compression rates for particular components can be estimated from the figures supplied. For example, on file HG0013 the SEQ field drops from being 22.0% of 2,200 MB to being 10.6% of 360.4 MB, an overall saving of more than 445 MB, and a reduction to around just 8% of the original SEQ requirement.

On the other hand, the QUAL field has both a larger alphabet and less repetition, because the estimated error is a function of many factors, and is only loosely correlated with its position in the underlying genome. It requires fully two-thirds of the striped compressed representation.

The SEQ field

We now focus on the SEQ field, and consider if there are additional storage savings possible.

One of the reasons that gzip obtains such good compression on SEQ components is the large number of repeated subsequences, which arise because of the process used to generate SAM files. The biological source material contains many copies of the underlying genetic sequences. When cut randomly into segments, each particular nucleotide in the original appears in any number of the reads that are reproduced into the SAM file. Within the SAM file each of these reads might partially or fully overlap with other reads in the file, or might be unique. Moreover, when reads do overlap, they are likely to be highly similar. If the process used to generate them were infallible, and if there were no mutations in any of the cells in the source material, the reads at any sequence location should be in perfect agreement. But there are discrepancies introduced by the inexactness of the process and machinery used; by the possibility that some of the reads arise from mutated cells; and by computation errors made when the alignments are identified.

Even so, there can be a high degree of repetition across the multiple reads that span any particular location in the genome, and even though it is a general-purpose text compression program rather than one tailored to DNA sequences, gzip does a good job of identifying and exploiting the common subsequences.

An obvious question is whether a tailored compression regime might do better. In particular, there is additional information associated with each read that could be used to explicitly identify the set of reads that are believed to have overlapping SEQ fields. It is not mandatory for SAM records to include a meaningful RNAME – like many of the fields listed in Table 1, it can be stored as a * to indicate "not present". But when it is present, it indicates which

	HG00113	HG00559	Local
Different RNAMEs	1	1	68
Overlapping reads (%)	97.1	99.9	80.8
Overlapping bases (%)	94.4	98.6	75.2
Median multiplicity	102	6	16

Table 5: Statistics for the three SAM files in terms of overlaps relative to the given reference sequences. The final row shows the median, taken over the set of all bases present in the file, of the number of bases that share the same offset in regard to the same RNAME sequence, counting one for bases that appear in read alignments with no RNAME specified.

reference sequence the SEQ field is like, and the numeric POS field indicates an offset relative to that reference sequence. When these two fields are available it is thus possible for an encoder to permute the records in the SAM so that all of the alignments that relate to a given reference sequence are placed in a cluster of consecutive records, and also for them to be ordered by POS within that cluster.

If the encoder permutes the records within the SAM file, there are then two options for decompression. The first is for a permutation vector to be added to the compressed representation, so that the decoder is able to invert the permutation and restore the original ordering. As noted in Section 2, if there are n records, then a total of $n\lceil\log_2 n\rceil$ bit suffices for this purpose. If the downstream applications do not require that the original SAM file ordering be retained – indeed, if there is no importance of any sort associated with the original ordering of the records, and their arrangement was an arbitrary artifact of the process that generated them – then there is no need to store the permutation vector, and the compression regime can be *information preserving* rather than lossless.

Table 5 shows the extent of the read overlaps in the three example files. Only the file Local has less than around 95% or more overlaps in terms of both reads and individual bases. It is lower is because no RNAME field is supplied for 18% of the reads, and hence it is not possible to identify overlaps for those SEQ components.

If it could be assumed that the set of reference sequences used in each SAM file was available to the compressor and decompresser as a static external resource, then each of the reads in the SAM file could be compressed relative to it. But this would be a risky assumption, and would mean that the compressed SAM file could not be regarded as being self-contained.

Presumed Reference Sequence

Instead, we construct what we call a *presumed reference sequence*, or PRS, that is specific to the SAM file in question, and does not require linkage to any external resources. Figure 4 shows how this is done, using as an example four reads with slightly different POS fields and the same RNAME field. First, the set of reads in the SAM file are ordered by the RNAME field, and then by the POS field specified for each one. Where there is overlap, the reads are aggregated by a simple majority vote to form a presumed reference sequence.

In Figure 4(a) it is supposed that four reads each of $20+$ bases are slightly offset from each other, and are the only four reads that span a section of the REF sequence. The presumed reference sequence is shown at the top, and is in complete agreement with the four reads in all but 11 (out of a total of 97) of the base positions, as shown in Figure 4(b). (For reasons that are explained shortly, the N in read four is not permitted to install an N into the PRS.) To encode these four reads, the PRS string spanning 38 bases is stored, then four sets of "offset, length, exceptions" information, one per read, detailing: the commencement within the PRS of that read (which can be inferred from the POS field); its length (which is usually constant throughout the whole SAM file); and a list of locations in the read where bases other than is stored in the PRS are to be inserted.

Representing exceptions

Figure 5 gives more details of the process used to encode the reads via copies from the PRS and a list of exceptions.

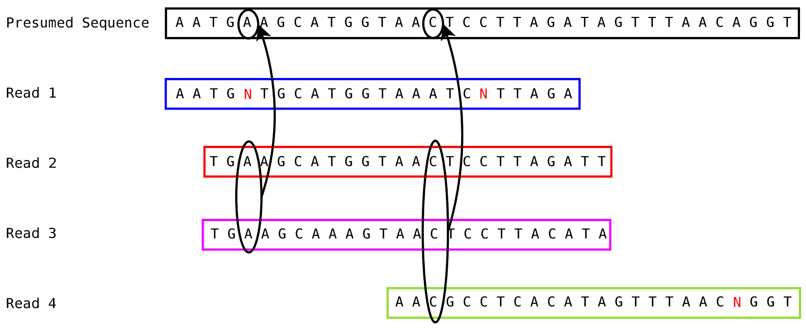
If the compressed SAM file is to be self-contained, the first component to be stored must be the PRS. It is a string of bases, typically over an alphabet of size $\sigma = 4$, covering symbols A, C, G, and T with their usual meanings, and requiring two bits per base to economically encode them. Note that N, the symbol used to indicate "unknown" symbols, is not permitted in the PRS. If it is the majority symbol – as is the case with the N in Read 4, it is replaced in the PRS by any other symbol.

Each of the reads relative to the PRS is stored as an offset relative to the previous read's POS; plus a length; plus a set of instructions from which the read can be reconstructed. To achieve the third component, each read is decomposed into alternating "copy" and "replace" counts, illustrated in the lower part of Figure 5. Because the exceptions are only required when a base differs from the one stored in the corresponding position in the PRS, it is beneficial to further split the stream of exceptions into four parts, denoted in the figure as "not A", "not C", and so on. For example, in Figure 5 the exception in Read 3 consists of the two bases AA. The PRS contains TG at the corresponding positions, so the first A is stored in the "not T" subsequence, and the second in the "not G" subsequence.

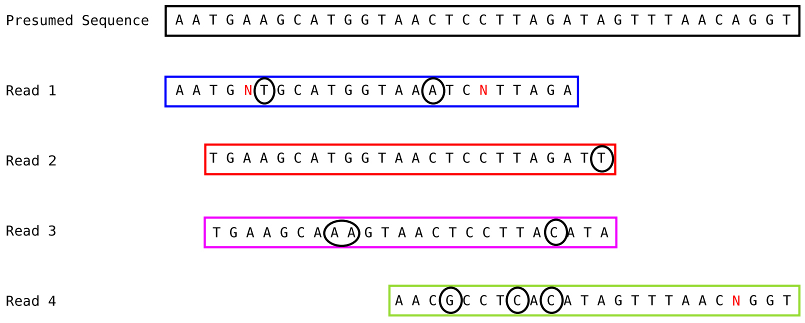
These nine elemental components are striped across nine arrays that collectively allow the set of reads to be reconstructed, provided that the PRS is also available. Each of the arrays can be thought of as being a set of integers with a specialized purpose and localized distribution pattern. For example, the values in the "Replace" array will typically be much smaller than the values in the "Copy" array, and should be stored using a different encoding.

The decision to avoid N values in the PRS is a consequence of their low frequency in the SEQ sequence. If N symbols were allowed in the PRS, the alphabet used to represent it has $\sigma = 5$ symbols. On the other hand, if the PRS is restricted to the standard $\sigma = 4$ bases, each can be represented directly using two bits. Moreover, because the PRS is an internally-stored aid to compression rather than an expected output of the process, it can, if it simplifies processing or saves space, be approximated. Hence, explicitly preventing N values does not damage the correctness of the arrangement, since whatever is in that position in the overlapping reads can be coded as an exception to the base that is arbitrarily used to replace the N.

In addition, because N is such a rare symbol, it is also helpful to code it differently when it appears in the four "not" sequences. Where an N appears in any of the overlapping reads, it is replaced temporarily by whatever symbol appears in that position in the PRS, and represented as a simple copy (or as part of a longer copy). To undo this deliberate simplification, an overall list of locations in the reads that *are* N is also maintained. This list is consulted as the final step in the decoding process, and any occurrences of N within the designated range of positions are reinstated into the output SEQ sequence before it is written. This approach implies a slight redundancy. But N symbols are relatively rare, and the cost of doing it this way is far less than the overhead cost of working with $\sigma = 5$ when coding the PRS, and of working with $\sigma = 4$ when coding the four "not" sequences.


Presumed Sequence
A A T G A A G C A T G G T A A C T C C T T A G A T A G T T T A A C A G G T
Read 1
A A T G N T G C A T G G T A A A T C N T T A G A
Read 2
T G A A G C A T G G T A A C T C C T T A G A T T
Read 3
T G A A G C A A A G T A A C T C C T T A C A T A
Read 4
A A C G C C T C A C A T A G T T T A A C N G G T

(a) Constructing the presumed reference sequence by majority vote.


Presumed Sequence
A A T G A A G C A T G G T A A C T C C T T A G A T A G T T T A A C A G G T
Read 1
A A T G N T G C A T G G T A A A T C N T T A G A
Read 2
T G A A G C A T G G T A A C T C C T T A G A T T
Read 3
T G A A G C A A A G T A A C T C C T T A C A T A
Read 4
A A C G C C T C A C A T A G T T T A A C N G G T

(b) Identifying exceptions to the presumed reference sequence.

Figure 4: Construction of a presumed reference sequence by taking the majority opinion of the overlapping reads at each position: (a) a set of overlapping reads, with N symbols considered to be non-voting; and (b) the locations in those reads at which discrepancies occur, again ignoring any Ns.

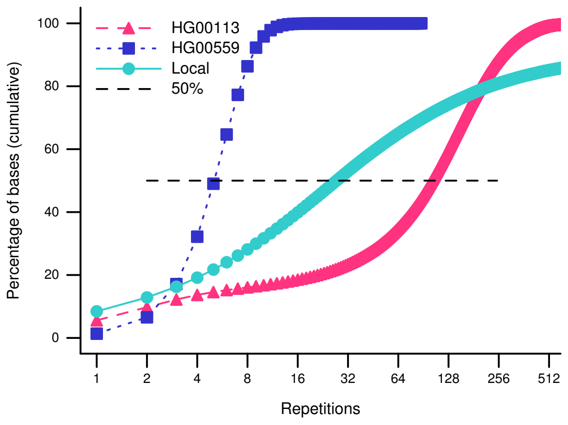


Figure 6: Cumulative plot of the fraction of bases in the SAM file as a function of the number of overlapping reads each base belongs to.

Figure 6 shows the extent to which bases overlap. To generate the three curves, the total set of bases in each file was ordered according to the number of other bases that were coincident with this one. For example, less than a quarter of the total number of bases in each of the three SAM files appeared as the only one aligned with that particular position in the RNAME sequence. More importantly, 50% of the bases share their position with 5 or more other bases in HG00559, with 15 or more other bases in Local, and with more than 100 other bases in HG00113. The number of overlaps arising from the multiplicity of reads is substantial.

Table 6 brings the various components together. Each of the data types comprising the compressed SEQ stream is shown, together with the number of instances of that type of object. The cost of storing each component using a suitable static code is also shown. For example, to code the "not" sequences, each of which consists of symbols over an alphabet of size $\sigma = 3$, the three binary codewords 0, 10, and 11 are used, with an average cost of not more than 1.67 bits per symbol, provided only the most frequent of the three alternatives is assigned the one-bit codeword. Similarly, a range of binary codes and Elias γ codes (see Moffat and Turpin (2002) for details) are used for the other components. The critical change that has been achieved compared to the gzip approach is that none of the values in Table 6 are based on adaptive (or even semi-static) models or codes.

As was already noted in connection with Table 5, the third of the data files, Local, contain a significant fraction of reads that are not associated with an identified reference sequence. These read alignments are coded as if they were non-overlapping, that is, as bases over an alphabet of size $\sigma = 4$ symbols, with the N symbols reinstated subsequently, and no use made of a PRS. Those costs are shown in the bottom part of the table.

Summed over the various components, Table 6 shows that the deconstructed SEQ stream can be represented in space that is always at least a little less than is required by gzip equivalent (note that the representations in Table 6 also absorb the separate POS field, stored as the offset), and on file Local is about half of that space. More importantly, the proposed approach is structured in a manner that has considerably more flexibility than gzip in terms of access options, because it is based entirely around static models and codes. Access operations on SAM-format data are discussed shortly, in Section 5.

The QUAL field

Genomic data is discrete rather than sampled-continuous, and hence, at face value, not amenable to lossy compres-

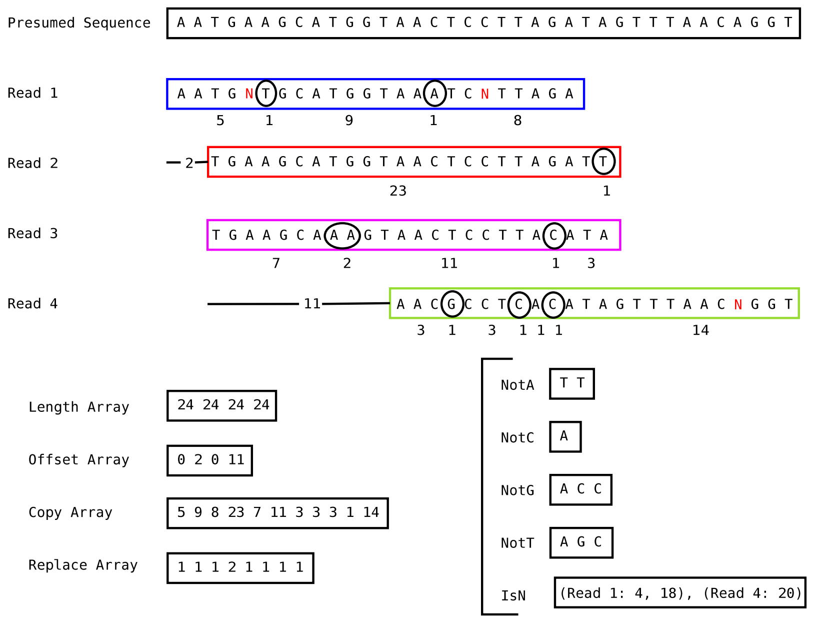

Presumed Sequence
A A T G A A G C A T G G T A A C T C C T T A G A T A G T T T A A C A G G T
Read 1
A A T G N T G C A T G G T A A A T C N T T A G A
5 1 9 1 8
Read 2
2
T G A A G C A T G G T A A C T C C T T A G A T T
23 1
Read 3
T G A A G C A A A G T A A C T C C T T A C A T A
7 2 11 1 3
Read 4
11
A A C G C C T C A C A T A G T T T A A C N G G T
3 1 3 1 1 1 14
Length Array
24 24 24 24
Offset Array
0 2 0 11
Copy Array
5 9 8 23 7 11 3 3 3 1 14
Replace Array
1 1 1 2 1 1 1 1
NotA
T T
NotC
A
NotG
A C C
NotT
A G C
IsN
(Read 1: 4, 18), (Read 4: 20)

Figure 5: Representing a group of SEQ fields as independent components relative to a presumed reference sequence.

Component	Code	HG00113		HG00559		Local	
		Number	Size (MB)	Number	Size (MB)	Number	Size (MB)
Reads with an RNAME *field supplied*							
PRS	binary(4)	52,641,432	12.55	58,432,811	13.93	106,253,672	25.33
Length array	constant	5,630,221	0.00	2,515,117	0.00	8,095,516	0.00
Offsets	Golomb	5,630,221	2.01	2,515,117	1.80	6,651,885	3.17
Copies	Elias γ	18,926,297	11.54	14,678,161	6.87	24,642,246	16.88
Replacements	Elias γ	14,358,719	5.09	12,882,985	4.25	19,814,961	5.66
not A	binary(3)	10,649,437	2.12	9,472,568	1.88	11,341,438	2.25
not C	binary(3)	11,813,272	2.35	8,686,985	1.73	12,796,833	2.54
not G	binary(3)	11,266,210	2.24	8,697,083	1.73	12,407,771	2.47
not T	binary(3)	10,850,150	2.16	9,432,063	1.87	11,332,794	2.25
is N	binary(2^{30})	6,446	0.01	226,187	0.26	98,348	0.31
Reads without an RNAME *field supplied*							
Bases	binary(4)	0	0.00	0	0.00	144,363,100	34.42
is N	binary(2^{30})	0	0.00	0	0.00	788,351	0.93
Total			40.06		34.32		96.22

Table 6: Costs for SEQ components when stored as shown in Figure 5. Direct application of gzip to the original SEQ and POS fields in a striped approach results in corresponding costs of 46.43 MB, 39.91 MB, and 232.54 MB respectively.

sion. But some components of the SAM-format data have elements of sampling associated with them, most notably the QUAL field (described in Section 3). Moreover, as is illustrated in Table 4, the SEQ components and the dominant space requirement when SAM data is compressed. As a tangible reminder of this, combining the data presented in Tables 3 and 6 implies that, summed over all of the SEQ values in HG00113, each base can be stored in an average of 0.58 bits. But the corresponding gzip'ed representation of QUAL elements requires an average of 3.97 bits each. This imbalance makes the QUAL fields costly indeed to store.

There are two key characteristics that contribute to QUAL sequences being harder to compress than SEQ components. First, they are over a larger alphabet. The ASCII-33 mapping that is used to convert probabilities into letters typically spans between ten and twenty values in a typical SAM file. And second, it is not possible to exploit the RNAME-based overlaps when compressing QUAL fields in the way that was possible with the SEQ fields, because different reads that cover the same base position are uncorrelated – the QUAL value is influenced by a wide range of factors other than the actual offset at which it occurs.

However, the QUAL values represent quantized values over a numeric domain, and so in some situations it may be appropriate to quantize them more coarsely than via the ASCII-33 representation described in Table 1. If a tolerance p is stipulated, and limited flexibility of values introduced, with the proviso that no QUAL score may be varied by more than p units form its original quantized value, then a spectrum of lossy representations can be introduced. When $p = 0$, the representation is lossless.

To exploit this possibility, we represent the QUAL sequence as a list of tuples, consisting of a value followed by a repeat count. This run-length encoded approach will then naturally exploit repeated values, if they can be created via the flexibility introduced by the lossy representation. To encode a QUAL sequence for a given parameter p, consecutive values from the QUAL are added to a growing run while the difference between the maximum and

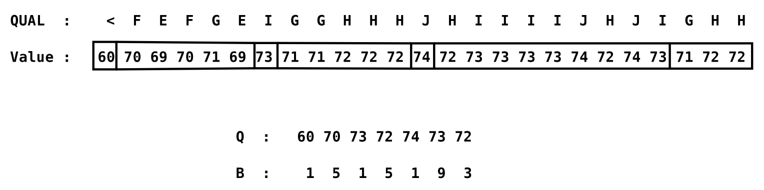

QUAL : < F E F G E I G G H H H J H I I I I J H J I G H H
Value : 60 | 70 69 70 71 69 | 73 | 71 71 72 72 72 | 74 | 72 73 73 73 73 74 72 74 73 | 71 72 72
Q : 60 70 73 72 74 73 72
B : 1 5 1 5 1 9 3

Figure 7: Lossy representation of QUAL fields. In this example $p = 1$, and each original value is represented by a mapped value that differs by at most one from the original.

Component	Code	HG00113 Size (MB)	HG00559 Size (MB)	Local Size (MB)
Fidelity parameter $p = 0$ (lossless)				
Run-length sequence	Elias γ	61.48	32.83	92.16
Byte sequence	ASCII	414.88	204.98	506.48
Byte sequence	binary (global)	311.16	153.74	379.86
Byte sequence	binary (local)	263.83	143.61	347.26
Fidelity parameter $p = 1$				
Run-lengths	Elias γ	66.89	32.98	80.60
Byte sequence	ASCII	300.98	131.70	311.51
Byte sequence	binary (global)	225.73	98.77	233.63
Byte sequence	binary (local)	195.10	94.86	214.30
Fidelity parameter $p = 2$				
Run-lengths	Elias γ	65.01	29.11	71.99
Byte sequence	ASCII	230.62	90.09	229.33
Byte sequence	binary (global)	172.96	67.57	172.00
Byte sequence	binary (local)	151.71	66.74	160.43
Fidelity parameter $p = 3$				
Run-lengths	Elias γ	60.34	24.88	64.28
Byte sequence	ASCII	177.62	64.53	176.24
Byte sequence	binary (global)	133.21	48.40	132.18
Byte sequence	binary (local)	118.93	49.23	126.26

Table 7: Lossy compression of QUAL fields. Each QUAL value is replaced by one that is at most p different from its true value. Direct application of `gzip` to the same SEQ data results in corresponding files of size 239.60 MB, 141.68 MB, and 349.83 MB respectively.

minimum of the values in the run is less than $2p$. Once a trigger item is encountered that would cause the difference to be greater than $2p$, a tuple is emitted comprising the current run length, and the mid-value that represents it. The trigger item is then the first value in the next run. Figure 7 gives and example of this process, with $p = 1$. In this example the 25 QUAL values are reduced to a total of 7 tuples, including one that includes 9 QUAL values in the range 72 to 74, all represented by the mid-value 73. With $p = 2$, the same sequence would be further reduced to just three runs, with mid-values of 60, 71, and 72 respectively.

To store the runs, we again seek to make use of static codes. Table 7 shows how this might be done, for a lossless representation with $p = 0$, and for three different lossy options with $p > 0$. In this set of measurements, the length of each run is assumed to be stored using the Elias γ code (see Moffat and Turpin (2002)); and three different approaches to representing each of the corresponding QUAL values are examined:

- as a plain ASCII bytes, as is used in the uncompressed SAM file;
- as a binary value using whole-of-file global parameters, using the number of bits indicated by the range of QUAL values stored in the SAM file; and
- as a binary value using per-alignment read parameters, with two additional bytes stored per alignment to indicate the upper and lower bounds of the local binary code.

	HG00113	HG00559	Local
$p = 0$	1.16	1.26	1.52
$p = 1$	1.61	1.97	2.48
$p = 2$	2.10	2.88	3.37
$p = 3$	2.72	4.01	4.38

Table 8: Average number of bases per run of QUAL values for three files and four different values of the fidelity parameter p.

The latter is superior in all cases, even allowing for the overhead caused by the two extra bytes. When $p = 0$ the combined cost of the runlengths and QUAL values is (unsurprisingly) greater than the cost of applying `gzip` to the same data. But as p is increased, and lossy compression is introduced, the cost decreases.

Table 8 lists the average length of the runs that are formed. As anticipated, increasing p results in increased run lengths, and hence better compression. On the other hand, lossy representations are always a risk, since future uses of data might require a fidelity of representation that seems unnecessary now. One option would thus be to store the QUAL values in lossy form using a relatively large value of p, plus store a difference list (also compressed) as a separate resource that would then allow exact reproduction of the original QUAL sequence, should it be required.

5 Querying SAM files

The key data extraction operation applied to SAM files is to isolate and present a window of the reads that it contains, identified by an RNAME and a set of offset positions within it. For example, if a particular trait is known to be encoded in some section of the chromosome, a researcher may interrogate the SAM-format data that has been generated for an individual, to see where and by how much that individual differs from the reference in regard to the identified range of bases. Because of the small but persistent possibility of error, all of the read alignments associated with that window of bases are extracted from the SAM file and displayed to the researcher.

Supporting localized extraction options via complete decoding and linear scan is expensive, even for uncompressed data. Tools for working with BAM format data are similarly costly, and involved decompression of non-trivial blocks of data, followed by sequential scanning over the read alignments, looking for overlaps.

The storage structure described in Section 4 emphasized simple static codes for two purposes:

- first and foremost, to avoid all obstacles to random-access decoding, so that given a set of pointers into the various streams of data, decoding can be commenced immediately from that index location; and
- second, to allow for faster decoding than is typically possible if adaptive models, or inverse Burrows-Wheeler transformations, or similar, need to be consulted for each character generated.

The information preserving reordering of the SAM file lines assists with these goals, grouping them first by RNAME, and then by offset relative to the start of it. To extract any/all reads that relate to a specified set of bases, the set of reads that is required is identified by seeking within the compressed representation, looking for the first read alignment whose last base overlaps with the search interval, and for the first read alignment thereafter whose first base is to the right of the search interval.

To allow such seek operations to take place, the set of read alignments will be sampled at regular intervals, and an index built that maps offset values into bit pointers into the compressed data stream (and into each of the distinct streams of bits that must be combined in order to decode). The sampling interval will control the tradeoff between speed of access and space required, with frequent samples allowing fast access, but requiring increased space.

6 Related Work

There has been a range of previous work that examines the problem of efficiently representing DNA data (the SEQ string that is a component of SAM-format files), including early discussions such as that provided by Grumbach and Tahi (1993), who identify the need to locate exact matches, palindromic matches, and complement matches at separations much greater than is the usual case in typical text compression applications. Ten years later, Manzini and Rastero (2004) describe an enhanced scheme that uses a finger-printing techniques to identify three kinds of long repetitions (exact, reverse-complement, and approximate, and possibly far apart) in DNA sequences, and uses a range of methods to code descriptions of the repetitions so identified. Their method is both fast and effective compared to other SEQ-specific approaches, but relies on an adaptive model (namely, the part of the sequence already encoded, which is used as a dictionary of long phrases), and hence is not suited to random-access decoding.

Cao et al. (2007) describe a compression approach based on multiple "experts", each of which forms a probability estimation for each symbol in the genome. The opinions of the experts are then weighted and combined, and an arithmetic coder used to convert the final overall probability distribution into an output bitstream. While this type of approach is interesting from an "exactly how much compression can be attained" point of view, it is at odds with our intention to make use of simple state-less codes that allow indexed random-access decompression.

Kuruppu et al. (2012) describe a DNA compression regime they call COMRAD, which builds an explicit dictionary of 16-base sequences, and then uses it iteratively to form longer recurring phrases, assigning a new identifier to each such extended phrase. It is an example of DNA-tailored grammar-based compression; and when applied to large sets of related genomes, is able to infer and exploit very long cross-genome repetitions. That is, the more closely related the set of sequences that is being processed, the better the more effective the compression. Kuruppu et al. also explored the scalability of their approach, by simulating the generation of extended sets of related genomes, and testing the performance of COMRAD against them.

Deorowicz and Grabowski (2011) consider genomic data stored in FASTQ-format, which, like SAM-format, maintains a QUAL string for each SEQ string, and creates files that can contain millions of short read alignments. They use an adaptive dictionary-based approach, and consider repetitions of 36 bases or more; one of the determining factors as to whether any given phrase is retained in the dictionary is the associated quality scores, working on the principle that low-quality phrases are less likely to recur than high-quality ones. As is proposed in Section 4, Deorowicz and Grabowski also make use of runlength information when storing the QUAL fields. They give results that show that their system DSRC achieves excellent compression with typical FASTQ files in the GB range being reduced to 20% or less of their original size.

Matos et al. (2012) also consider the question of multi-sequence alignment compression. They describe an approach similar to that summarized in Section 4, and derive a sequence that they call the "estimated ancestor". At the core of their mechanism is a two-dimensional context predictor (similar to the type of predictor used for bi-level image compression) that when coupled with blended probability estimates and an arithmetic coder is able to represent a set of related SEQ components in under one bit per base.

Yanovsky (2011) presents a compression implementation for multi-alignment SEQ values called ReCoil, which is designed to handle large files of genomic data stored on disk (rather than in main memory) and where repetitions might be widely separated. In this approach, reads that share common subsequences of 15 or more bases are identified, and a common substitution made at all locations, thereby saving space. The main contribution of the paper is showing how the required steps can be mapped onto sequential scanning and sorting processes that are efficient when the data is held on secondary storage.

Cox et al. (2012) apply the well-known Burrows-Wheeler transform to multi-alignment short read genomic data. But unlike the general-purpose BWT-based compression program bzip2, which uses blocks of just 900 kB, here very large numbers of reads can be accommodated through the suffix-sorting process that generates the BWT. The transformed string is then coded using a context-based estimator, and arithmetic coding. Excellent compression outcomes are achieved, because all of the like subsequences are brought together by the large-scale BWT process, and hence the probability estimates that are generated are relatively highly skewed, and the emitted arithmetic tend to be very short. It is not clear whether the same techniques can be applied to the QUAL fields that dominate SAM-format files.

One potential problem with multi-alignment compression is the need for the RNAME and POS fields to be supplied. While the methods presented in Section 4 include a

PRS in the compressed package, rather than simply referring to an external reference sequence, they nevertheless require the reads to have been aligned at the time they were generated. When read alignments have not been provided with RNAME and POS fields, we have coded them as unreferenced components, and used less effective techniques, as shown in the bottom rows of Table 6. To address this problem, Jones et al. (2012) include a *de novo assembly* component in their `Quip` software, that seeks out possible overlaps of reads seeded using overlapping 12-grams, and uses a probabilistic Bloom filter to reduce the amount of memory space required while this is taking place. The rest of `Quip` makes use of an order-12 (on bases) context-based predictor, and arithmetic coding to convert those predictions into a bitstream.

In work that is closely related to the proposal presented here, Daily et al. (2010) focus on simple static codes such as Golomb codes, Rice codes, and Elias codes, and have created a tool called `GenCompress` that handles multi-alignment files with reference to an externally-stored RNAME sequence. While we do not wish to make use of explicit external reference sequences, there are techniques in their work that may also be applicable when we construct our own implementation.

In a similar vein, Wan et al. (2012) extend generic SEQ compression to consider how best to handle collections of related reads. They carry out a detailed study of the QUAL field that is part of SAM- and FASTQ-format files, and consider mapping transformations – including lossy ones – that improve the compressibility of this data. A range of codes are considered for representing the mapped values, including binary and other static representations. Wan et al. conclude that general purpose compressors such as `gzip` and `bzip2` are less effective for QUAL values than are simple codes, and that compression effectiveness can be traded off against representational fidelity; in this regard, the preliminary results presented above can be regarded as a partial verification of their observations.

A wide range of other techniques have been proposed: Tembe et al. (2010) represent all possible distinct pairs of base and quality value using Huffman codes; Christley et al. (2009) store only the variations between sequences, coding relative to a reference sequence; and Kozanitis et al. (2010) divide reads into fragments of a chosen size, and note that neighboring quality values are correlated and can be handled using a Markov model.

7 Summary

We have described structures and techniques suitable for representing SAM-format files containing genomic data. The next significant step in this project is to implement the proposed combination of mechanism as an integrated compression tool, and verify that it is as effective as is indicated by the results obtained during this feasibility study. We will also implement the required random access operations, and measure their efficiency; beyond that we plan to seek ways of supporting that capability using modern succinct data structures so that the cost of the additional index information is minimized (or indeed, free). Our overall objective is to provide fast random interval-based access and compact storage requirements in a single package. The work presented here lays the foundations for such a development, and gives clear guidance as to the future path of this project.

References

Ansorge, W. (2009), 'Next-generation DNA sequencing techniques', *New Biotechnology* **25**(4), 195–203.

Bell, T. C., Cleary, J. G. and Witten, I. H. (1990), *Text Compression*, Prentice Hall, Englewood Cliffs, NJ.

Cao, M. D., Dix, T. I., Allison, L. and Mears, C. (2007), A simple statistical algorithm for biological sequence compression, *in* 'Proc. IEEE Data Compression Conference', pp. 43–52.

Christley, S., Lu, Y., Li, C. and Xie, X. (2009), 'Human genomes as email attachments', *Bioinformatics* **25**(2), 274–275.

Cox, A. J., Bauer, M. J., Jakobi, T. and Rosone, G. (2012), 'Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform', *Bioinformatics* **28**(11), 1415–1419.

Daily, K., Rigor, P., Christley, S., Xie, X. and Baldi, P. (2010), 'Data structures and compression algorithms for high-throughput sequencing technologies', *BMC Bioinformatics* **11**, 514.

Deorowicz, S. and Grabowski, S. (2011), 'Compression of DNA sequence reads in FASTQ format', *Bioinformatics* **27**(6), 860–862.

Grumbach, S. and Tahi, F. (1993), Compression of DNA sequences, *in* 'Proc. IEEE Data Compression Conference', pp. 340–350.

Jones, D. C., Ruzzo, W. L., Peng, X. and Katze, M. G. (2012), 'Compression of next-generation sequencing reads aided by highly efficient de novo assembly', *Nucleic Acid Research* pp. 1–9.

Kozanitis, C., Saunders, C., Kruglyak, S., Bafna, V. and Varghese, G. (2010), Compressing genomic sequence fragments using SLIMGENE, *in* 'Proc. 14th Ann. Int. Conf. Research in Computational Molecular Biology', RECOMB'10, pp. 310–324.

Kuruppu, S., Beresford-Smith, B., Conway, T. C. and Zobel, J. (2012), 'Iterative dictionary construction for compression of large DNA data sets', *IEEE/ACM Trans. Comput. Biology Bioinform.* **9**(1), 137–149.

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G. and Durbin, R. (2009), 'The sequence alignment/map format and SAMtools', *Bioinformatics* **25**(16), 2078–9.

Manzini, G. and Rastero, M. (2004), 'A simple and fast DNA compressor', *Softw., Pract. Exper.* **34**(14), 1397–1411.

Matos, L., Pratas, D. and Pinho, A. (2012), Compression of whole genome alignments using a mixture of finite-context models, *in* 'Image Analysis and Recognition - 9th International Conference (ICIAR)', pp. 359–366.

Moffat, A. and Turpin, A. (2002), *Compression and Coding Algorithms*, Kluwer Academic, Boston, MA.

Navarro, G. and Mäkinen, V. (2007), 'Compressed full-text indexes', *ACM Comput. Surv.* **39**(1).

Tembe, W., Lowey, J. and Suh, E. (2010), 'G-SQZ: compact encoding of genomic sequence and quality data', *Bioinformatics* **26**(17), 2192–2194.

Wan, R., Anh, V. N. and Asai, K. (2012), 'Transformations for the compression of FASTQ quality scores of next-generation sequencing data', *Bioinformatics* **28**(5), 628–635.

Yanovsky, V. (2011), 'ReCoil: An algorithm for compression of extremely large datasets of DNA data', *Algorithms for Molecular Biology* **6**(23).

Parallel Execution of Prioritized Test Cases for Regression Testing of Web Applications

Deepak Garg **Amitava Datta**

School of Computer Science and Software Engineering
The University of Western Australia,
Perth, Western Australia 6009,

email: `deepak@csse.uwa.edu.au`

email: `amitava.datta@uwa.edu.au`

Abstract

We present a new approach for automatically prioritizing and distributing test cases on multiple machines. Our approach is based on a functional dependency graph (FDG) of a web application. We partition the test suite into test sets according to the functionalities and associate the test sets with each module of the FDG. The high priority modules and their associated test sets are then distributed evenly among the available machines. Moreover, we further prioritize the test cases by using inter-procedural control flow graphs within individual functional modules. Our suggested approach reduces the test suite execution time and helps in detecting the faults early in a regression testing cycle. We demonstrate the effectiveness of our technique through an experimental study of a web application and measuring the performance of our technique by using the well known APFD metric.

Keywords: Regression testing, Test case prioritization, Web applications, Parallel execution

1 Introduction

Web applications typically use a complex and multi-tiered, heterogeneous architecture including web servers, application servers, database servers and client interpreters. Web applications undergo maintenance at a faster rate than any other software system (Elbaum et al. 2003), as new functionalities are introduced depending on user requirements. The main aim of testing of a web application is to detect faults in the required services and functionalities and fix these faults, to enable it to function according to specifications (Lucca & Fasolino 2006). A test suite for testing a web application consists of many test cases. The serial execution of a test suite on a single machine might take many hours (Lastovetsky 2005) depending on the size of an application, the machine where the test suite is run and its workload.

For every new version of a web application released, regression testing is required to test the compatibility of the new features with the previously tested functionalities. New test cases are generated to perform regression testing. Since web applications typically have a rapid turn around time, it becomes very difficult to execute all the test cases within a specified amount of time. The cost of re-running all test cases may be expensive and not always useful as sometimes only selected functionalities need to be tested. Hence, test cases are usually prioritized during testing in order to discover the likely vulnerable parts of the code early so that developers have more time to identify and debug the faults. Many strategies have been proposed for prioritizing C (Elbaum et al. 2000) (Elbaum et al. 2002) and Java programs (Harrold et al. 2001) (Do et al. 2006) (Jeffrey & Gupta 2006). For web applications, Sampath et al. (Sampath et al. 2008) suggested prioritization techniques using user-session based test cases.

Even though prioritization mitigates some of the drawbacks of executing a complete set of test cases, execution of test cases on a single machine may not achieve the rapid testing criterion of large web applications. Also, most organizations can afford to deploy multiple machines for testing. Hence, a possible way of expediting the speed of regression testing is to run disjoint parts of the same test suite on multiple machines. In this paper, we investigate the parallel execution of prioritized test cases. We first construct a *functionality dependency graph* (FDG) of the entire web application from its UML specification. A node in the FDG is a unique functionality and a directed edge from node m to node n indicates the functional dependency of node n on node m. We partition the entire test suite into test sets such that each test set is associated with a unique functional module in the FDG.

Next, we identify a subgraph S of the FDG for prioritization. This subgraph consists of the nodes that have been modified after the previous regression testing cycle and nodes that are dependent on these modified nodes. We assign priorities to the nodes of S. The test sets are executed according to these priorities when a single machine is used for testing. For multiple machines, we sort the nodes of S in priority order and allocate nodes from different priority groups approximately evenly among the available machines. We further prioritize the execution of test sets in each machine by using code level *control flow graphs* (CFG). We execute the test sets allocated to each machine simultaneously in parallel. Finally, we collect the test results in a single machine. In this paper, we suggest our approach using web applications but our approach is quite general and can be used for any kind of software application if we can construct a functionality dependency graph for a software system.

Our main contributions in this paper are: **1.** Detection of modified functional modules in web application. **2.** Prioritization of the test cases using FDG and CFG. **3.** Partitioning of the prioritized test suite into different test sets for parallel execution on different machines. The rest of the paper is organized as follows. Section II presents the background study related to prioritization techniques and parallel execution of test cases. Section III presents the work related to generation of functional test cases from UML Activity diagrams and the generation of the functionality dependency graph. Section IV describes our approach that includes partitioning of test suite into test sets, prioritization framework, collecting test sets into groups for distributing to individual machines, parallel prioritization and parallel execution of test cases. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes black box tests at the system level when information between requirements, test case, and test failures is maintained by the software development team.

Elbaum et al. (Elbaum et al. 2004) proposed a prioritization technique that improves fault detection rates. They applied different prioritization techniques to different programs and identified a technique that is cost effective for early detection of faults. Their method is based on total function coverage prioritization, which orders the test cases according to the number of functions they cover. If multiple test cases cover the same number of functions, then this technique orders them randomly. They found that the performance of this technique varied according to the program attributes, change attributes and test suite characteristics. Rothermel et al. (Rothermel et al. 2000) (Rothermel et al. 2001) suggested the prioritization of test cases when the software is written in a single language. Their technique constructs control flow graphs and uses those graphs to select the test cases related to the modified versions of the software.

Wong et al. (Wong et al. 1997) prioritized test cases using the criterion of extending cost per additional coverage. They proposed a technique that is a combination of minimization and prioritization to determine which regression tests should be re-run. They first find the minimal subset in terms of the number of test cases that preserves the same test coverage as the original test set. Then they sort the test cases in order of increasing cost per increasing coverage and then select the top n test cases for revalidation. The results suggest that this technique can provide software testers with cost-effective alternatives to help conduct quick regression testing under budget constraints and time pressure.

Sampath et al. (Sampath et al. 2008) suggested a test suite prioritization technique for web applications by test lengths, frequency of appearance of request sequences, systematic coverage of parameter values and their interactions. They considered frequency of user requests and interaction of parameter values in the requests. A test case that is designed according to user sessions is based on a series of HTTP requests consisting of base requests and name-value pairs that are used to access that application. A base request is a HTTP request to access both static and dynamic content in web pages. They use the entire test suite for execution but they ordered the test cases on the basis of user session requests to detect the faults early in test suite execution.

Existing prioritization techniques related to web applications prioritize the test cases when the test cases are a part of a single test suite as there is only one processing queue that selects the test cases to run. Qu et al. (Qu et al. 2008) describes the prioritization problem using parallel scenario. They presented their results for a standalone application that was installed on one machine. They tested the Microsoft PowerPoint application as their target application. Their approach was not designed to test the application according to its functional requirements. Hence their approach is not applicable for regression testing of complex web applications.

Haftmann et al. (Haftmann et al. 2005) suggested a technique for parallel execution of test cases but their technique involves the testing of only the databases. Chakraborty et al. (Chakraborty & Shah 2011) suggested an approach for parallel execution of test cases by collecting the data from manual test processes and they assumed that manual test execution time and automated test execution time are equal. In case of complex web applications, manual test execution time and automated test execution time may vary. It is not possible to test the functional requirements with the approach suggested by Chakraborty et al. as they do not consider the functional specifications while extracting the dependency graph.

None of these papers suggested an automated parallel prioritization approach to test web applications according to the functional specifications for regression testing of complex web applications. In this paper, we propose a new automated approach for execution of prioritized test suite by distributing it into several test sets. The execution of parallel prioritized test sets reduces the execution time and detects faults early. Our technique involves two major challenges: partitioning and ordering. In partitioning a test suite, we have to decide which test cases need to be executed on which machine and ordering is used to decide in which order the subsets of the test cases are executed in each machine.

3 Related Work

3.1 Generation of test cases

Törsel et al. and Tung et al. suggested automatic generation of test cases for web applications (Tung et al. 2010) (Torsel 2011). They described a fully automated approach to generate test cases for functional testing. We generate functional test cases from the functional specifications of web applications and convert them into a C# format readable by the test tool Selenium. After generating all the test cases, we partition them into *test sets* according to the FDG (defined below). The test cases in one particular test set belongs to a unique functionality and is associated with a node of the FDG.

3.2 Functionality and Functional Modules

Functionality refers to user actions such as keyboard and mouse events required to navigate through web applications (Sampath et al. 2007) (Di Lucca et al. 2003). UML sequence diagrams provide information about functionality and the interaction among different objects in a web application (Cartaxo et al. 2007). Functionality is defined in the specification documents and is provided by the user interface (Heinecke et al. 2010). A *functional module* is a collection of different functions (at the code level) to enable the functionality to behave according to the specifications.

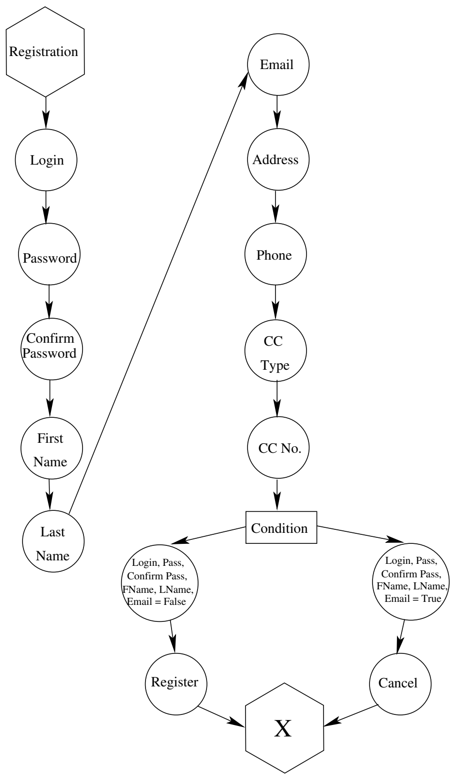


Figure 1: Control Flow Graph (CFG) for the *Registration* node in the FDG of the *Online Bookstore* application.

3.3 Functionality dependency graph

A *FDG* (Zimmermann & Nagappan 2007) (Samuel et al. 2005) is a directed graph that is used to describe the relationship between functionalities in web applications. For an FDG $G = \{V,E\}$, the node set V is the set of functionalities in a web application and E is the set of directed edges that represents the dependency relationship among the functionalities. A directed edge from $m \in V$ to $n \in V$ indicates the functional dependency of the module n on module m.

3.4 Control flow graph

We assume that each functionality is composed of a class or a combination of classes in the source code of a web application. We extracted CFG for every functional module in the FDG of the *Online Bookstore*[1] application using the algorithm by Rothermel et al. (Rothermel et al. 2000). Each module in a CFG is either a C# or an ASPX (Active Server Pages Extended) source code module. We show an example CFG in Fig. 1.

[1]available freely at www.gotocode.com

4 Our approach

We partition the complete test suite into test sets, each test set is associated with a unique functional module or node in the FDG. We prioritize the test cases within each test set using the CFG of the corresponding functional module

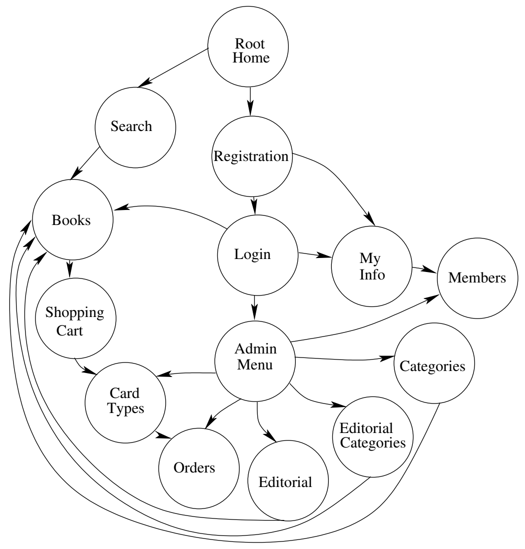


Figure 2: The Functionality Dependency Graph (FDG) for the *Online BookStore* application.

4.1 Partitioning of Test Suite

We extract the FDG of the *Online Bookstore* application from its functional specifications based on our previous work (Garg & Datta 2012). We captured various functional requirements using UML. We captured the interactions between the requirements. We manually extracted the FDG for the *Online Bookstore* application from the functional specifications as shown in Fig. 2 but depending upon the technologies, other methods of generating FDG could be applied. Fig. 2 shows the various functional modules for this web application.

We generate test cases for all these functional modules and initially randomly store them in a test suite. We partition the entire test suite into test sets, such that each test set is associated with a unique functional module. We automatically identify the test cases that are associated with a particular functional module by reading the test cases and matching the statements. If a test case contains the statements that are required to test some functionality F_i, we store that test case separately in a test set T_i that is associated with F_i. This computation is done in a single machine that we call as the *test server.*

4.2 Prioritization of test cases for each functional module

Rothermel et al. defined the problem of test suite prioritization in (Rothermel et al. 2001). Given T as a test suite, P is the set of all test suites that are the prioritized orderings of T obtained by permuting the tests of T and F is a function obtained from P to the reals, the problem is to find a permutation, $T' \in P$ such that $(\forall T'')(T'' \in P)[F(T') \geq F(T'')]$.

Given a functional module F_i (a node in the FDG), we prioritize the test cases in the corresponding test set T_i according to the CFG of F_i. The modules with the dotted circles in Fig. 3 refer to the modified source code modules of the *Online Bookstore* web application. Each code module in a CFG is associated with a test case or series of test cases. The modified code modules in CFG are executed in priority compared to the unmodified modules. Our prioritization strategy is as follows:

- The modified modules closer to the root or the class definition are given highest priority in execution of test cases in a test set.
- If two modified modules are at the same level in CFG, the modules having more dependent modules will be given priority in execution.
- We randomly order the test cases belonging to the remaining unmodified modules.

Note that, this prioritization is done for all the functional modules in the FDG after we generate the test cases in the test suite. As all the functional modules in the FDG may be tested in different regression test cycles, we need to prioritize the test cases for each module using the corresponding CFG. This computation is done in the test server.

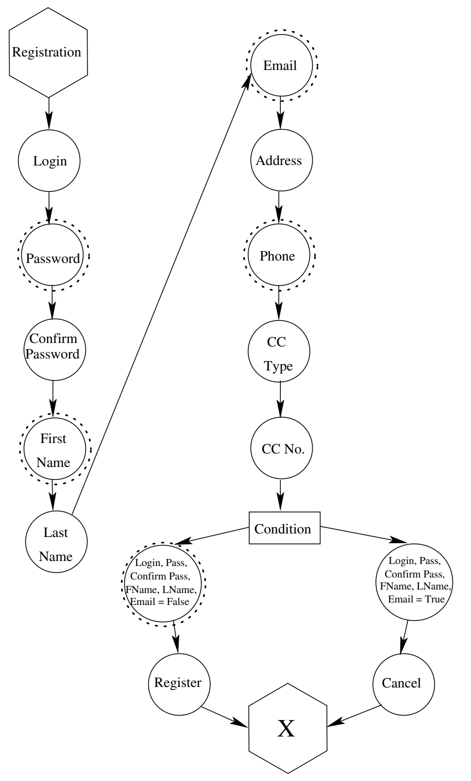


Figure 3: Control Flow Graph (CFG) with modified modules of the *Registration* functional module of the *Online Bookstore* application.

4.3 Extraction of the affected subgraph from the FDG

We recall that a directed edge from node m to node n in the FDG indicates that node n is dependent on node m. We also say node m *invokes* node n. We assign priorities to the nodes of the FDG in the following way:

1. A newly introduced node in the FDG is given the highest priority. If there are multiple newly introduced nodes, they are assigned the highest priorities in an arbitrary order.
2. The modified nodes in the FDG are given the next lower priorities.
3. The next lower priorities are assigned to the nodes that directly or indirectly invoke the modified or newly introduced nodes. A higher priority among these nodes is assigned to a node that is closer (in terms of path length) to a newly introduced or modified node. In case of multiple paths, the node that is following the shortest path to a newly introduced or modified node will be given priority.
4. All other nodes (except the nodes that are invoked by the modified or newly introduced nodes) are assigned the next lower priority in an arbitrary order.
5. The nodes that are invoked either directly or indirectly by the modified or newly introduced nodes are assigned the least priorities. These nodes have been tested in the previous regression test cycles and they are unchanged. Hence we assume that they need not be tested in the current test cycle. Hence these nodes are called *unaffected nodes.* All other nodes are called *affected* nodes.

We use the following observation for extracting the affected subgraph of the FDG.

Lemma 1 *The affected nodes in the FDG form a connected subgraph of the FDG.*

proof 1 *Our prioritization scheme ensures that every directed path from the root to a leaf of the FDG has all the affected nodes as a connected sub-path. We prove this by contradiction. Consider three consecutive nodes m, n and p on a root to leaf directed path such that m and p are affected but n is not affected. Such a situation will make the affected subgraph disconnected. However, p is affected and n invokes p and hence, our prioritization scheme ensures that n is also affected, a contradiction.*

We extract the *affected* subgraph S of the FDG by performing a depth-first search. The depth-first search picks up the subgraph containing only the affected nodes, as the search backtracks whenever it encounters an unaffected node. The search returns with the affected subgraph due to Lemma 1. The nodes in S are distributed to the available machines in our parallel prioritization scheme. This is discussed below. This computation is done in the test server.

4.4 Allocating functional modules to machines

The test server allocates the functional modules to the machines participating in the parallel test execution. It is easy to distribute the test sets for the functional modules to different machines if the number of functional modules F is less than or equal to the number of available machines M. Each machine can be allocated the test set of one functional module. However, realistically complex web applications consist of a large number of functional modules and

F is usually much greater than M. Hence, we need to allocate multiple functional modules and their associated test sets to each available machine. We construct subsets of functional modules in such a way that each machine is allocated approximately an equal number of functional modules as well as the priorities of the different functional modules in each subset are also approximately equal.

We construct the subsets in the following way. Each node in the selected subgraph S of the DFG has a priority associated with it. We sort the nodes according to these priorities and store in an array A. From the discussion in Section 4.3, the nodes can have four different priorities $p_i, 1 \leq i \leq 4$. We denote the number of nodes with priority p_i as $|p_i|$. We allocate $\lceil \frac{|p_i|}{M} \rceil$ $(1 \leq i \leq 4)$ nodes from the priority class p_i to each of the M machines. Though it is possible that the last machine is allocated less than $\frac{|p_i|}{M}$ nodes, nodes from each priority class are approximately evenly distributed among the M participating machines.

4.5 Parallel execution strategy

The functional modules and their associate test sets are allocated to the participating machines by the test server according to the strategy discussed in Section 4.4.

Fig. 4 shows the setup of the entire test process. The main machine acts as a hub and stores all the test case execution data and behaves like a test server. The other computing nodes execute the allocated test sets in parallel. Each machine stores the test execution results and these results are sent to the test server for constructing the combined test report.

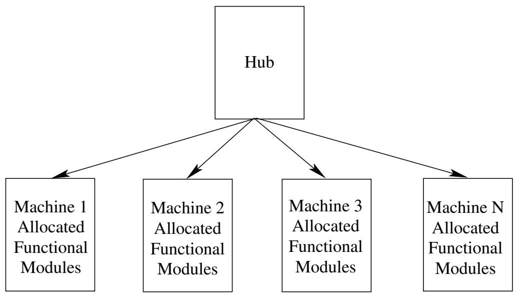

Hub
Machine 1 Allocated Functional Modules
Machine 2 Allocated Functional Modules
Machine 3 Allocated Functional Modules
Machine N Allocated Functional Modules

Figure 4: Parallel execution of test sets

5 Experimental evaluation

In this section, we discuss our experimental set up using our suggested approach.

To perform the experimental evaluation, we extended some of the functionalities of this original application to make it more complex. This application is an online shopping portal for buying books. This application uses ASPX for its frontend and MySQL for its backend connectivity. The application allows the users to search for books by different keywords, add to the shopping cart and proceed to orders.

We randomly seeded 20 faults in various modified functionalities of the *Online Bookstore*. The faults were assumed to have similar cost levels. Three different kinds of faults (Guo & Sampath 2008) were seeded in the application: Logical Faults, Form Faults and Appearance Faults. A logical fault in the program code relates to business logic and control flow e.g., if the user inputs the same string for the *password* and *confirm password* fields, still the application displays the error message *Password and Confirm Password fields don't match.* Form faults in the program code modifies and displays name-value pairs in forms. Appearance faults controls the way in which a web page is displayed. We seeded faults in the various modified functionalities like *Registration*, *Members*, *MyInfo*, *Login* and *Books* and assumed that the faults behave like real faults.

We used C# to implement our proposed approach. We generated 130 test cases from the UML Activity diagrams for the *Online Bookstore* and converted them to C# test scripts readable by the Selenium test tool for automatic test suite execution (Torsel 2011). The generated test cases were assumed to be non-redundant and were generated according to the functional specifications.

We partitioned the test suite into different test sets and associated these test sets with the different functional modules. Each test set associated with a functional module is composed of test cases related to that specific functionality. The *Online Bookstore* application has 14 functional modules (Fig. 2) and hence the test suite was divided into 14 test sets. Though our framework is general and can be applied when only some of the functional modules are modified, we modified all the 14 functional modules to evaluate the worst-case performance of our parallel prioritization scheme. Hence according to the discussion in Section 4.3, all the nodes had the highest priority. We used three computers for running the tests. The first two computers were allocated five functional modules each and the last was allocated the remaining four modules. The test cases were executed in parallel on their respective machines. We performed the experiments using Selenium Grid (Bruns et al. 2009). For comparing our results with a random distribution of test cases, we also randomly distributed the test cases among the three machines and executed them in parallel.

Rothermel et al. (Rothermel et al. 2001) presented the APFD (Average Percentage of Faults Detected) metric for measuring fault detection rates of test suites in a given order. APFD values range from 0 to 1; higher numbers imply faster (better) fault detection rates (Elbaum et al. 2001).

APFD can be calculated using the following formula:

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + ... + TF_{\mathrm{n}}}{mn} + \frac{1}{2n}$$

TF_i is the position of first test in T that exposes fault i
n= no. of faults
m= no. of test cases

Informally, APFD measures the area under the curve that is plotted by the percentage of faults detected by prioritized test case order and the test suite fraction.

We collected the test execution results and used the APFD metric to determine whether our approach detects faults earlier and faster compared to the random ordering of the test cases. We applied the APFD metric separately for the test cases run on each of the three participating machines.

Threats to Validity: We used three different machines with different hardware configurations to execute the test cases in parallel. All these machines had different workload conditions when we were running our experiments by publishing the web application on the virtual web server of the school. We noticed that the test execution times differed when we repeated our experiments. We manually seeded the faults in the web application. The faults may not be evenly distributed among the functionalities. Although they are considered to be faults of equal severity, faults with different severity levels may vary the results. The functional test case execution time may differ due to the varying lengths of test cases.

6 Results and Analysis

Table 1: Results for Random Ordering

%age of test suite run	Machine 1	Machine 2	Machine 3
10%	0	0	21.68
20%	0	0	61.28
30%	0	0	66.85
40%	0	0	66.85
50%	0	32.44	66.85
60%	19.80	47.06	66.85
70%	46.90	47.06	66.85
80%	46.90	47.06	66.85
90%	46.90	47.06	66.85
100%	50.61	47.06	67.18

We compared the test results for the random approach with our suggested approach using the APFD metric. We recall that we distributed the test cases randomly among the three participating machines in the random approach. The first two machines were allocated five modules each and the last machine was allotted the remaining four modules. Table I shows the results for the random approach.

We have shown the results in 10% increments. We use the APFD metric to explain our results. We note that the random ordering of the test set in the first machine has not detected any faults for the first 50% of the test set execution. The APFD results for 100% test set execution is 50.61. Similarly, the random ordering of the test set in the second machine has not detected any fault for the first 40% of the test set execution. The APFD result for 100% test set execution is 47.06.

Table 2: Results from our approach

%age of test suite run	Machine 1	Machine 2	Machine 3
10%	77.64	74.95	97.17
20%	77.64	74.95	97.17
30%	92.35	91.15	97.17
40%	92.35	91.55	97.17
50%	92.35	91.55	97.17
60%	92.35	91.55	97.17
70%	92.35	91.55	97.17
80%	92.35	91.55	97.17
90%	92.35	91.55	97.17
100%	92.35	91.55	97.17

Table II shows the APFD results using our prioritization approach. The test sets allocated to the first machine are able to detect all the faults in the first 30% of the test set execution. The APFD result for 100% test set execution using our prioritization approach is 92.35. The test sets allocated to the second machine are able to detect all the faults in the first 30% of the test set execution. The APFD result for 100% test set execution in using our prioritization approach is 91.55. Similarly, the test sets allocated to the third machine are able to detect all the faults in the first 10% of the test set execution. The APFD result for 100% test set execution for this test set is 97.17.

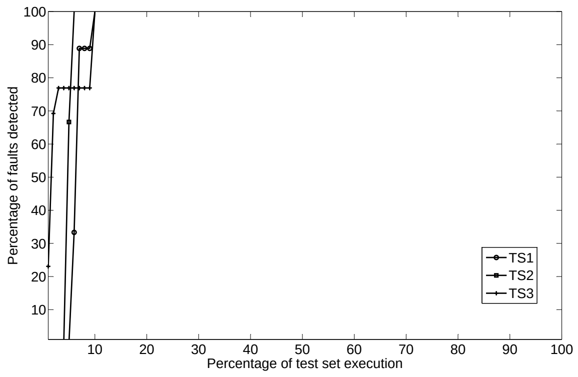


Figure 5: Faults detected (Random ordering of test sets)

Fig. 5 shows the execution results of random ordering of all three test sets. Fig. 5 shows that many faults were detected after executing close to 100% of the test cases. Fig. 6 shows the execution results of the test sets that are prioritized using our approach. Fig. 6 shows that many of the faults were detected in the first 30% of the test set execution.

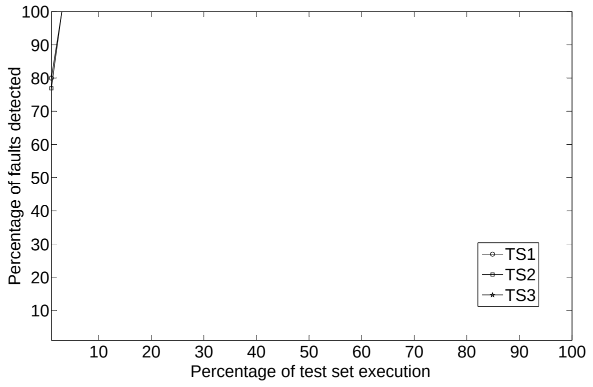


Figure 6: Faults detected (Prioritization Approach)

The execution for the entire test suite on a single machine took approximately 9 hours. After implementing our suggested approach, we were able to execute all the test cases in less than 3 hours and could detect all the faults in the first hour.

7 Conclusions and Future Work

We have proposed a novel parallel prioritization approach for regression testing of complex web applications in this paper. We prioritize the test case executions at two levels, by choosing and prioritizing the functional modules from the functional dependency graph and then ordering the test cases within each test set by using the control flow graphs at the code level. We then distributed the functional modules and their associated test sets among different machines. We measured the performance of our approach using the APFD metric.

We validated the results using various different test combinations and found that our approach is able to detect the faults early and within a small amount of time. In the future, we will validate our results on several other web applications. We will consider real faults with different cost levels. As we have shown, our first 30% execution of test sets detects most of the faults. In the future, we will suggest a technique that will select the test cases related only to the modified functionalities in web applications and execute only those test cases that will provide maximum fault detection. This may help to reduce the total test execution time even more, as we need to execute only a subset of test cases.

References

Bruns, A., Kornstadt, A. & Wichmann, D. (2009), 'Web application tests with selenium', *Software, IEEE* **26**(5), 88 –91.

Cartaxo, E., Neto, F. & Machado, P. (2007), Test case generation by means of uml sequence diagrams and labeled transition systems, *in* 'Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, ISIC', pp. 1292 –1297.

Chakraborty, S. & Shah, V. (2011), Towards an approach and framework for test-execution plan derivation, *in* 'Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)', pp. 488 –491.

Di Lucca, G., Fasolino, A., Tramontana, P. & De Carlini, U. (2003), Abstracting business level uml diagrams from web applications, *in* 'Proceedings of the Fifth IEEE International Workshop on Web Site Evolution', pp. 12 – 19.

Do, H., Rothermel, G. & Kinneer, A. (2006), Prioritizing junit test cases: An empirical assessment and cost-benefits analysis, *in* 'Empirical Softw. Engg.', Vol. 11, Kluwer Academic Publishers, Hingham, MA, USA, pp. 33–70.

Elbaum, S., Karre, S. & Rothermel, G. (2003), Improving web application testing with user session data, *in* 'Proceedings of the 25th International Conference on Software Engineering', pp. 49 – 59.

Elbaum, S., Malishevsky, A. G. & Rothermel, G. (2000), Prioritizing test cases for regression testing, *in* 'Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis', ISSTA '00, ACM, New York, NY, USA, pp. 102–112.

Elbaum, S., Malishevsky, A. & Rothermel, G. (2001), Incorporating varying test costs and fault severities into test case prioritization, *in* 'Proceedings of the 23rd International Conference on Software Engineering', ICSE '01, IEEE Computer Society, Washington, DC, USA, pp. 329–338.

Elbaum, S., Malishevsky, A. & Rothermel, G. (2002), Test case prioritization: a family of empirical studies, *in* 'IEEE Transactions on Software Engineering', Vol. 28, IEEE Press, Piscataway, NJ, USA, pp. 159 –182.

Elbaum, S., Rothermel, G., Kanduri, S. & Malishevsky, A. G. (2004), Selecting a cost-effective test case prioritization technique, *in* 'Software Quality Control', Vol. 12, Kluwer Academic Publishers, Hingham, MA, USA, pp. 185–210.

Garg, D. & Datta, A. (2012), Test case prioritization due to database changes in web applications, *in* 'Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)', pp. 726 –730.

Guo, Y. & Sampath, S. (2008), Web application fault classification - an exploratory study, *in* 'Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement', ESEM '08, ACM, New York, NY, USA, pp. 303–305.

Haftmann, F., Kossmann, D. & Lo, E. (2005), Parallel execution of test runs for database application systems, *in* 'Proceedings of the 31st international conference on Very large data bases', VLDB '05, VLDB Endowment, pp. 589–600.

Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A. & Gujarathi, A. (2001), 'Regression test selection for Java software', *SIGPLAN Not.* **36**, 312–326.

Heinecke, A., Bruckmann, T., Griebe, T. & Gruhn, V. (2010), Generating test plans for acceptance tests from uml activity diagrams, *in* 'Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)', pp. 57 –66.

Jeffrey, D. & Gupta, N. (2006), Test case prioritization using relevant slices, *in* 'Proceedings of the 30th Annual International Computer Software and Applications Conference, COMPSAC '06', Vol. 1, pp. 411 –420.

Lastovetsky, A. (2005), 'Parallel testing of distributed software', *Inf. Softw. Technol.* **47**, 657–662.

Lucca, G. A. D. & Fasolino, A. R. (2006), Testing web-based applications: The state of the art and future trends, *in* 'Quality Assurance and Testing of Web-Based Applications', Vol. 48, pp. 1172 – 1186.

Qu, B., Nie, C. & Xu, B. (2008), Test case prioritization for multiple processing queues, *in* 'Proceedings of the International Symposium on Information Science and Engineering, ISISE '08', Vol. 2, pp. 646 –649.

Rothermel, G., Harrold, M. J. & Dedhia, J. (2000), 'Regression test selection for C++ software', *Software Testing, Verification and Reliability* **10**(2), 77–109.

Rothermel, G., Untch, R., Chu, C. & Harrold, M. (2001), Prioritizing test cases for regression testing, *in* 'IEEE Transactions on Software Engineering', Vol. 27 of *10*, pp. 929 –948.

Sampath, S., Bryce, R., Viswanath, G., Kandimalla, V. & Koru, A. (2008), Prioritizing user-session-based test cases for web applications testing, *in* 'Proceedings of the 1st International Conference on Software Testing, Verification, and Validation', pp. 141 –150.

Sampath, S., Sprenkle, S., Gibson, E., Pollock, L. & Greenwald, A. (2007), Applying concept analysis to user-session-based testing of web applications, *in* 'IEEE Transactions on Software Engineering', Vol. 33 of *10*, pp. 643 –658.

Samuel, P., Mall, R. & Sahoo, S. (2005), Uml sequence diagram based testing using slicing, *in* '2005 Annual IEEE, INDICON', pp. 176 – 178.

Srikanth, H., Williams, L. & Osborne, J. (2005), System test case prioritization of new and regression test cases, *in* 'ISESE', pp. 64–73.

Torsel, A.-M. (2011), Automated test case generation for web applications from a domain specific model, *in* 'Proceedings of the IEEE 35th Annual Conference on Computer Software and Applications Conference Workshops (COMPSACW)', pp. 137 –142.

Tung, Y.-H., Tseng, S.-S., Lee, T.-J. & Weng, J.-F. (2010), A novel approach to automatic test case generation for web applications, *in* 'Proceedings of the 10th International Conference on Quality Software (QSIC)', pp. 399 –404.

Wong, W., Horgan, J., London, S. & Agrawal, H. (1997), A study of effective regression testing in practice, *in* 'Proceedings of The Eighth International Symposium On Software Reliability Engineering', pp. 264 –274.

Zimmermann, T. & Nagappan, N. (2007), Predicting subsystem failures using dependency graph complexities, *in* 'Proceedings of the 18th IEEE International Symposium on Software Reliability, ISSRE '07', pp. 227 –236.

A Paradox for Trust and Reputation In the E-commerce World

Han Jiao[1] **Jixue Liu**[1] **Jiuyong Li**[1] **Chengfei Liu**[2]

[1] School of Computer and Information Science
University of South Australia,
Adelaide, South Australia 5095,
Email: (han.jiao,jixue.liu,jiuyong.li)@unisa.edu.au

[2] Faculty of Information and Communication Technologies
Swinburne University of Technology ,
Melbourne, VIC3122, Australia,
Email: cliu@swin.edu.au

Abstract

Trust and reputation systems are widely adopted in the e-commerce environment to help buyers choose trustworthy sellers. It is a normal thought that the higher the reputation is, the more trustworthy its holder should be. However, our research discloses that under certain circumstances, a high-reputation seller has greater intention to cheat, which means that buyers should trust the low-reputation sellers better in those cases. We term this phenomenon Trust-Reputation Paradox. The theoretical proof, based on the game theory, is conducted to show the existence of the paradox. The root causes of this abnormality are revealed and discussed. In the end, we provide some guidelines for trust and reputation system designers to avoid this obscure pitfall.

Keywords: trust, reputation, trustworthiness, paradox, game theory

1 Introduction

The research on trust has been taken to the center stage in the field of electronic commerce, covering a wide range of topics (Momani & Challa 2010). After a long period of deliberation, some agreements have been made in understanding the concepts of trust, trustworthiness and reputation. Trust is the trustor's, i.e., an online buyer's, willingness to be vulnerable to the trustee, i.e., a seller, based on the belief that the trustee will act in a manner consistent with the trustor's expectation (Pavlou & Gefen 2004). Trustworthiness is an attribute of a trustee reflecting the extent that he/she is worthwhile to trust (Mayer et al. 1995, Gefen et al 2008). In (Jøsang et al. 2007), reputation is "what is generally said or believed about a person's or a thing's character or standing." The difference between reputation and trustworthiness is that reputation is a public opinion, while trustworthiness emphasizes a personal and subjective opinion (Wang & Vassileva 2007). They become two exchangeable concepts when putting them into the scope of the vast web where usually no private knowledge is assumed between a trustor and a trustee (Jøsang et al. 2007), because the absence of private knowledge makes the public opinions, i.e., reputation the only channel to get to know the trustee's character. From this perspective, reputation fully represents trustworthiness. Researchers design trust and reputation models to better represent the actual character of trustees so that a trustor can make use of it as reference.

Currently, two major methodologies are dominating the construction of trust and reputation models (Artz & Gil 2007, Bonatti et al. 2005): policy-based methods and reputation-based methods. Policy-based methods, such as (Olmedilla et al. 2004) and (Li et al. 2009), establish trust by exchanging digital credentials based on the defined policies or protocols. Reputation-based methods like (Maximilien & Singh 2002, Teacy et al. 2006, Rouhomaa et al. 2007), which this study concerns, predict whether trust can be formed through measuring reputation scores or a ranking list calculated by specific algorithms on historical data. Besides, there are some social network trust models such as (Nguyen et al. 2010) and (Bhuiyan et al. 2010) that use a chain of recommendations from different nodes to help form trust.

Many reputation models are studied (Wang & Vassileva 2007), although they seem to be quite vulnerable to attack (Hoffman et al. 2009). The model designers always make an implicit assumption that the higher the reputation is, the better the reputation holder can be trusted. Our research starts with verifying the assumption and discloses that the relationship between trust formation and reputation is more complex. A rational buyer should, sometimes, prefer trusting a low-reputation seller to a highly reputed one. We name this seemingly impossible phenomenon the Trust-Reputation Paradox. A game-theory model is used to illustrate what the paradox is, why it happens and how it impacts the reputation policies. The contributions of this paper are as follows:

- The Trust-Reputation Paradox is disclosed and proved, which indicates that the relationship between reputation and trust is not always "the-higher-the-better".
- A game-theory model is presented to reveal the the relationship between trust formation and reputation.
- Several guidelines on how to manage reputation are provided to help reputation system designers to avoid the paradox and to improve their models.

The paper is structured as follows. In Section 2, the research scenario is given together with some preliminary definitions. In Section 3, we present a model to illustrate the relationship between reputation and trust formation. In Section 4, we make further discussions on the model and reveal how and why the paradox would happen. Several guidelines to avoid falling into the paradox are provided as well. In the last section we make conclusions, limitation discussions and talk about the future work.

2 Scenario and Preliminaries

Assume that there is a n-seller e-commerce market. All the sellers provide similarly featured goods that offer the same functionalities but with different properties, such as price, quality and after-sale services. The n sellers hold reputations $r_1, r_2, ..., r_n$, respectively.

Definition 1 (Reputation Event). *In a reputation system, Reputation Events (RE) are the events whose occurrences cause reputation change. The set E of all reputation events is denoted by $E = \{e_1, e_2, ..., e_m\}$.*

The elements in Set E are the specific types of reputation events, like a "positive feedback event" or a "negative feedback event". There can be many types of REs in a reputation system. However, they are generally divided into two groups, the positive REs that cause reputation increase and the negative REs causing reputation decrease.

Definition 2 (Reputation Policy). *In a reputation system with RE set $E = \{e_1, e_2, ..., e_m\}$, Reputation Policy is the collection of rules defining how reputation changes (at different reputation values) with the occurrence of a possible RE. The Reputation Policy is denoted by a function $\rho(r, e)$, where r is the reputation of an entity and $e \in E$ is the RE causing the change. $\rho(r, e)$ is the amount of reputation change at reputation r if e occurs.* A good example to demonstrate reputation event and policy is the systems that use average of ratings as reputations. A lot of such systems have been implemented (Amazon 2012, Epinion 2012). Suppose a positive rating deserves $+1$ and a negative rating gained -1. Each time a rating being given to reputation holder is regarded as a reputation event. The final reputation of an entity is the average of all its ratings. If we use e_m^+ and e_m^- to represents the mth positive and negative ratings, after e_m^+ or e_m^- occurs, the reputation will become $\frac{r_i \times (m-1)+1}{m}$ and $\frac{r_i \times (m-1)-1}{m}$, respectively. Then, the reputation policy can be represented as

$$\begin{aligned} \rho(r_i, e_m^+) &= \tfrac{r_i \times (m-1)+1}{m} - r_i = \tfrac{-r_i+1}{m} \\ \rho(r_i, e_m^-) &= \tfrac{r_i \times (m-1)-1}{m} - r_i = \tfrac{-r_i-1}{m}. \end{aligned} \quad (1)$$

There are also cases in which reputation change is not related to reputation. For example, in eBay (eBay 2012) where the reputation system uses a simple summation method to calculate reputation, every reputation event (customer rating) will increase or decrease 1 point on the reputation. The amount of reputation change is fixed value rather than a function of reputation r. This can be seen as a special case of our reputation policy definition where the coefficient of r is zero.

Although reputation change is reflected by the superficial value marked on each reputation holder, it has more profound impact varying the potential benefit behind the mask. Because reputation is regarded as a reference to judge the character of an entity, a high reputation value represents a better character, which implies more potential safety and security in online interactions. These may bring the reputation

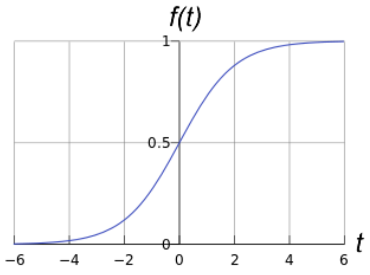


Figure 2: An example of a sigmoid function

holder more benefits. To measure the benefit of reputation, we define the concept of reputation utility.

Definition 3 (Reputation Utility). *Given a reputation holder whose reputation is r, the reputation utility of the holder, denoted by $U(r)$, is the benefit brought by its reputation. The function U is termed reputation utility function. We call $\mu(r) = U'(r)$ the reputation utility density function.*

Reputation utility can be represented in many forms. For example, it can be expressed as a part of the profit of an e-commerce seller. Suppose that a seller's profit of each goods unit is a constant value (the price and the cost do not change frequently). The seller's total profit depends on how many items he can sell out. If we further assume that the market volume in a certain duration is fixed, the seller's total profit is related to how much market share he seizes. In general, any kinds of benefits brought by reputation can be viewed as reputation utility.

Intuitively, a better reputation helps the seller gain more market share, which implies more utilities. However, different market environments may have different detailed relationships between reputation and reputation utility. In this case, a sigmoid function, such as $f(t) = \frac{1}{1+e^{-t}}$ is often used (Tang et al. 2012, Michalski et al. 1986).

For example, for a reputation system having the definition domain in $[0, r_{max}]$. The reputation utility function can be assumed to be a moved sigmoid function $U(r) = \frac{U_{max}}{1+e^{(-r+r^*)}}$, where r^* is the shift amount and U_{max} is the coefficient for scaling. Fig. 1(a) shows the general trend of such sigmoid function, whose properties can be generally described by the following statements.

1. The higher the reputation is, the more reputation utility it represents. $U(r)$ is a monotone increasing function of r, i.e., if $r_i \geq r_j$ $U(r_i) \geq U(r_j)$.

2. A threshold reputation utility exists between 0 and U_{max}, i.e., $\exists\ 0 \leq U_o \leq U_{max}$ such that $U(0) = U_o$. Even if a seller's reputation is down to 0, it still gains some utility.

3. There is r^* between 0 and r_{max} such that if $0 \leq r \leq r^*$, $U(r)$ is a convex function ($\mu(r)$ is increasing) and if $r^* \leq r \leq r_{max}$, $U(r)$ is a concave function ($\mu(r)$ is decreasing). r^* is a point of inflection.

Let us explain the above properties in detail. The first one complies with our common sense. The better the reputation is, the more benefit it can bring to its holder. The second one also reflects a common psychology of the buyers who usually would like to give new sellers some chances, which means that a new seller with an entry-level reputation value still gains some utility. The last property can also be understood from the buyer's view. A very normal thought

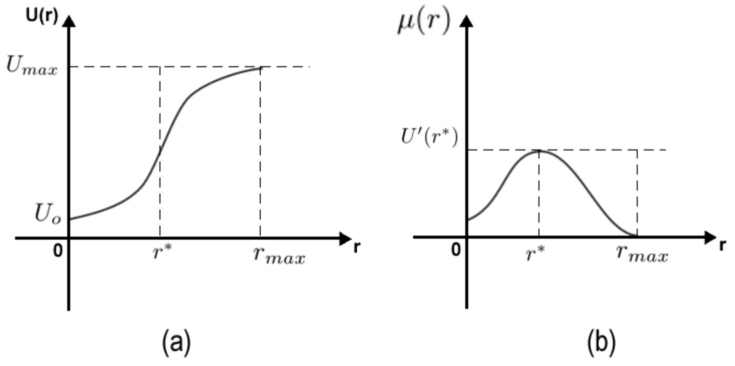


Figure 1: Reputation Utility Function Curves and Reputation Utility Density Function Curve

is that two sellers both with very high reputations would be treated as having almost the same level of trust. Buyers Choosing whom depends on their goods difference, rather than the reputations. Similarly, two sellers both with very low reputations are also of very slight difference, because their reputations are not worthwhile to notice. This phenomenon implies that when the reputation goes very high, any increment of reputation only invites very minor reputation utility increase and so is for otherwise low reputation decrease. This also complies with the law of marginal utility theories in Economics, which forms the reason of the third property. Note that r^* may not be exactly in the middle between 0 and r_{max}. At r^*, the reputation utility gains the highest changing speed.

The general curve of the density function $\mu(r)$ is in Fig. 1(b). As shown, $\mu(r)$ is non-negative in the entire domain. $\mu(r)$ increases with r in $[0, r^*]$ and decreases in $[r^*, r_{max}]$. Its maximum value is gained at r^*. Note that $\mu(r_{max})$ can be greater than, less than or equal to $\mu(0)$. The figure only gives an example.

Definition 4 (Reputation Utility Gain). *In a reputation system with utility function $U(r)$, given the difference of two reputations to be Δ, the increment of reputation utility along with the reputation increment Δ is termed the reputation utility gain. Let $G(r, \Delta) = U(r + \Delta) - U(r)$, $G(r, \Delta)$ is termed reputation utility gain function.*

$G(r, \Delta)$ is a very important function in our model. It can be either positive or negative, because Δ can be either positive or negative. Please also note that Δ may not be a fixed value. It can be a function of r or of other factors, depending on the reputation environment. We will discuss this later in detail.

The above four definitions pave the way for further analysis. In the next section, we start modeling buyer and seller transactions, from with we will draw the conditions of trust formation between them.

3 Linking Reputation and Trust: the Model of Online Trust Game

In this section, we propose a two-player two-stage sequential game model to describe the process of online transactions. The purpose is to find the relationship between reputation and trust formation. By judging when the equilibrium of the game falls down to "trusted behaviors", we get the conditions when a seller can be trusted.

3.1 The online Trust Game

The two players in the game are a buyer and a seller. At beginning, the buyer has to decide whether to trust the seller. If the buyer chooses not to trust the seller, the game finishes with zero payoffs for both players. Otherwise, the buyer must pay the cost to the seller before the seller delivers any goods/services. After receiving the payment, the seller decides either to deliver the promised goods, or to cheat. If the seller acts as promised, the buyer's requirements are fulfilled and the seller gets some benefit plus an increment of reputation utility. If the seller cheats, the buyer loses its payment and the seller gets a benefit that is greater than that of acting honestly. However, the seller's reputation will decrease which results in a decrease of reputation utility.

Fig. 3 gives the detailed payoffs, denoted in pairs by (x, y), in which x is for the buyer and y is for the seller. Since our focus is the seller's trustworthiness, the buyer's payoff is simply denoted as B and $-B'$. The payoff of the seller is the benefit from the sold items plus the utility gain from its reputation change. We assume that the occurrence of any actions as promised trigger a positive reputation event e_+ and any cheating actions trigger a negative reputation event e_-. If the seller behaves as promised, the buyer gets its benefit B. Meanwhile, the seller also gets the benefit p. Because the seller keeps the promise, his/her reputation increases and brings more reputation utility. Since the reputation change is $\rho(r, e_+)$, the increment of reputation utility can be denoted as $U(r+\rho(r, e_+))-U(r) = G(r+\rho(r, e_+), \rho(r, e_+))$. The utility increment stands for the increased sale volume caused by reputation increase. If the volume is in the number of items sold, the utility increment is volume increment over the total volume of the seller. It is expected to be a small fraction. By multiplying it with p, it is transformed into the profit increase of the seller due to the honest transaction. Similar things happen when the seller cheats. In that case, the seller will get the benefit p^* expected to be greater than p. But because of the reputation decrease, his/her profit will also decrease by $[U(r-\rho(r, e_-))-U(r)]p$. Please note that both $\rho(r, e_+)$ and $\rho(r, e_-)$ are positive values.

We assume that the game is of complete and perfect information (we will discuss this later). Trust can be formed if and only if the equilibrium yields to (trust and pay, act as promised). We use backward induction to find the conditions of such equilibrium and give the following theorem.

Theorem 1 (Trust Utilization Theorem). *In a buyer-seller online game in Fig.3, the buyer trusts the seller if and only if the reputation utility gain between $r+\rho(r, e_+)$ and $r-\rho(r, e_-)$ is greater than a constant*

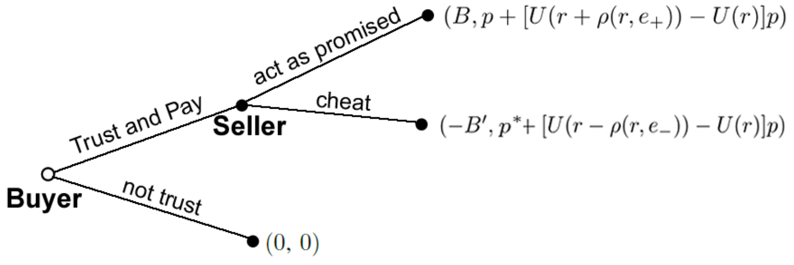

act as promised
$(B, p + [U(r + \rho(r, e_+)) - U(r)]p)$
Trust and Pay
Seller
cheat
$(-B', p^* + [U(r - \rho(r, e_-)) - U(r)]p)$
Buyer
not trust
$(0, 0)$

Figure 3: Online Trust Game

$\frac{p^*-p}{p}$, *i.e.*,

$$G(r + \rho(r, e_+), \rho(r, e_+) + \rho(r, e_-)) > \frac{p^* - p}{p} \tag{2}$$

Proof From backward induction, the equilibrium falls on (trust and pay, act as promised) when the following inequality holds:

$$p + [U(r + \rho(r, e_+)) - U(r)]p > \\ p^* + [U(r - \rho(r, e_-)) - U(r)]p$$

Because $p + [U(r + \rho(r, e_+)) - U(r)]p > 0$, the seller has the intention to encourage the buyer to trade with him/her. If the above inequality holds, the sellers gains less when cheating than acting as promised. The buyer knows this, so the buyer will choose to trust and get his requirement fulfilled.

By mathematical manipulation, we have

$$U(r + \rho(r, e_+)) - U(r - \rho(r, e_-)) > \tfrac{p^*-p}{p}$$

According to Definition 4, we have $G(r + \rho(r, e_+), \rho(r, e_+) + \rho(r, e_-)) = U(r + \rho(r, e_+)) - U(r - \rho(r, e_-))$. Therefore,

$$G(r + \rho(r, e_+), \rho(r, e_+) + \rho(r, e_-)) > \tfrac{p^*-p}{p}$$

■

Theorem 1 discloses a very important conclusion: whether a seller can be trusted does not depend on an absolute reputation value. It depends on the whether the difference of reputation utility gain after two different event occurrence is big enough. This invites two questions regarding to the paradox we mentioned before: 1)are there any cases that a high reputation holder, because it has a small reputation utility gain under the different event occurrences, is not worthwhile to trust? 2)are there any cases that a low reputation holder, because it has a large reputation utility gain under different reputation events, is trustworthy? Theoretically, we cannot eliminate this possibility. To further explore the question, we have to put them into some specific reputation policies.

4 The Conditions of Trust-Reputation Paradox Formation

We first adopt the accumulative reputation policy, which is used by the well-known e-commerce service provider eBay.com (eBay 2012), to demonstrate the paradox formation condition. Then we extend the conclusion into a typical case where linear reputation policy formulas are used. After that, we discuss on more general cases.

4.1 Possible Paradox in eBay

Originally, eBay used an accumulative reputation policy. A positive feedback increases 1 point on the reputation of a seller and a negative feedback decreases 1 point. The reputation of a seller is the summation of all the past feedback points. Later, eBay added the positive feedback rate as another indicator to better represent sellers' true character. Our analysis is only on the accumulative reputation policy.

In the eBay's accumulative reputation policy, there are only two reputation events e_+ and e_-, which represent the positive and negative event, respectively. The policy can be represented by $\rho(r, e_+) = \rho(r, e_-) = 1$. By putting it into the conclusion in Theorem 1, a seller in eBay is trustworthy if and only if the follow condition holds.

$$G(r + 1, 2) > \tfrac{p^*-p}{p}$$

Now we want to solve the above inequality of r to see the trustworthy range of a reputation holder. Let us first look at the monotonicity of $G(r + 1, 2)$. As previously defined, we assume the maximum reputation of the current market is r_{max}, the threshold reputation to start gaining utility is 0 and the inflection point is r^*. Since $G(r+1, 2) = U(r+1) - U(r-1)$, by differentiating the both sides, we have

$$G'(r + 1, 2) = U'(r + 1) - U'(r - 1)$$

Because $U'(r) = \mu(r)$, so

$$G'(r + 1, 2) = \mu(r + 1) - \mu(r - 1)$$

According to Fig. 1(b), $\mu(r)$ is monotone increasing in $[0, r^*]$ and is monotone decreasing in $[r^*, r_{max}]$. Because compare with the whole definition domain, the reputation change caused by one reputation event is quite minor. Therefore, $\mu(r + 1)$ is almost the same as $\mu(r)$. So we have $\mu(r + 1) - \mu(r - 1) \geq 0$ in $[0, r^*]$ and $\mu(r + 1) - \mu(r - 1) \leq 0$ in $[r^*, r_{max}]$, which means the following:

$$\begin{cases} G'(r + 1, 2) \geq 0 & \text{if } 0 \leq r \leq r^* \\ G'(r + 1, 2) \leq 0 & \text{if } r^* < r \leq r_{max} \end{cases} \tag{3}$$

Based on the above formula, in $[0, r^*]$, $G(r)$ is a monotone increasing function having a minimum value $G(0)$. In $[r^*, r_{max}]$, $G(r)$ is a monotone decreasing function having a minimum value $G(r_{max})$. In the whole definition domain, $G(r)$ gains its maximum value at r^*. This divided the problem into 5 different cases, as shown in Fig. 4.

If the maximum value $G(r^*)$ is smaller than $\frac{p^*-p}{p}$, as shown in Fig 4(a), $G(r)$ has no chances to satisfy the formula 2, which means that the seller always cheats and the current reputation policy fails to produce any incentives for the joined sellers.

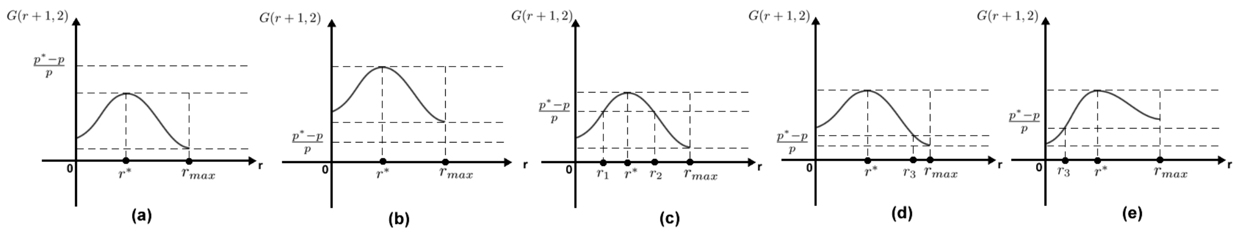


Figure 4: The 4 cases of trust-reputation paradox under eBay's reputation policy

If the minimum value $G(0)$ and $G(r_{max})$ are greater than $\frac{p^*-p}{p}$, as in Fig 4(b), the trust condition will always be satisfied. This implies that buyers should trust all the sellers regardless their reputations. All the sellers spontaneously act honestly and there is no need to impose a reputation policy. In the third situation shown in Fig. 4(c) where $\frac{p^*-p}{p}$ is between the maximum and minimum values, there must be r_1 in $[0, r^*]$ and r_1 in $[r_*, r_{max}]$ such that $G(r_1) = G(r_2) = \frac{p^*-p}{p}$. Buyers will only trust the sellers whose reputations are in $[r_1, r_2]$. This is where the the Trust-Reputation Paradox happens. In such a case, the buyer will prefer the sellers whose reputations are in $[r_1, r2]$ to the sellers whose reputations are even greater than r_2. In Fig. 4(d) where $\frac{p^*-p}{p}$ is between $G(0)$ and $G(r_{max})$ and $G(0) > G(r_{max})$, the paradox also happens. The reputation values that are greater than r_3 should not be trusted. The paradox also occurs. In Fig. 4(e), $\frac{p^*-p}{p}$ is also between $G(0)$ and $G(r_{max})$ but with $G(r_{max})$ and $G(0) < G(r_{max})$. The reputations greater than r_3 should be trustworthy. So there is no paradox happening.

So we find something weird that offends our common sense: sometimes, buyers prefer trusting sellers with low reputation rather than with high reputation. Actually, the reason behind the paradox is very reasonable. $U(r)$ is a concave function in $[r^*, r_{max}]$, which means that the higher the reputation is, the less market it brings for every increment of reputation. When a reputation point r is less than r_1 of Fig.4 (b), every increment of reputation only increases a so tiny sale volume that is not enough to bring the benefit to compensate the difference between a normal product sale and a lemon product sale, the seller has the motivation to cheat. We here name that reputation point **Utility Upper Limit (UUL)**. In the eBay cases, this is r_2 in the Fig. 4(c). If the highest reputation in the entire definition domain is lower than UUL, the sellers will never get to the upper limit. The paradox will never happen as well. On the other hand, when the reputation is lower than a certain point from which every increment of reputation only increases such a small market share that can not bring enough benefit to compensate the difference of the benefit between a normal product sale and a lemon goods sale, the sellers will also cheat. We term the reputation point **Utility Lower Limit (ULL)**. The sellers who hold the reputations lower than ULL do not care to damage their reputation because it is not worthwhile to cherish. Only the sellers in between care their reputations.

4.2 The Paradox Formation in Linear Reputation Policies

In section 4.1, we discussed the paradox under the eBay accumulative reputation policy. In this section, we consider linear reputation policies. A linear reputation policy means that the reputation change is a linear function of reputation r.

Reputation, in our common sense, is hard to get but easy to lose. In practice, most linear reputation polices also follow this point. This implies that when reputation is increasing, its increase speed will become lower as the reputation becomes higher. When reputation is decreasing, its decrease speed will also become lower as the reputation goes down. Reflected in the reputation policy functions, $\rho(r, e_+)$ is a linear function of r with a negative gradient and $\rho(r, e_-)$ is a linear function of r with a positive gradient. Suppose $\rho(r, e_+) = -\alpha \cdot r + \beta$ and $\rho(r, e_-) = \alpha \cdot r + \beta_2$, in which $\alpha \geq 0$ and $\beta, \beta_2 \geq 0$. Since we assume the reputation domain is $[0, r_{max}]$, at $r = 0$ the reputation cannot decrease any more. This means that the reputation decrease function is a linear function crossing the original point. Therefore, we have $\beta_2 = 0$. Their curves are presented in Fig. 5(a). The eBay's reputation policy function is given in Fig. 5(b), which can be seen as special case of linear reputation policies. The trust-reputation paradox for linear reputation policies can be concluded by the following lemmas and the theorem.

Lemma 1 (Monotonicity of Reputation Utility Gain $\alpha < 1$). *In a buyer-seller online game in Fig.3 with linear reputation policy, let $\Delta = \rho(r, e_+) + \rho(r, e_-) = \beta$, when $\alpha < 1$, reputation utility gain function is a monotone increasing function in $[0, \frac{r^*-\beta}{1-\alpha}]$ and is a monotone decreasing function in $[\frac{r^*-\beta}{1-\alpha}, \frac{r_{max}-\beta}{1-\alpha}]$. Its maximum value is gained at $\frac{r^*-\beta}{1-\alpha}$.*

Proof Based on the definition 4, we have

$$G(r+\rho(r, e_+), \Delta) = U(r+\rho(r, e_+)) - U(r-\rho(r, e_-)).$$

By differentiating both sides, we get

$$G'(r+\rho(r, e_+), \Delta) = U'(r+\rho(r, e_+)) - U'(r-\rho(r, e_-)).$$

Since $U'(r) = \mu(r)$, we have $G'(r + \rho(r, e_+), \Delta) = \mu(r + \rho(r, e_+))(1 + \rho'(r, e_+)) - \mu(r - \rho(r, e_-))(1 - \rho'(r, e_-))$.

Because $\rho(r, e_+) = -\alpha \cdot r + \beta$, we have $\rho'(r, e_+) = -\alpha$. Similarly, $\rho'(r, e_-) = \alpha$, by putting all of them into the $G'(r + \rho(r, e_+), \Delta)$, we get

$$G'(r - \alpha r + \beta, \beta) = (1 - \alpha)[\mu(r - \alpha r + \beta) - \mu(r - \alpha r)] \quad (4)$$

When $\alpha < 1$, according to Fig. 1(b), $\mu(r)$ is monotone increasing in $[0, r^*]$ and is monotone decreasing in $[r^*, r_{max}]$. So $\mu(r - \alpha r + \beta)$ is monotone

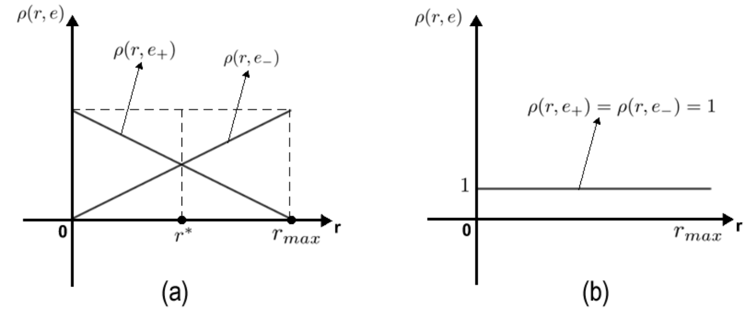


Figure 5: The general trend of linear reputation policies and one of its special case, which is the eBay reputation policy.

increasing in $[0, \frac{r^*-\beta}{1-\alpha}]$ and is monotone decreasing in $[\frac{r^*-\beta}{1-\alpha}, \frac{r_{max}-\beta}{1-\alpha}]$. The positive $(1-\alpha)$ does not change the monotonicity. Therefore,

$$\begin{cases} G'(r-\alpha r+\beta,\beta) \geq 0 & \text{if } 0 \leq r \leq \frac{r^*-\beta}{1-\alpha} \\ G'(r-\alpha r+\beta,\beta) \leq 0 & \text{if } \frac{r^*-\beta}{1-\alpha} < r \leq \frac{r_{max}-\beta}{1-\alpha} \end{cases} \tag{5}$$

Formula (5) shows that the reputation utility gain $G(r-\alpha r+\beta,\beta)$ is an increasing function when $0 \leq r \leq \frac{r^*-\beta}{1-\alpha}$ and is an decreasing function when $\frac{r^*-\beta}{1-\alpha} < r \leq \frac{r_{max}-\beta}{1-\alpha}$. The maximum value of $G(r-\alpha r+\beta,\beta)$ is gained at $\frac{r^*-\beta}{1-\alpha}$. The minimum value is gained at 0 and $\frac{r_{max}-\beta}{1-\alpha}$. ■

Similarly, we can conclude the monotonicity of reputation utility gain when $\alpha > 1$ and $\alpha = 1$, as in the following Lemmas. The proofs are left out s-ince they are very similar to Lemma 1.

Lemma 2 (Monotonicity of Reputation Utility Gain $\alpha > 1$). *Assume all the conditions in Lemma 1 except for $\alpha > 1$, reputation utility gain function is a monotone decreasing function in $[0, \frac{r^*-\beta}{1-\alpha}]$ and is a monotone increasing function in $[\frac{r^*-\beta}{1-\alpha}, \frac{r_{max}-\beta}{1-\alpha}]$. Its minimum value is gained at $\frac{r^*-\beta}{1-\alpha}$.*

Lemma 3 (Monotonicity of Reputation Utility Gain $\alpha = 1$). *Assume all the conditions in Lemma 1 except for $\alpha = 1$, reputation utility gain between $r + \rho(r, e_+)$ and $r - \rho(r, e_-)$ is a constant $G(\beta, \beta)$.*

Lemma 1 to Lemma 3 give the conclusions about the monotonicity of reputation utility gain function. They pave the way for the following discussion on the trust formation and the paradox occurrence. Basical-ly, if trust can be formed in a low reputation range but cannot be formed in a high reputation range, the Trust-Reputation Paradox will happen. We still use the conclusion from Theorem 1 to determine whether a reputation value is trustworthy. According to the theorem, trust can be formed only if the Formula 2 holds. Under linear reputation policy, Formula 2 is transformed into the following format.

$$G(r-\alpha r+\beta,\beta) > \frac{p^*-p}{p}$$

The left side of the above formula is the reputa-tion utility gain after positive and negative reputation event. The righthand side $\frac{p^*-p}{p}$ represents how many times of extra benefit that seller can obtain if he/she cheats. It varies in different markets for different sell-ers. The above formula has close relationship with the equation $G(r-\alpha r+\beta,\beta) = \frac{p^*-p}{p}$. We call the equa-tion "KEY equation" and denote $R = r_1, r_2, ...r_s$ to be the set containing all the real solution for the equa-tion. Under linear reputation policy and a sigmoid-like utility function, the equation sometimes does not have any real solutions and sometimes has 1 or 2 so-lutions. The trust formation and paradox occurrence have close relationship with those real solutions. We conclude them into the following two lemma and the-orem.

Lemma 4 (Trust Formation for Small Utility Gain). *Assume all the conditions in Lemma 1-3 and denote the maximum value of reputation utility gain function to be G_{max}, if $\frac{p^*-p}{p} > G_{max}$, there is no real solutions for KEY Equation. All sellers are not trustworthy, regardless their reputations.*

Proof From Theorem 1, trust can be formed when the following inequality holds:

$$G(r+\rho(r,e_+), \rho(r,e_+)+\rho(r,e_-)) > \frac{p^*-p}{p} \tag{6}$$

Because $G(r+\rho(r,e_+), \rho(r,e_+)+\rho(r,e_-)) \leq G_{max}$ and $\frac{p^*-p}{p} > G_{max}$, the above inequality will never hold. So trust can not be formed regardless reputa-tion. ■

Lemma 5 (Trust Formation for Big Utility Gain). *Assume all the conditions in Lemma 1-3 and denote the minimum value of reputation utility gain function to be G_{min}, if $\frac{p^*-p}{p} < G_{min}$, there is no real solu-tions for KEY Equation. All sellers are always be trustworthy, regardless their reputations.*

The proof is left out since it is similar to that of Lemma 4.

The situation becomes more complex if $\frac{p^*-p}{p}$ is between G_{max} and G_{min}. It generally divided into 6 cases that are concluded by the following theorem. The theorem excludes the cases when $\alpha = 0$, since in such a case the reputation utility gain is a fixed value (Lemma 3), which means that we only need to sim-ply compare a fixed value with $\frac{p^*-p}{p}$ to determine the trust formation. This situation may incur the occur-rence of the paradox, therefore we name the following theorem Trust-Reputation Paradox Theorem.

Theorem 2 (Trust-Reputation Paradox Theo-rem)*In a buyer-seller online game in Fig.3 with linear reputation policy, if $\frac{p^*-p}{p}$ is between the minimum val-ue and the maximum value of reputation utility gain function in the whole reputation domain, trust can only be formed at the reputations where the reputa-tion utility gain is greater than $\frac{p^*-p}{p}$.*

Proof There are 6 different cases as shown in Fig. 6(a)-(f). They can be summarized as follows.

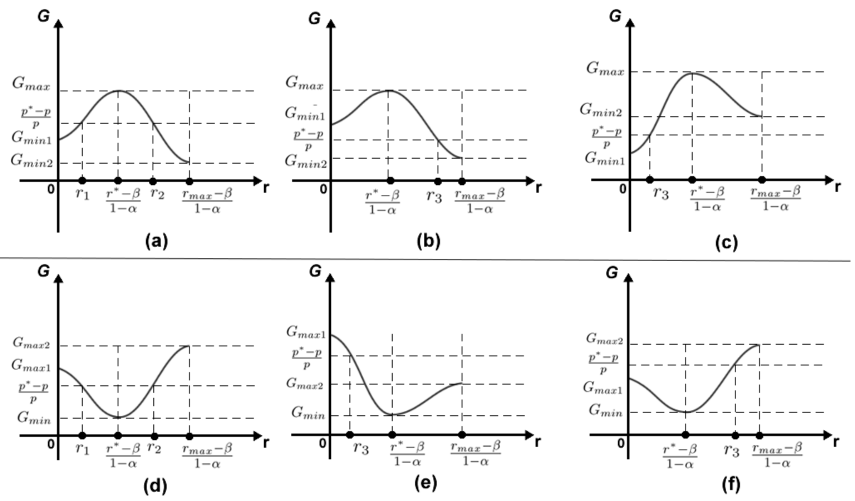


Figure 6: The conditions of paradox formation for the linear reputation policies. (a),(b) and (c) are for $\alpha < 1$. (d),(e) and (f) are for $\alpha > 1$.

1) when $\alpha < 1$, reputation utility gain function reaches its maximum value (G_{max}) at $\frac{r^*-\beta}{1-\alpha}$ and two minimum values (G_{min1} and G_{min2}) at 0 and $\frac{r_{max}-\beta}{1-\alpha}$, respectively

1. if $G_{max} \leq \frac{p^*-p}{p} \leq max(G_{min1}, G_{min2})$ as shown in Fig. 6(a), there are r_1 in $[0, \frac{r^*-\beta}{1-\alpha}]$ and r_2 in $[\frac{r^*-\beta}{1-\alpha}, r_{max}]$ so that $G(r_1 - \alpha r_1, \beta) = G(r_2 - \alpha r_2, \beta) = \frac{p^*-p}{p}$. Sellers are only trustworthy when their reputations are between r_1 and r_2.
2. if $G_{max} \geq G_{min1} \geq G_{min2}$ and $G_{min1} \leq \frac{p^*-p}{p} \leq G_{min2}$ as shown in Fig. 6(b), there is r_3 such that $G(r_3 - \alpha r_3, \beta) = \frac{p^*-p}{p}$. Sellers are only trustworthy when their reputations are less than r_3.
3. if $G_{max} \geq G_{min1} \geq G_{min2}$ and $G_{min1} \leq \frac{p^*-p}{p} \leq G_{min2}$ as shown in Fig. 6(c), there is r_3 such that $G(r_3 - \alpha r_3, \beta) = \frac{p^*-p}{p}$. Sellers are only trustworthy when their reputations are greater than r_3.

2) when $\alpha > 1$, reputation utility gain function reaches its minimum value (G_{min}) at $\frac{r^*-\beta}{1-\alpha}$ and two maximum values (G_{max1} and G_{max2}) at 0 and $\frac{r_{max}-\beta}{1-\alpha}$, respectively

1. if $G_{min} \leq \frac{p^*-p}{p} \leq min(G_{max1}, G_{max2})$ as shown in Fig. 6(d), there are r_1 in $[0, \frac{r^*-\beta}{1-\alpha}]$ and r_2 in $[\frac{r^*-\beta}{1-\alpha}, r_{max}]$ so that $G(r_1 - \alpha r_1, \beta) = G(r_2 - \alpha r_2, \beta) = \frac{p^*-p}{p}$. Sellers are only trustworthy when their reputations are not between r_1 and r_2.
2. if $G_{max1} \geq G_{max2} \geq G_{min}$ and $G_{max2} \leq \frac{p^*-p}{p} \leq G_{max1}$ as shown in Fig. 6(e), there is r_3 such that $G(r_3 - \alpha r_3, \beta) = \frac{p^*-p}{p}$. Sellers are only trustworthy when their reputations are less than r_3.
3. if $G_{max2} \geq G_{min1} \geq G_{min}$ and $G_{max1} \leq \frac{p^*-p}{p} \leq G_{max2}$ as shown in Fig. 6(f), there is r_3 such that $G(r_3 - \alpha r_3, \beta) = \frac{p^*-p}{p}$. Sellers are only trustworthy when their reputations are greater than r_3.

All the mentioned r_1, r_2 and r_3 are the real solutions of the KEY equation $G(r - \alpha r + \beta, \beta) = \frac{p^*-p}{p}$ in the corresponding cases. The proof for each case is similar. For example, in the first case as shown in Fig. 6(a) where $G_{max} \leq \frac{p^*-p}{p} \leq max(G_{min1}, G_{min2})$, from Theorem 1, trust can be formed when the following inequality holds:

$$G(r + \rho(r, e_+), \rho(r, e_+) + \rho(r, e_-)) > \frac{p^*-p}{p}$$

The reputation utility gain between r_1 and r_2 is greater $\frac{p^*-p}{p}$, therefore, only $[r_1, r_2]$ is a trustworthy reputation range. The proof of other cases are similar to this and are left out. ■

In the case of Fig. 6(a)(b)(d)(e), paradox will occur because there are some untrustworthy range in high reputations but trustworthy range in low reputations. For example, In (a), reputations greater than r_2 are not trustworthy but reputations between r_1 and r_2 are trustworthy which gives us a show of paradox. In some cases (Fig. 6(a)(d)), the equation have two real solutions. In other cases, it has only one real solutions. In fact, Lemma 4 and Lemma 5 give the situations where the equation has no real solutions. Once given the detailed format of the reputation policy function ρ and the utility gain function G, we are able to solve the equation and then discuss the trust formation issue. In the next section, we will give a use case to determine these key points.

From Theorem 2, we know that for linear reputation policies, the gradient α of $\rho(r, e)$ is very important. The numeric relationship between the gradient and 1 determines the monotonicity of utility gain function, and therefore, determines the conditions that the paradox occur. Up to now, we can safely conclude that when $\frac{p^*-p}{p}$ is in between the maximum utility gain (G_{max} in Theorem 2) and the minimum utility gain (G_{min} in Theorem 2), the paradox may happen. At that time, the trustworthy range of reputation is still upon the gradient α.

Table 1: Seller Information in Use Case

Reputation	Utility	Market Share
73.2	0.0000075045	0.0000007166
61.3	0.0000000001	0.0000000000
61.0	0.0000000000	0.0000000000
87.5	0.9241418200	0.0882463138
80.0	0.0066928509	0.0006391004
73.9	0.0000151121	0.0000014431
57.0	0.0000000000	0.0000000000
58.1	0.0000000000	0.0000000000
98.5	0.9999986290	0.0954898814
92.4	0.9993891206	0.0954316795
59.2	0.0000000000	0.0000000000
77.1	0.0003706061	0.0000353892
69.8	0.0000002505	0.0000000239
85.4	0.5986876601	0.0571686921
67.2	0.0000000186	0.0000000018
99.8	0.9999996264	0.0954899767
89.5	0.9890130574	0.0944408691
80.1	0.0073915413	0.0007058184
53.8	0.0000000000	0.0000000000
80.6	0.0121284350	0.0011581444
96.5	0.9999898700	0.0954890450
54.2	0.0000000000	0.0000000000
88.4	0.9677045353	0.0924061180
72.5	0.0000037266	0.0000003559
88.7	0.9758729786	0.0931861228
91.9	0.9989932292	0.0953938758
66.0	0.0000000056	0.0000000005
89.8	0.9918374288	0.0947105683
75.3	0.0000612797	0.0000058516
69.1	0.0000001244	0.0000000119

4.3 A Use Case

Now we give a use case to demonstrate how to use Theorem 2 to distinguish whether a reputation environment is sufferring from the trust-reputation paradox. The scenario of the use case is artificially created only for the demonstration purpose.

Suppose that there is a 30-seller reputation-based e-commerce platform. The sellers are selling "similarly featured goods", otherwise it makes no sense to compare their reputations. "Similarly featured goods" means that the goods can offer the same type of functionalities or can fulfill the same sort of requirement for customers. For example, when two TV set sellers supply "similarly featured goods", their TV sets are all used in the same way. Similarly featured goods do not imply that the goods are identical. The two TV set sellers can provide different brands of TV sets with different quality attributes and prices.

We artificially generate all the sellers' reputations by using a random number generation program. The reputation definition domain is [0,100]. The generated reputations follow a normal distribution with the the mean of 75 and the variance of 15. We use the sigmoid function $U(r) = \frac{1}{1+e^{(-r+85)}}$ as reputation utility function. The *ith* seller's market share is determined by $\frac{U(r_i)}{\sum_{j=1}^{30} U(r_j)}$, where the denominator is actually the summation of all the sellers' reputation utility. Detailed data are given in Table 1. We also define the reputation policy of this environment to be linear functions. The positive and negative reputation change functions are $\rho(r, e_+) = -0.1r + 10$ and $\rho(r, e_+) = 0.1r$. In a certain period, the whole market can sell 100 items, which can be treated as market volume $V = 100$.

Now assume that there is a new male seller whose

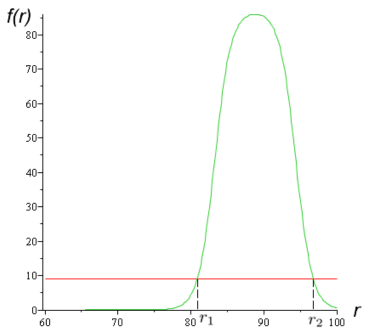


Figure 7: The condition of paradox in the use case

reputation is r. His average profit (p) of honestly selling one item is 10. But if he cheats, he will gain more profit ($p^* = 100$). We would like to know whether the paradox will happen if the given seller's information remains unchanged. If the paradox is going to happen, what are the reputation range that the paradox is going to happen? At beginning, the new seller joins the market with reputation r, his original market share will be $\frac{U(r)}{U(r)+10.4722994109}$, in which 10.4722994109 is the summation of all the reputation utility from the Column 2 in Table 1. If he cheats, his reputation will become $r - 0.1r = 0.9r$. Then his market share will be $\frac{U(0.9r)}{U(0.9r)+10.4722994109}$. If he acts honestly, his reputation will become $r - 0.1r + 10 = 0.9r + 10$. Then his market share will be $\frac{U(0.9r+10)}{U(0.9r+10)+10.4722994109}$. Because $U(r) = \frac{1}{1+e^{(-r+85)}}$ and $\alpha < 1$, according to Theorem 2, the condition of the seller not cheating is as follows:

$$\begin{aligned}&\Big(\frac{\frac{1}{1+e^{(-0.9r+75)}}}{10.4722994109+\frac{1}{1+e^{(-0.9r+75)}}} - \\ &\frac{\frac{1}{1+e^{(-0.9r+85)}}}{10.4722994109+\frac{1}{1+e^{(-0.9r+85)}}}\Big) \times V \times p > \\ &> \frac{p^*-p}{p}\end{aligned} \qquad (7)$$

By denoting the left side of the above inequality as $f(r)$, we could draw the curve of it, shown in Fig. 7. The basic shape of the diagram is quite similar to the one in Fig 6(a). As we expected, the reputation utility is small. This is determined by the environment rather than the our calculation method. The y-axis stands for the reputation utility gain between cheating and acting honestly. The x-axis is reputation. The chart shows that between $r_1 = 80.68526866$ and $r_2 = 96.88983853$, the above inequality holds. That means if the reputation is lower than 80.68526866 or greater than 96.88983853, the seller will be not trustworthy. Therefore, a paradox turns up that a high reputation holder with the reputation greater than 96.88983853 should not be trusted. This implies that the our assumed reputation environment is being impacted by trust-reputation paradox.

4.4 Some Discussion on Non-linear Reputation Policies

Non-linear reputation policies can be in countless forms, which means that fixed forms for a general discussions is not possible. In most cases, we have to use Theorem 1 as a general extermination rule to analyze each of them as a specific case. But there exists a general rule for a small portion of non-linear policies, which can be summarized as the following corollary, based on Theorem 2.

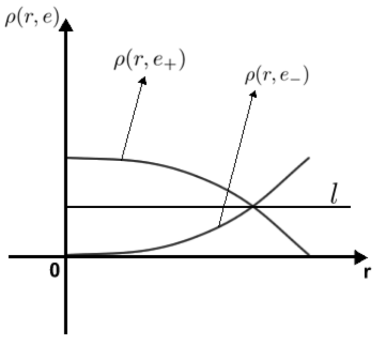


Figure 8: An example of a non-linear reputation policy

Corollary 1 (About Symmetrical Non-learning Policy). *Assume all conditions of Theorem 2. Let ρ be symmetric to and cross line $u = l$ (parallel to r-axis)and is also crossing the intersection of $\rho(r, e_+)$ and $\rho(r, e_-)$, the conclusion of Theorem 2 also stands.*

Proof Let us paste the formula from Lemma 1 to here.

$$G'(r + \rho(r, e_+), \Delta) = \mu(r + \rho(r, e_+))(1 + \rho'(r, e_+)) - \mu(r - \rho(r, e_-))(1 - \rho'(r, e_-))$$

If $\rho(r, e_+)$ and $\rho(r, e_-)$ are symmetric about a line that is parallel to r-axis and that also is crossing the intersection of $\rho(r, e_+)$ and $\rho(r, e_-)$, at any reputation point r, $\rho'(r, e_+) = -\rho'(r, e_-)$. An example curve is presented in Fig. 8. The above formula can be reformed as

$$G'(r + \rho(r, e_+), \Delta) = (1 + \rho'(r, e_+))[\mu(r + \rho(r, e_+)) - \mu(r - \rho(r, e_-))]$$

Whether $G'(r + \rho(r, e_+), \Delta)$ is positive is also determined by the monotonicity of $\mu(r + \rho(r, e_+))$ and $1 + \rho'(r, e_+)$. The rest of the proof is same as the process in Theorem 2, except that a fixed α as the first-order derivative of reputation policy function is replaced by $\rho'(r, e_+)$. So the details are left out.

■

In fact, most non-linear reputation policies are designed in the mentioned symmetrical way. The reason is that people expect that regardless how much reputation is increased by positive events, they should be able to be decreased by negative events. A very common non-linear reputation policy is shown in Fig. 8. In this reputation policy, reputation increase speed will go down as reputation goes up. But reputation decreasing speed will go high as reputation goes up. This implies that in such policies, reputation is very hard to gain but quite easy to lose. It follows our common senses towards reputation.

4.5 Some Guidelines to Avoid the Paradox

The Trust-Reputation Paradox makes reputation systems lose their functionalities to provide references to buyers. Therefore, we should try our best to avoid its occurrence. We here provide some guidelines for reputation system designers to avoid the paradox occurrence in their systems.

- Whether trust can be formed from reputations does not depend on the absolute reputation value. It depends on how reputation change will incur the behind utility change. In e-commerce market, the utility finally manifests into the profit of the sellers. Before starting the system design work, it is a good idea to know how reputation is related to reputation utilities.
- Any reputation platforms may involve several kinds of goods for trading. Note that an effective reputation policy for a kind of goods does not have the universal effectiveness for all kinds of goods. $\frac{p^*-p}{p}$ is a crucial parameter that we used to compare with reputation utility gain. This parameter is different from goods to goods. So make your reputation policy customizable from goods to goods is also a good choice.
- A clear signal for Trust-Reputation Paradox is that there is a large amount of sellers who have their reputations at a certain level and do not want to improve them.
- It is a good idea to make reputation hard to gain and easy to lose. However, if it is too hard to gain, sellers may give up. Because they find that they have to spend too much cost to gain a decent reputation that cannot bring their adequate rewards. In that case, the paradox occurs.
- We recommend you to frequently survey the middle-ranked and low-ranked holder to detect the strength of their intention to improve their reputation. We strongly recommend to constantly survey the high-ranked holders to detect the strength of their intention to maintain their reputation. If the strength is low, the paradox may be approaching.
- If the paradox is on show in the reputation systems, try to find whether the reputation policy makes the reputation utility change too fast or too slow. Then make adjustment on the policy.
- Using a multi-dimensional indicator as reputation is a very good idea to avoid the paradox, since even some of the dimensions have been trapped in the paradox, others are still functioning. The customers still can get a meaningful references from the reputations.

5 Conclusion, Limitation and Future Work

In this paper, we design a game-theory model to construct the bridge between reputation and trust formation. Based on the model, we give theoretical proofs of the existence of the so-called trust-reputation paradox, which offends a common sense that reputation should be "the higher the better". Our proofs cover all the situations for linear reputation policies and parts of non-linear reputation policies. We also provide some guidelines for reputation policy designer to better off their work.

Our work has the following limitations:

1. For non-linear reputation policies that are not symmetric to the line parallel to r-axis and crossing the intersection of $\rho(r, e_+)$ and $\rho(r, e_-)$, we can not give a universal conclusion to tell when the paradox is going to happen.
2. We did not provide methods to determine the relationship between reputation utility and reputation, which means that detailed format of reputation utility function $U(r)$ is hard to define.

3. Although the conclusion we give is theoretically proven, we still need to use real-world data to test the actual existence of the paradox.

We believe that the non-linear reputation policies limitation is difficult to solve, since there is no general regular patterns can be drawn from those reputation policies. The last two limitations form our future work. We will do research on 1) determining, or at least instantiatedly determining the relationship between reputation and reputation utility; and 2) finding real-world examples to prove the existence of the paradox in our life.

References

Amazon (2012). A famous e-commerce web site. www.amazon.com.

Artz, D. and Gil, Y. (2007), A survey of trust in computer science and the Semantic Web, Journal of Web Semantics, 5(2), 58-71.

Bhuiyan, T., Xu, Y., Josang, A., Liang, H. and Cox, C. (2010), Developing trust networks based on user tagging information for recommendation making, in The 11th International Conference on Web Information Systems Engineering (WISE), pp 357-364. 357-364.

Bonatti, P., Duma, C., Olmedilla, D. and Shahmehri, N. (2005), An Integration of Reputation-based and Policy-based Trust Management, in The 1st Semantic Web and Policy Workshop in the 4th International Semantic Web Conference.

eBay (2012). A Famous Online Market Place. http://www.ebay.com.

Epinion (2012). A famous web site for product review and rating. www.epinion.com.

Gefen, D., Benbasat, I. and Pavlou, P. (2008), A Research Agenda for Trust in Online Environments, Journal of Management Information Systems, 24(4), 275-286.

Hoffman, K., Zage, D. and Nita-Rotaru, C. (2009), A Sruvey of Attack and Defense Techniques for Reputation Systems, ACM Computing Survey, 42(1), 1-31.

Jøsang, A., Ismail, R. and Boyd, C. A. (2007), A survey of trust and reputation systems for online service provision, Decision Support Systems, 43(2), 618-644.

Li, J., Li, N. and Winsborough, W. H. (2009), Automated trust negotiation using cryptographic credentials, ACM Transactions on Information and System Security (TISSEC), 12(1), 1-35.

Mayer, R. C., Davis, J. H. and Schoorman, F. D. (1995), An Integrative Model of Organizational Trust, The Academy of Management Review, 20(3), 709-734.

Maximilien, E. M. and Singh, M. P. (2002), Conceptual model of web service reputation, ACM SIGMOD Record, 31(4), 36-41.

Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (1986), Machine Learning: An Artificial Intelligence Approach, Volume II, Morgan Kaufmann.

Momani, M. and Challa, S. (2010), Survey of Trust Models in Different Network Domains, International Journal of Ad hoc, Sensor & Ubiquitous Computing (IJASUC), 1(3), 1-19.

Nguyen, H. T., Zhao, W. and Yang, J. (2010), A Trust and Reputation Model Based on Bayesian Network for Web Services, in The 8th IEEE International Conference on Web Services (ICWS), pp 251-258.

Olmedilla, D., Lara, R., Polleres, A. and Lausen, H. (2004), Trust Negotiation for Semantic Web Services, in The 1st International Workshop on Semantic Web Services and Web Process Composition, pp 81-95.

Ruohomaa, S., Kutvonen, L. and Koutrouli, E. (2007), Reputation Management Survey, in The Second International Conference on Availability, Reliability and Security (ARES), pp 103-111.

Pavlou, P. A. and Gefen, D. (2004), Building Effective Online Marketplaces with Institution-Based Trust, Information Systems Research, 15(1), 37-59.

Tang, J., Gao, H., Liu, H. and Sarma, A. D. (2012), eTrust: Understanding Trust Evolution in an Online World, in The 18th ACM Conference on Knowledge Discovery and Data Mining (SIGKDD).

Teacy, W. T. L., Patel, J., Jennings, N. R. and Luck, M. (2006), TRAVOS: Trust and Reputation in the Context of Inaccurate Information Sources, Autonomous Agents and Multi-agent Systems, 12(2), 183-198.

Wang, Y. and Vassileva, J. (2007), A Review on Trust and Reputation for Web Service Selection, in The 27th International Conference on Distributed Computing Systems Workshops (ICDCSW), pp 25-33.

A Performance Cost Evaluation of Aspect Weaving

Miguel García[1] **Francisco Ortin**[1] **David Llewellyn-Jones**[2] **Madjid Merabti**[2]

[1] Computer Science Department, University of Oviedo
Calvo Sotelo s/n, 33007, Oviedo, Spain
Email: `garciarmiguel@uniovi.es`, `ortin@lsi.uniovi.es`

[2] School of Computing & Mathematical Sciences, Liverpool John Moores University
Byrom Street, Liverpool L3 3AF, United Kingdom
Email: `D.Llewellyn-Jones@ljmu.ac.uk`, `M.Merabti@ljmu.ac.uk`

Abstract

Aspect-Oriented Software Development (AOSD) facilitates the modularisation of different crosscutting concerns in software development. In AOSD, aspect weaving is the composition mechanism that combines aspects and components in an aspect-oriented application. Aspect weaving can be performed statically, at load time or at runtime. These different kinds of weavers may entail a runtime performance and a memory consumption cost, compared to the classical object-oriented approach. Using the *Dynamic and Static Aspect Weaving* (DSAW) AOSD platform, we have implemented three different scenarios of security issues in distributed systems (access control / data flow, encryption of transmissions, and FTP client-server). These scenarios were developed in both the aspect-oriented and object-oriented paradigms in order to evaluate the cost introduced by static and dynamic aspect weavers. A detailed quantitative evaluation of runtime performance and memory consumption is presented.

Keywords: Aspect-oriented software development, runtime performance, memory consumption, aspect weaving, DSAW.

1 Introduction

The *Aspect-Oriented Software Development (AOSD)* (Irwin et al. 1997) paradigm allows developers to make good use of the *Separation of Concerns* (SoC) principle (Hürsch & Lopes 1995) when developing applications. AOSD offers a direct support to modularise different functionalities that cut across system software. The modularisation of crosscutting concerns prevents tangling of the application source code, making it easier to debug, maintain and modify (Parnas 1972). Typical examples of crosscutting concerns are persistence, authentication, logging and tracing (Ortin et al. 2004). The process of integrating aspects into the main application code is called *weaving* and an aspect *weaver* is the tool that performs it (Ackoff 1971). The weaving process can be performed statically (compile time or load time) or dynamically (at runtime). Dynamic weaving AOSD platforms offer a powerful mechanism to dynamically adapt running applications, modifying their functionality while the system is being executed (Popovici et al. 2002). Application concerns can be modified, inserted or removed without stopping the application execution. However, in some scenarios, the use of AOSD may involve an increase of runtime performance and memory consumption (Garcia et al. 2012).

There are specific scenarios where it is necessary to adapt running applications in response to runtime emerging requirements (Vinuesa et al. 2008), such as distributed systems security (Garcia et al. 2012). Distributed systems involve the interaction between disparate and independent entities working toward a common goal (Belapurkar et al. 2009). As the number and arrangement of these potentially mobile entities may change, these systems are commonly required to be flexible and scalable. Under these circumstances, security in distributed systems is a complex issue to be considered. The security concerns of distributed systems can be modularised using AOSD, becoming possible to adapt the security measures without compromising their global security, even when their sizes and arrangements change at runtime (Garcia et al. 2012).

Our objective is to compare the runtime performance and the memory consumption of aspect-oriented and object-oriented programming (OOP) paradigms, evaluating the cost of aspect weaving. For this purpose, we have assessed three different scenarios of distributed systems security, where both static and dynamic weaving is appropriate. These examples consider access control, data flow, and data encryption. The solutions based on AOSD were developed using the *Dynamic and Static Aspect Weaving* (DSAW) (Vinuesa et al. 2008) platform. DSAW is an AOSD platform that supports both static and dynamic weaving, allowing the modification of application concerns at runtime. By using this platform, it is possible to dynamically modify the flow, access and encryption of data dynamically. Therefore, the security measures of the distributed systems can be adapted when required, varying in size and arrangement. Following a statistically performance evaluation methodology (Georges et al. 2007), these implementations are quantitatively assessed. A comparison between both paradigms is presented to estimate the penalty introduced by the AOSD approach, compared to the OOP one.

The remainder of this paper is structured as follows. Section 2 describes static and dynamic weaving,

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation: projects TIN2008-00276 *Improving Performance and Robustness of Dynamic Languages to develop Efficient, Scalable and Reliable Software*, and TIN2011-25978 *Obtaining Adaptable, Robust and Efficient Software by including Structural Reflection to Statically Typed Programming Languages*.

and the DSAW platform. The implemented applications used in the assessment are presented in Section 3. In Section 4, we evaluate and discuss the results of both approaches. Section 6 presents the conclusions and future work.

2 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) (Irwin et al. 1997) is a concrete approach to implement the principle of *Separation of Concerns (SoC)* (Hürsch & Lopes 1995). AOSD facilitates the modularisation of different functionalities that cut across the entire system software (i.e., crosscutting concerns).

An *aspect* is a piece of code that cannot be encapsulated in a method or procedure, being scattered throughout the source code of an application. Common examples of aspects include transaction control, memory management, threading, persistence or logging (Hürsch & Lopes 1995).

With the classic object-oriented paradigm, crosscutting concerns in a system cannot be modularised as regular classes. AOSD handles this problem, allowing the separation of those concerns whose code is commonly tangled with the code of other classes. The major benefits of this approach are higher level of abstraction, concern reuse, higher legibility and improved software maintainability (Hürsch & Lopes 1995).

The final application is built by weaving the application aspects with the corresponding classes (Figure 1). The output code mixes the aspect code with the application functionality modularised in traditional classes. The aspect weaver performs this code processing, offering a higher level of modularisation to the programmer. As shown in Figure 1, the major difference between AOSD and OOP is that the object-oriented programmer has to decide where to place the code of the crosscutting concerns, whereas the aspect weaver automates this process. In this paper, we evaluate the cost of this automation in three different scenarios.

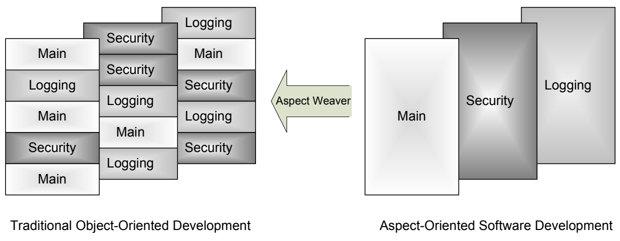


Figure 1: Traditional object-oriented development vs. aspect-oriented development.

2.1 Aspect Weaving

Once both the application components and the aspects are developed, it is necessary to build the final program (i.e., aspect weaving). The compiler of an object-oriented language receives the application source code and generates the executable file. In AOSD, the code (either source or binary) must also be processed by the aspect weaver to obtain the final program with full functionality. This process can be performed statically (at compile time or load time) or dynamically (at runtime).

Aspects should define the way they are related to application components, so that the aspect weaver can generate the final application by mixing the code of both kinds of modules. There are specific elements of the programming language semantics where the aspect code may be injected. These semantics elements are stable points of execution called *join-points* (Irwin et al. 1997). Therefore, it is necessary to describe the mapping between join-points and aspect code. For this purpose, *pointcuts* are defined as a set of join-points (usually using regular expressions) plus, optionally, some of the values in the execution context of those join-points (Kiczales et al. 2001).

2.1.1 Static Weaving

The majority of existing AOSD implementations provide static weavers. Static weavers combine the aspect and component functionality prior to application execution. This combination consists in inserting calls to *advice* in the components code. An advice is a method-like construct used to define the additional behaviour to be injected in the join-points expressed by a pointcut (Kiczales et al. 2001). An advice is the part of an aspect that modularises the code of the crosscutting concern.

This type of weaving commonly causes little performance penalty because all the code is combined and statically optimized before its execution. Since the application is woven before its execution, when a new aspect is required at runtime the application should be stopped, recompiled, rewoven and restarted, losing the non-persistent state of the process. There are scenarios where running applications require the dynamic addition, deletion or modification of aspects, and hence a dynamic weaving approach is more suitable (Ortin & Cueva 2004).

2.1.2 Dynamic Weaving

There are applications that need to be adapted at runtime in response to changes in their execution environment (Popovici et al. 2002, Zinky et al. 1997, Ségura-Devillechaise et al. 2003). An example is the so-called *autonomic software*; these systems should be able to repair, manage, optimise or recover themselves (Kephart & Chess 2003).

In the case of dynamic weaving, the program is compiled in the traditional way and an executable file is obtained. This program does not need to foresee which modules may be adapted at runtime. When the running program needs to be modified, it can be dynamically woven with new aspects that adapt the behaviour of the application.

The main advantage of this kind of weaving is that it supports the dynamic adaptation of programs, plus the modularisation of the different application concerns. Therefore, the resulting code is more adaptable and reusable, and both the aspects and the basic functionality can evolve independently (Pinto et al. 2001). However, this dynamic adaptation commonly entails a runtime performance cost (Böllert 1999).

2.2 Dynamic and Static Aspect Weaver

Existing dynamic weaving tools such as AOP/ST, PROSE, DAOP, JAC, CLAW, LOOM.NET, JAsCo or DSAW (Vinuesa et al. 2008) can be used to adapt running applications to new requirements, not foreseen at design time. Since we want to evaluate the cost of both static and dynamic weaving, a platform that supports both approaches may facilitate our work. That was the main reason why we selected DSAW (Vinuesa & Ortin 2004, Ortin et al. 2011), an

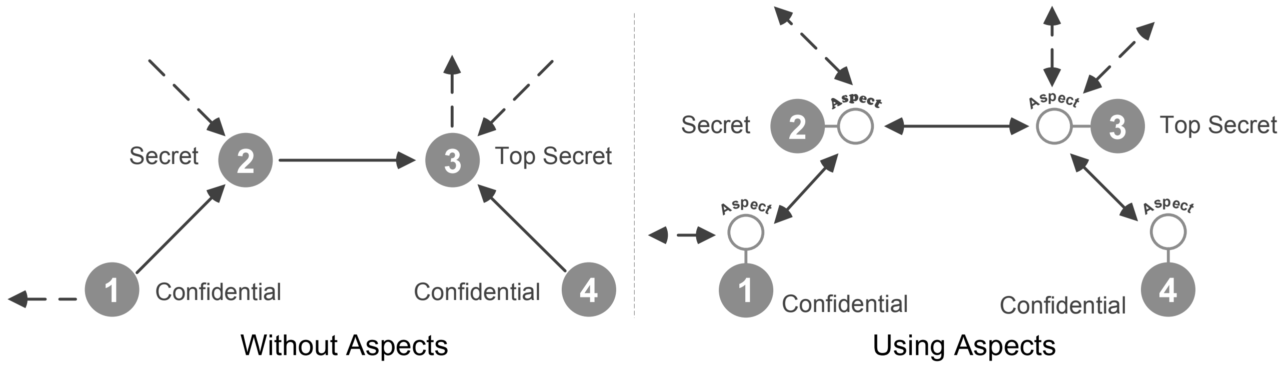


Figure 2: Distributed system with different authorisation levels.

aspect-oriented software development platform that supports homogeneous static and dynamic weaving. Its main features are:

1. Full dynamic weaving: DSAW instruments applications enabling all the application *join-points*. Since the adaptive JIT compiler of the CLR optimizes the code introduced by DSAW, the penalization is not significant. Then, DSAW allows runtime (un)weaving of aspects, even at join-points that were not woven before the application was executed.

2. Platform independence: It is designed over the .NET virtual machine reference standard (ECMA 2005). As a result, any .NET application can be run over DSAW.

3. Language independence: DSAW performs the adaptation applications at the virtual machine level, instrumenting the Intermediate Language (IL) of executable files and libraries. Any .NET high-level programming language can be used to program application components and aspects.

4. Weave-time independence: Aspect and component implementations do not depend on the type of weaving to be performed. Therefore, changing from dynamic to static, and vice versa, is a straightforward task.

5. Wide range of join-points: DSAW offers a wide and flexible set of join-points to facilitate the adaptation of applications for both dynamic and static scenarios.

3 Use cases

The objective of this paper is to compare runtime performance and memory consumption of the OOP and AOSD approaches. In order to do this, we have used both the static and dynamic weavers of DSAW, developing security issues of distributed systems (Garcia et al. 2012). DSAW has been used to implement two specific scenarios: access control / data flow and encryption of transmissions. A third scenario taking an existing real application, a FTP client and server, has also been used in our experiments. Details of these implementations are presented in (Garcia et al. 2012). These three scenarios were developed using both AOSD and the traditional OOP paradigm.

In the first scenario, we tackle the vulnerabilities caused by the flow of data through a network. Each node in the network has an authorization level. The security policy of the distributed system dictates that a node with an authorization level can only send and receive information from those nodes with greater or equal authorization level (NCSC 1990). The left part of Figure 2 shows an example. Nodes 1 and 4 can send information to any other node because the *confidential* level is the lowest one. Node 2 can only send information to node 3, since the *secret* authorization level is lower than *top secret*. Finally, node 3 cannot send information to anyone because it has the highest authorization level.

The traditional implementation only considers one-to-one relationships (Lang & Schreiner 2002), implying restrictions on data flow in point-to-point networks with changing topologies. For example, nodes 1 and 4 in Figure 2 have the same access level, but they cannot exchange information because node 3 cannot relay messages to nodes 2 and 4.

We have used the DSAW static weaver to implement a distributed system with this security policy, which guarantees the secure transmission of information over changing topologies, tagging data with the authorization levels of nodes. Applications are built relying on the classical *send* and *receive* operations, and aspects intercept these two messages to include the following functionalities:

1. Encryption of information to avoid unauthorized access to it.

2. Authentication to grant the user the appropriate authorization level.

3. Data tagging to determine how information flows across the network and to control the access to it.

As shown in the right part of Figure 2, all nodes can now exchange information between them regardless of their authorization level, because aspects control the data flow and restrict the access to data. As a result, nodes 1 and 4 can securely exchange data through nodes 2 and 3.

The second scenario is based on distributed systems made up by mobile devices, where network topologies and communication channels may dynamically change. If the user is connected to a distributed system and it is detected that the communication channel is not secure any more, encryption of transmissions may be required. Therefore, a dynamic encrypting aspect is woven with the application that uses the distributed system, while the system is running. The aspect is even able to forward the channel to another secure one if the mobile device allows it. Any kind of encryption or forwarding aspect can be woven at runtime, because the DSAW dynamic weaver does not impose any coupling between aspects and components. Finally, if the mobile device returns to a trusted environment, the encryption aspect is unwoven to avoid the unnecessary overhead of encryption.

The last scenario is a common client-server FTP environment[1]. We have added dynamic aspect weaving to the existing client and server applications in order to cipher all messages exchanged between them, when a more secure communication is needed. It is feasible to cipher the channel when critical information is exchanged, e.g. during the client login process, and to use the default channel when the exchanged information is not so important. In a standard client-server FTP communication, the information is sent and received directly. On the other hand, in an enhanced scenario where cipher is enabled using aspects, all the information passes through the dynamically woven aspect, responsible for encryption and decryption. Before either the server or the client sends a FTP command, the aspect encrypts the message; and just after a FTP command is received, the same aspect decrypts it. Thus, the exchanged information travels ciphered using the same channel, transparently to both the client and the server. If the aspect is then unwoven, the information flows as it does in the original scenario.

4 Evaluation

In this section, we present an assessment that compares runtime performance and memory consumption of the DSAW platform and the traditional OOP. The first subsection outlines the experimental methodology employed, describing the hardware, programming language and the scenarios used. For each use case described in the previous section, we present data of the runtime performance and memory consumption when using the AOSD and OOP paradigms. Finally, we present a discussion of the measurements obtained.

4.1 Methodology

We have implemented in DSAW the three use cases described in Section 3, using the aspect-oriented paradigm to modularise the crosscutting concerns in the applications. The first scenario (access control and data flow) is composed of three single nodes, and the second one (encryption) uses two single nodes and two encryption/decryption nodes –implementation details are presented in (Garcia et al. 2012). In order to compare this approach with OOP, we have also implemented the crosscutting concerns tangling their code with the rest of modules in the application, following the conventional approach of object-orientation. The two different versions of the three use cases vary in the way crosscutting concerns are tangled: by a human or by an aspect weaver. The objective is to measure the memory consumption and runtime performance cost of DSAW static and dynamic weaving. All the applications were developed in the C# programming language.

Regarding data analysis, we have followed the methodology proposed by Georges (Georges et al. 2007) to evaluate the runtime performance of applications, including those executed on virtual machines that provide JIT compilation. In this methodology, the start-up performance measures how quickly a system can run a relatively short-running application. To measure start-up performance, a two step methodology is used:

1. We measure the elapsed execution time of running multiple times the same program. This results in p (we have taken $p = 30$) measurements x_i with $1 \leq i \leq p$.

2. The confidence interval for a given confidence level (95%) is computed to eliminate measurement errors that may introduce a bias in the evaluation. The confidence interval is computed using the *Student's* t-distribution because we took $p = 30$ (Lilja 2000). Therefore, we compute the confidence interval $[c_1, c_2]$ as:

$$c_1 = \overline{x} - t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}} \qquad c_2 = \overline{x} + t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}}$$

   Being $\overline{x}$ the arithmetic mean of the x_i measurements, $\alpha = 0.05$ (95%), s the standard deviation of the x_i measurements, and $t_{1-\alpha/2;p-1}$ defined such that a random variable T, that follows the *Student's* t-distribution with $p-1$ degrees of freedom, obeys $Pr[T \leq t_{1-\alpha/2;p-1}] = 1 - \alpha/2$.

The data provided is the mean of the 95% confidence interval.

To measure runtime performance, we have instrumented the code with hooks that registers the value of high-precision time counters provided by the Windows 7 operating system. This instrumentation calls the native function `QueryPerformanceCounter` of the `kernel32.dll` library. This function returns the execution time measured by the operating system Performance and Reliability Monitor (MicrosoftTechnet 2012). We measured the difference between the beginning and the end of exchanging a set of messages to obtain the total execution time. Tests were made with different message sizes.

For memory consumption, we measured the maximum size of working set memory used by the process (the `PeakWorkingSet` property). The working set of a process is the number of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an application to use without triggering a page fault. The working set includes both shared and private data. The shared data comprises the pages that contain all the instructions that the process executes, including instructions from the process modules and the system libraries.

These implementations have been compared using the .NET Framework 2.0 build 50727 for 32 bits, over a Windows 7 x64 operating system. All tests have been carried out on a lightly loaded 2.13GHz Intel Core 2 Duo system with 4GB of RAM.

4.2 Evaluation

To evaluate the cost of static and dynamic weaving in DSAW, we have developed the three proposed scenarios using the traditional object-oriented programming paradigm. Using object-orientation, we have extended the implementations with access control and data flow security measures in the first use case, and encryption in the other two scenarios. These same functionalities were also developed as separate aspects, using the AOSD paradigm.

In the control access and data flow scenario, we have used static weaving to inject the aspect in the original system. In the encryption and FTP use cases, the DSAW dynamic weaver was employed. Following the start-up methodology presented in Section 4.1, we measured the influence of the number of messages on the performance and memory penalties. Since both penalties remained constant, we used a fixed number

[1]We have used the FTP.Net client (`http://ftpnet.sourceforge.net`) and the SimpleFTP Server (`http://www.tudra.net/wp/2007/10/15/simpleftp-server`).

of 6,000 messages for the first and third applications, and 100,000 messages for the second one.

Figures 3 to 5 show the runtime performance penalty in the three scenarios. For each application, we increase the number of words contained in the messages in order to analyse the influence of message sizes in runtime performance. Performance penalties are calculated relative to the corresponding OOP implementation. Values are the difference between the DSAW and OOP execution times, divided by the value of the OOP implementation (expressed in percentage form) for each message size (expressed in number of words).

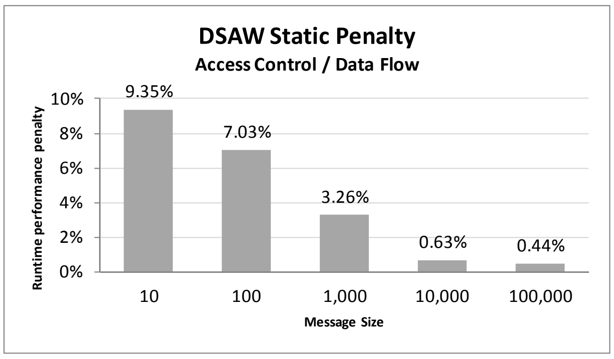


Figure 3: Runtime performance penalty of the Access Control / Data Flow application.

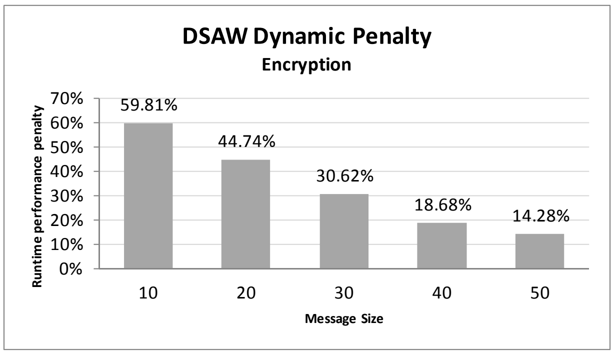


Figure 4: Runtime performance penalty of the Encryption application.

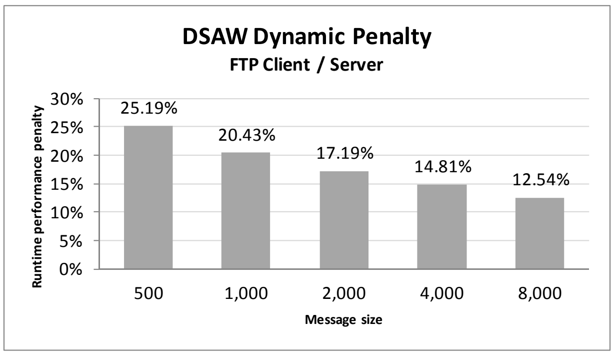


Figure 5: Runtime performance penalty of the FTP application.

Concerning the memory consumption, Tables 1 to 3 show the memory usage in the three scenarios: access control / data flow (OOP vs. DSAW static), encryption (OOP vs. DSAW dynamic) and FTP client-server (OOP vs. DSAW dynamic). Memory consumption is expressed in KBytes, and message size in number of words per message.

4.3 Discussion

After presenting the data in Figures 3 to 5 and Tables 1 to 3, three issues are highlighted. The first one is related to the runtime performance cost of weaving. In all scenarios, the performance costs decrease as the size of messages increases. In the first application, the performance cost of static weaving varies from 9.35%, with the minimum message size, to 0.44%, when messages are 10,000 times greater. In the second scenario, the cost decreases from 59.81% (10 words per message) to 14.28% (50 words per message). Finally, for the FTP application shows a performance penalty of 25.19% for the smallest message, while this penalty drops to 12.54% when the message size is multiplied by 16. Therefore, runtime performance penalty grows as the size of the message drops. This dependency is caused by the number of intercepted joinpoints executed, which remains constant in each experiment. For bigger messages, the overall execution time rises, but the injection code executed (the number of joinpoits) stays the same. The FTP application shows a smaller dependency on the message size. This is because the application executes 10 commands (such as creating, changing and erasing directories), and only one is file (message) transmission.

The second discussion is the different performance penalties depending on the type of weaving. In the static scenario, runtime performance penalty is between 9.35% and 0.44%. When the size of messages is significantly high, the cost of static weaving is almost negligible. However, the cost of dynamic weaving is more notable. The encryption application showed a performance penalty between 59.81% and 14.28%, and 25.19% and 12.54% in the case of FTP. This higher performance cost of dynamic weaving is caused by different factors. First, the execution of the dynamic weaver is included in the overall execution time. Second, the runtime examination of joinpoint registration (checking whether there are aspects waiting for a joinpoit to be executed) also implies a performance price. Finally, when a joinpoint is reached, registered aspects are called by means of an indirection (a reference); whereas static weaving simply tangles the code, enabling the optimizations performed by the JIT compiler (Redondo et al. 2008, Ortin et al. 2009).

The last issue is related to memory consumption. In every scenario, the memory consumption penalty is not affected by the size of messages: standard deviations of the three applications where 0.14%, 0.34% and 0.26%, respectively. The static weaving technique has shown an average memory consumption increase of 2%. This average cost augments to 60% and 47% when dynamic weaving is used in the two last scenarios. This difference is due to the additional code and the registered aspects per joinpoint table implemented by the dynamic weaver to allow dynamic adaptation of components. Therefore, the cost of dynamic weaving examples has been higher than static ones, for both runtime performance and memory consumption.

5 Related Work

There are some existing works that compare the runtime performance of different AOSD platforms (Vinuesa et al. 2008, Bijker 2005, Vanderperren & Suvée

Message size	10	100	1,000	10,000	100,000
Object-Oriented	28,001	28,112	29,532	38,272	51,788
DSAW Static	28,735	28,859	30,387	39,400	53,204

Table 1: Memory consumption (KBs) of the Access Control / Data Flow application.

Message size	10	20	30	40	50
Object-Oriented	27,408	27,440	27,524	27,517	27,650
DSAW Dynamic	43,880	43,880	44,187	43,980	44,148

Table 2: Memory consumption (KBs) of the Encryption application.

Message size	500	1,000	2,000	4,000	8,000
Object-Oriented	33,708	33,799	33,799	33,832	33,764
DSAW Dynamic	49,484	49,600	49,768	49,772	49,716

Table 3: Memory consumption (KBs) of the FTP application.

2004), but there are not many that compare AOSD implementations with equivalent OOP versions. Hilsdale and Hugunin (Hilsdale & Hugunin 2004) assess both run-time and compile-time performance of AspectJ, introducing a simple logging policy as an aspect in a benchmark. In this experiment, the aspect is woven statically. It logs all the method entries of the XSLTMark benchmark. The obtained results are fairly similar to the results presented in this work. The runtime performance cost of the AOSD version compared to the hand-coded one is around 3%. Moreover, the authors present a comparison between the original application (without any modification) and the AOSD version (with logging) to evaluate the overhead introduced by the logging aspect. Using different versions of the aspect code, this overhead is greatly reduced from 2,500% to 22%.

Regarding the use of AOSD to implement and adapt security measures, Viega (Viega et al. 2001) proposes an extension of the C programming language to support aspects. This extension allows the definition of security policies apart from the application code. AspectJ has also been used for security issues. Huang (Huang et al. 2004) presents a generic and reusable library to introduce security mechanisms in Java developments. This library provides reusable and generic aspects in AspectJ, as practical software components, and a prototype implementation of a common security-relative API for AOSD. Kim and Lee (Taeho & Hongchul 2008) also use AspectJ to discuss how authorisation capabilities could be added to existing well-structured, object-oriented systems. In these works, neither run-time performance nor memory consumption is assessed.

6 Conclusions

Aspect-Oriented Software Development allows the separation of crosscutting concerns in software development. The modularisation of different application concerns provides higher level of abstraction, concern reuse, higher legibility and improved software maintainability. However, in some scenarios, the use of AOSD may involve an increase of runtime performance and memory consumption. This paper presents a comparison of runtime performance and memory consumption between three different applications developed using AOSD and the classical object-oriented approach. All the applications were developed in the DSAW platform and using the C# programming language. The three applications apply security measures to distributed systems: access control and data flow, communications encryption, and FTP client-server. The first application is statically woven, whereas the two last ones require dynamic weaving.

The assessment of runtime performance has shown that the DSAW static weaver have entailed a performance cost of 9.35%, compared to the traditional object-oriented development. When the size of the messages increases, the performance cost decreases to values near to zero. In the dynamic weaving scenarios, runtime performance penalties were 59.81% and 25.19%, dropping to values around 13% when message sizes grow. Different factors in the dynamic weaving technique implemented by DSAW causes this performance increase compared to static weaving. In all the scenarios, memory consumption has not depended on the size of messages. The memory usage penalty of static weaving was around 2%, and 60% and 47% in the case of dynamic weaving.

We plan to apply this evaluation methodology to commercial aspect-oriented applications developed in other platforms such as AspectJ, Spring AOP, JAsCo or JBoss AOP. Regarding the use of AOSD in distributed systems security, future work will be focused on applying the AOSD approach to develop other security measures such as *Intrusion Detection* or *Load Balancers*. A more involved question concerns how our approach can be suitably generalised in distributed “systems-of-systems” scenarios (Ackoff 1971). We aim to address this challenge in the future by considering more flexible means of defining pointcuts, such as by allowing pointcuts to be defined in a reactive manner, taking into account the results of analysis of multiple interacting applications within a network, and using different techniques together with our approach (Söldner et al. 2008, Tanter et al. 2009).

The current documentation and implementation of this work can be freely downloaded from (DSAW 2010).

References

Ackoff, R. (1971), ‘Towards a system of systems concepts’, *Management Science* **17**(11), 661–671.

Belapurkar, A., Chakrabarti, A., Ponnapalli, H., Varadarajan, N., Padmanabhuni, S. & Sundarrajan, S. (2009), *Distributed Systems Security: Issues, Processes and Solutions*, Wiley.

Bijker, R. (2005), Performance effects of aspect oriented programming, *in* '3rd Twente Student Conference on IT, Enschede June'.

Böllert, K. (1999), On weaving aspects, *in* 'Workshop on Object-Oriented Technology', Springer-Verlag, p. 302.

DSAW (2010), 'Using DSAW in distributed system security systems', `http://www.reflection.uniovi.es/dsaw/download/2010/ietsw/`.

ECMA (2005), 'TG3. Common Language Infrastructure (CLI). Standard ECMA-335'.

Garcia, M., Llewellyn-Jones, D., Ortin, F. & Merabti, M. (2012), 'Applying dynamic separation of aspects to distributed systems security: a case study', *Software, IET* **6**(3), 231–248.

Georges, A., Buytaert, D. & Eeckhout, L. (2007), 'Statistically rigorous Java performance evaluation', *ACM SIGPLAN Notices* **42**(10), 57–76.

Hilsdale, E. & Hugunin, J. (2004), Advice weaving in AspectJ, *in* 'International Conference on Aspect-Oriented Software Development (AOSD)', ACM, pp. 26–35.

Huang, M., Wang, C. & Zhang, L. (2004), Toward a reusable and generic security aspect library, *in* 'AOSD Technology for Application-Level Security (AOSDSEC)', Vol. 4, pp. 5–10.

Hürsch, W. & Lopes, C. (1995), 'Separation of concerns', *Northeastern University, February* .

Irwin, J., Kickzales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. & Loingtier, J. (1997), Aspect-oriented programming, *in* 'European Conference on Object-Oriented Programming (ECOOP)', pp. 220–242.

Kephart, J. & Chess, D. (2003), 'The vision of autonomic computing', *Computer* **36**(1), 41–50.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. (2001), An overview of AspectJ, *in* 'European Conference on Object-Oriented Programming (ECOOP)', Springer, pp. 327–354.

Lang, U. & Schreiner, R. (2002), *Developing secure distributed systems with CORBA*, Artech House Publishers.

Lilja, D. (2000), *Measuring computer performance: a practitioner's guide*, Cambridge University Press.

MicrosoftTechnet (2012), 'Windows server techcenter: Windows performance monitor'.

NCSC, N. C. S. C. (1990), 'Trusted network interpretation environments guideline'.

Ortin, F. & Cueva, J. M. (2004), 'Dynamic adaptation of application aspects', *Journal of Systems and Software* **71**(3), 229–243.

Ortin, F., Lopez, B. & Perez-Schofield, J. (2004), 'Separating adaptable persistence attributes through computational reflection', *Software, IEEE* **21**(6), 41–49.

Ortin, F., Redondo, J. M. & Baltasar García Perez-Schofield, J. (2009), 'Efficient virtual machine support of runtime structural reflection', *Science of Computer Programming* **74**(10), 836–860.

Ortin, F., Vinuesa, L. & Felix, J. (2011), 'The DSAW Aspect-Oriented Software Development Platform', *International Journal of Software Engineering and Knowledge Engineering* **21**(7), 891.

Parnas, D. (1972), 'On the criteria to be used in decomposing systems into modules', *Communications of the ACM* **15**(12), 1053–1058.

Pinto, M., Amor, M., Fuentes, L. & Troya, J. M. (2001), Run-time coordination of components: Design patterns vs. component-aspect based platforms, *in* L. Bergmans, M. Glandrup, J. Brichau & S. Clarke, eds, 'Workshop on Advanced Separation of Concerns (ECOOP)'.

Popovici, A., Gross, T. & Alonso, G. (2002), Dynamic weaving for aspect-oriented programming, *in* 'International Conference on Aspect-Oriented Software Development', ACM Press, pp. 141–147.

Redondo, J. M., Ortin, F. & Cueva, J. M. (2008), 'Optimizing reflective primitives of dynamic languages', *International Journal of Software Engineering and Knowledge Engineering* **18**(6), 759–783.

Ségura-Devillechaise, M., Menaud, J., Muller, G. & Lawall, J. (2003), Web cache prefetching as an aspect: towards a dynamic-weaving based solution, *in* 'International Conference on Aspect-Oriented Doftware Development (AOSD)', ACM, p. 119.

Söldner, G., Schober, S., Schröder-Preikschat, W. & Kapitza, R. (2008), 'AOCI: Weaving components in a distributed environment', *On the Move to Meaningful Internet Systems: OTM 2008* pp. 535–552.

Taeho, K. & Hongchul, L. (2008), 'Establishment of a security system using aspect oriented programming', *2008 International Conference on Control, Automation and Systems* pp. 863–866.

Tanter, É., Fabry, J., Douence, R., Noyé, J. & Südholt, M. (2009), Expressive scoping of distributed aspects, *in* 'ACM international Conference on Aspect-Oriented Software Development (AOSD)', ACM, pp. 27–38.

Vanderperren, W. & Suvée, D. (2004), Optimizing JAsCo dynamic AOP through Hotswap and Jutta, *in* 'AOSD Workshop on Dynamic Aspects', Vol. 3.

Viega, J., Bloch, J. & Chandra, P. (2001), 'Applying aspect-oriented programming to security', *Cutter IT Journal* **14**(2), 31–39.

Vinuesa, L. & Ortin, F. (2004), 'A Dynamic Aspect Weaver over the. NET Platform', *Metainformatics* **3002**, 197–212.

Vinuesa, L., Ortin, F., Felix, J. M. & Alvarez, F. (2008), DSAW - a dynamic and static aspect weaving platform., *in* 'International Conference on Software and Data Technologies (ICSOFT)', INSTICC Press, pp. 55–62.

Zinky, J., Bakken, D. & Schantz, R. (1997), 'Architectural support for quality of service for CORBA objects', *Theory and Practice of Object Systems* **3**(1), 55–73.

The Design and Implementation of Clocked Variables in X10

Daniel Atkins[1] **Alex Potanin**[1] **Lindsay Groves**[1]

[1] School of Engineering and Computer Science,
Victoria University of Wellington,
Wellington, New Zealand
Email: {Daniel.Akins,alex,lindsay}@ecs.vuw.ac.nz

Abstract

This paper investigates the addition of Clocked Variables to the X10 Programming Language. Clocked Variables work well for primitives and objects with only primitive fields, but incur substantial performance penalties for more complex objects. We discuss ways to deal with these issues.

Keywords: X10, Concurrency, Clocks

1 Introduction

Distribution and parallelization are an important part of computing today; with the focus of processor manufacturers turning away from higher speeds, and towards larger numbers of cores, proper utilization of such resources is becoming more and more important (Saraswat et al. 2007, Murthy 2008). Unfortunately, many current programming languages don't provide the necessary support for easily writing thread-safe programs. To address this issue, IBM have been developing a new programming language called X10 (Saraswat et al. 2007, 2011, Charles et al. 2005). X10 is a strongly typed, concurrent, imperative, and object-oriented programming language, making it quite similar to popular existing languages such as Java and C++. X10 was designed with multi-core and clustered systems in mind. The goal of X10 is to allow programmers to easily produce code that can be distributed over multiple cores and/or machines, with good scalability (Murthy 2008). This means that concurrent programs become much easier to write, as the language has many built-in constructs to aid programmers in achieving their goals (Ebcioglu et al. 2005).

Many concurrent algorithms maintain two states; a current state and a next state. Operations are performed on the current state, and the results cause the next state to be updated. When the current state has been fully processed, the next state becomes the current state, and the algorithm continues. This can lead to code bloat, as maintaining these two states requires some extra book-keeping, and there is a risk of introducing bugs into a program by accidentally using the wrong state when performing an operation.

Clocked Variables allow variables to have different states depending on how they are used. This allows the prevention of race-conditions as each thread is guaranteed a consistent view of the clocked variables. This is achieved by requiring that updates to a clocked variable do not become visible until every thread has indicated that they are ready to progress to the next state. In the case of clocked Primitives, this reduces the requirement of having two explicit states to simply maintaining a single object in memory that automatically performs updates and state transitions at the appropriate times. More interesting is the case of Clocked References, which require a more complex clocking mechanism—it is these that we investigate in this paper. We implement Clocked Variables in X10, guided by the X10 Design Document (Saraswat 2011). While this paper is written with X10 in mind, the concepts it presents are applicable to any programming model that uses phased execution controlled by barriers.

The rest of this paper is structured as follows: Section 2 introduces the X10 Language, and details some of the language-specific constructs that are required to understand this paper. Section 3 discusses Clocked Variables, with a focus on how Clocked References are handled. Section 4 describes the experimental setup used to benchmark the performance of clocked variables, and Section 5 evaluates the performance of Clocked Variables. Section 6 gives further discussion of the results, and the paper is then is concluded by Section 7.

The main contributions of this paper are:

- Extension of the X10 programming language to include Clocked Variables,
- Case studies that use the new Clocked Variables,
- Performance evaluation of Clocked Variables in X10.

2 The X10 Programming Language

X10 contains several language constructs that allow programmers to readily, and easily, write concurrent code (Ebcioglu et al. n.d.). **Places**, which can be thought of as analogous to processes, provide a shared memory environment in which concurrent code can be executed. This memory is not shared *between* Places, which allows Places to be parallelised, and distributed across multiple machines. *Within* Places, concurrent execution is achieved by the use of **Activities**, which are analogous to Threads.

A Clock is an object that provides a programmer with a means of synchronizing concurrently executing threads—an important idea in a distributed system (Lamport 1978). In X10, this synchronization is achieved through the use of a barrier-style structure based on Lamport's Logical Clock (Lamport 1978); the clock object maintains a total count of the number of Activities (threads) that have registered with the clock, and a separate count of the number of activities that are currently active—i.e, not currently waiting for the clock to advance to the next phase. When an activity wishes to advance to the next phase, the clock first decrements the count of alive activities, and if this is zero, atomically advances the phase of the clock. The calling activity is blocked by placing it in a loop until the clock advances (busy-waiting).

Clocks in X10 maintain an invariant `GlobalRef` field that refers explicitly to the original instance of the `Clock`, so that no matter where any copies may end up, they can always refer to the same `Clock` object. By forcing all updates to the internal fields of the clock to always execute at the root `Clock` object, the same state is seen by all copies of the clock at all times—an important part of ensuring proper synchronization!

X10 is built primarily as an extension to Java, using Polyglot (Nystrom et al. 2003) to handle the translation from X10 source code to Java source code. X10 can also be compiled to C++ source code. The X10 Runtime is written primarily in X10 itself, giving it the ability to be compiled into one of several back-ends. Currently, there is a Java-based runtime environment (using the X10 Runtime as libraries for the JVM), a C++ based runtime environment, and a CUDA (Compute Unified Device Architecture—a parallel computing architecture developed by Nvidia, that can be executed on GPUs) runtime environment.

3 Clocked Variables

The clocked variables described in this paper are based on the design outlined in the X10 Design Document (Saraswat 2011). There, the intent is for only val (final variables that can be altered once per clock phase) and stack local variables to be able to be clocked, as dealing with object references was considered too hard. The intent of this paper is to explore that claim and to investigate if it is possible to have *any* form of variable able to be clocked. Extensions are proposed in the Design Document to allow methods, objects, fields and types to be clocked as well; but we do not consider these in this paper. We deal only with the idea of clocked primitives and references, and how interactions with them might proceed.

A clocked variable is functionally similar to a normal, unclocked variable—a location in memory in which a primitive value, or a reference to an object, is stored and can be accessed. However, in a clocked environment, a clocked variable becomes quite different to an unclocked variable, in terms of how and when it can be updated and accessed.

3.1 Design of Clocked Variables

During a single clock phase, the value of a clocked variable remains **fixed**. If the variable is written to, or updated in some way, the change does not become visible until the **end** of the clock phase. Figure 1 gives an example of code that demonstrates this.

We require that clocked variables only be written to **once** during any given clock phase—writing to a clocked variable more than once in a given clock phase is a runtime exception. Clocked variables may be *read* any number of times during a given clock phase, but we require that this value remains constant for the duration of the phase. If a clocked variable is written to, or updated in any way, the new value must take effect *between* the clock phases. The idea, then, is that clocked variables provide the same functionality as manually maintaining two separate states in

```
clocked var i:Int = 5;
i = 6;
Console.OUT.println(i); //Prints 5
Clock.advanceAll();
Console.OUT.println(i); //Prints 6
i = 0;
Console.OUT.println(i); //Prints 6
Clock.advanceAll();
Console.OUT.println(i); //Prints 0
```

Figure 1: Example of Clocked Code

a concurrent algorithm, but without all of the extra book-keeping.

Only allowing one write per phase may appear to be an odd design decision; primarily this was done to meet the proposal for Clocked Variables given in the X10 Design Document (Saraswat 2011). However, that document specifies this behaviour for variables marked with the keyword `val`—that is, variables that are final, but when clocked, can be updated once per phase. In this case, it **is** an error to write to the variable more than once per phase, as the variables are *final*. Under clocking, the original design allows such variables to be re-initialised once per phase. We did not adhere strictly to this design, as we allow the clocking of non-val variables, and allowing more than one write per phase was considered. However, this limit was deemed necessary to deal with some of the issues raised by clocking reference types, as discussed later.

3.2 Clocked Primitive Types

The design of clocked variables started with primitives, as they are conceptually easier to deal with than objects and references. Our design for clocked primitives is based on the outline for clocked vals given in the X10 Design Document, but has been extended to cover non-local vars and fields as required. We also depart from the Design Document in that clocked primitives can still be used *outside* of a clocked environment (ie: with a block encapsulated by `clocked(Clock)`, `clocked finish`, or `clocked async`)—they simply revert to behaving like an unclocked variable of the appropriate type.

The basic design is that of wrapper classes—instead of dealing with the primitive variable directly, all interactions are abstracted away by "Clocked Primitive" objects that sit between the primitive variable and the rest of the program. One of these wrapper classes is needed for each of the thirteen primitive types available in X10.

The design of the wrapper classes is reasonably simple. Each class contains two fields of the appropriate primitive type: one to hold the *current* value of the clocked variable, and one to hold the *next* value. Only two operations are supported on clocked primitives:

read returns the current value of the clocked variable. Can be performed any number of times.

write updates the next value of the clocked variable. Can only be performed *once* per clock phase.

3.3 Clocked References

Like Clocked Primitives, Clocked References are expected to maintain a constant value during a clock phase, and then to update that value at the end of each clock phase. Unlike Clocked Primitives, this is not a simple matter of just executing `current = next`.

Clocked References are not dealt with in the design document, save for defining the concept of a "clocked field" that might exist inside such an object. Thus, the design for Clocked References is entirely our own, and is based on the design of Clocked Primitives.

As a Clocked Reference must encapsulate a reference type (not a primitive), using generics to describe a general "Clocked Reference" was deemed the best approach. The bigger issue is that an object may contain references to other objects. Clearly updating such a complex structure would not be a simple task. So, how do we successfully update a Clocked Reference?

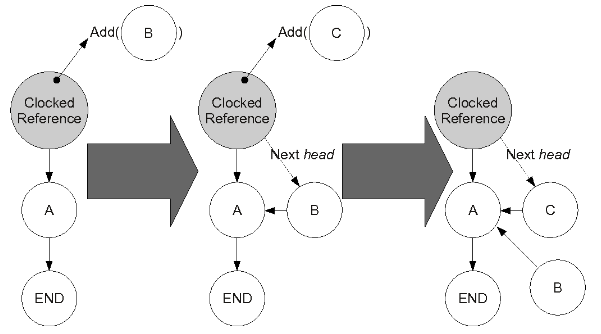


Figure 2: A clocked LinkedList behaves oddly under a call to add(node)

The answer is not simple. Figure 2 shows the behaviour of a clocked LinkedList during a call to add. Notice that we have a clocked reference to the head of the list. A call to add a node to the list, executed on the head node, adds a node to the list—but this alteration is not visible yet, as any *reads* of the graph are done from the current state, and the alteration is performed on the next state. Another add call is made; how do we resolve this? We need to ensure that we have access to a up-to-date version of the list, but the first change has not yet been commited. We cannot set the *next* field of the Node correctly, and the list enters an inconsistent state. Ideally, we would require all such alterations to be performed on a version of the graph that is kept up-to-date. It becomes clear that we cannot simply just maintain two states for the object being referenced by the clocked reference; we need to do this for the *entire* object graph it is connected to. But how, then, do we propagate changes to the current state when the clock advances?

There are many ways in which a Clocked Reference could update its value—the simplest of which is a deep-clone of the entire object graph. In fact, X10 readily provides a method to perform exactly that operation; one which even takes cycles into account. This is the method used in the design of Clocked References within this paper, as time constraints meant other avenues could not be fully explored. To avoid issues caused by calling multiple updates on the graph in one clock phase, the write operator was limited to only one write per phase, as specified in the Design Document (Saraswat 2011). With the naïve deep-cloning method of updating Clocked References, however, it could be argued that this was unnecessary, as the two states are completely separate object graphs. In the interest of exploring more interesting update mechanisms, however, we felt it was necessary to enforce this limit.

Having chosen a solution to the problem of updating a clocked reference, we turn to the operations that can be performed on a clocked reference. Immediately we can see that the operations used with clocked primitives are not going to suffice. `Read` still functions well enough, as it now just returns a reference to the current value of the clocked reference. `Write` proves a little more troublesome. We don't want to support an operation that replaces the next value wholesale—instead, we want to be able to give out a reference to the next value to allow programs to alter it in less destructive ways (such as updating a field, or calling a method, etc). After some consideration, it was decided that Clocked References would not support *any* operations, as there was no easy way to pass only the required changes to the graph as a parameter. Instead, direct access to the current and next values of the clock reference would be performed via method calls (`readableObject()` and `writableObject()` respectively).

3.4 Back-end Design

Having described the design of clocked primitives and references, we now describe the design of the actual clocking mechanism itself, and how it fits into the overall X10 architecture. This is not touched on at all in the X10 Design Document, and as such, is entirely our own design.

There are two main alternatives for the back-end of this system. The first puts the onus on the Clock to keep track of Clocked Variables and perform the updates. The second shifts this responsibility to the spawning Activity itself.

3.4.1 The GlobalRef Method

The first implementation of Clocked Variables uses the GlobalRef structure (an X10 type that can be used to access objects across Place boundaries) to ensure that all operations performed on the object are executed in the correct place—this is especially vital, otherwise the state of the object becomes inconsistent.

A list of GlobalRef objects is maintained by the Clock object. When a clocked variable is registered on that clock, its GlobalRef object is copied to the "root" of the clock (the `Place` it uses for its fields) and added to the list. Then, when the Clock advances from one phase to the next, it calls the next() method on all of the members of the list. Figure 3 illustrates this.

3.4.2 The Map Method

For this design, the onus of keeping track of clocked variables falls on the activity in which they were declared. Each clocked variable is assigned an integer id upon construction, and a mapping from this id to the clocked variable is stored in a HashMap within the activity. The activity then registers the clocked variable on the same clock (if any) that the activity is registered on. This is accomplished simply by passing the integer id to the clock, which stores it in an ArrayList in the "root" Clock. Since the id is invariant, the fact that this value crosses places during this action does not raise any concerns.

At the end of the clock advancement step, the clock passes its internal list to each activity that it is associated with. Then, each activity scans the list for any ids that exist in the Map—if it finds any, the activity issues a call to next() on that clocked variable. Figure 4 illustrates this.

This design has the advantage of storing very little state within the Clock object itself: a list of primitive integers, rather than a larger struct. Since the burden of updating the clocked variables falls on the activity itself, there is also no need to switch places in order to update them—and this way, clocked variables in different places (and thus Activities) are updated concurrently, rather than consecutively

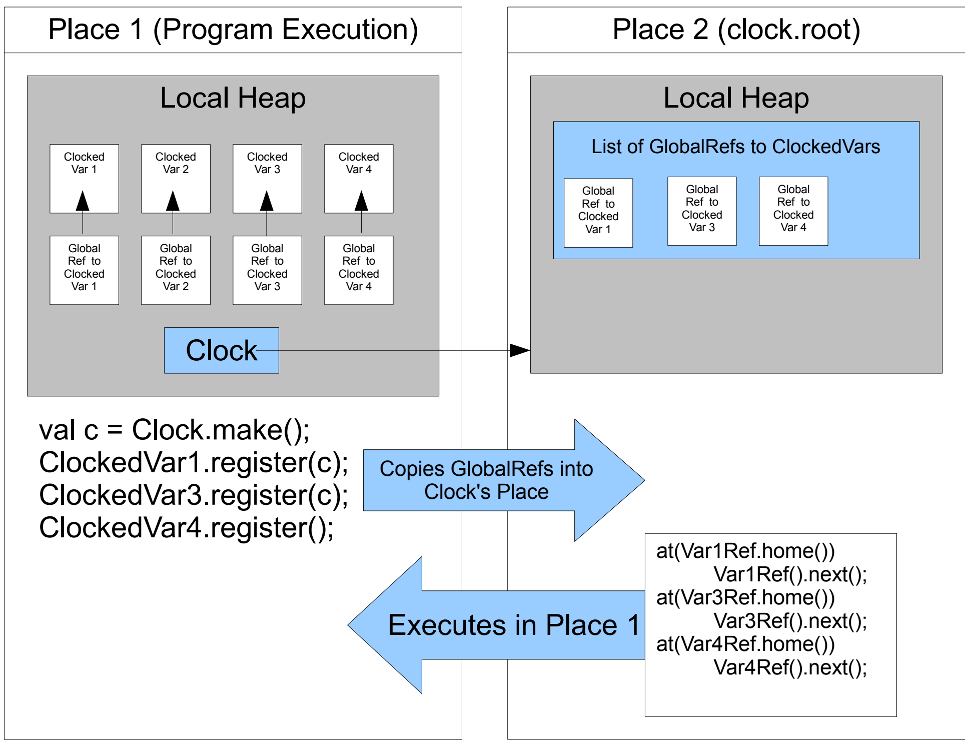

Place 1 (Program Execution)
Place 2 (clock.root)
Local Heap
Clocked Var 1
Clocked Var 2
Clocked Var 3
Clocked Var 4
Global Ref to Clocked Var 1
Global Ref to Clocked Var 2
Global Ref to Clocked Var 3
Global Ref to Clocked Var 4
Clock
Local Heap
List of GlobalRefs to ClockedVars
Global Ref to Clocked Var 1
Global Ref to Clocked Var 3
Global Ref to Clocked Var 4
val c = Clock.make();
ClockedVar1.register(c);
ClockedVar3.register(c);
ClockedVar4.register();
Copies GlobalRefs into Clock's Place
Executes in Place 1
at(Var1Ref.home()) Var1Ref().next();
at(Var3Ref.home()) Var3Ref().next();
at(Var4Ref.home()) Var4Ref().next();

Figure 3: An activity in Place 1 using 4 ClockedVars under the GlobalRef method

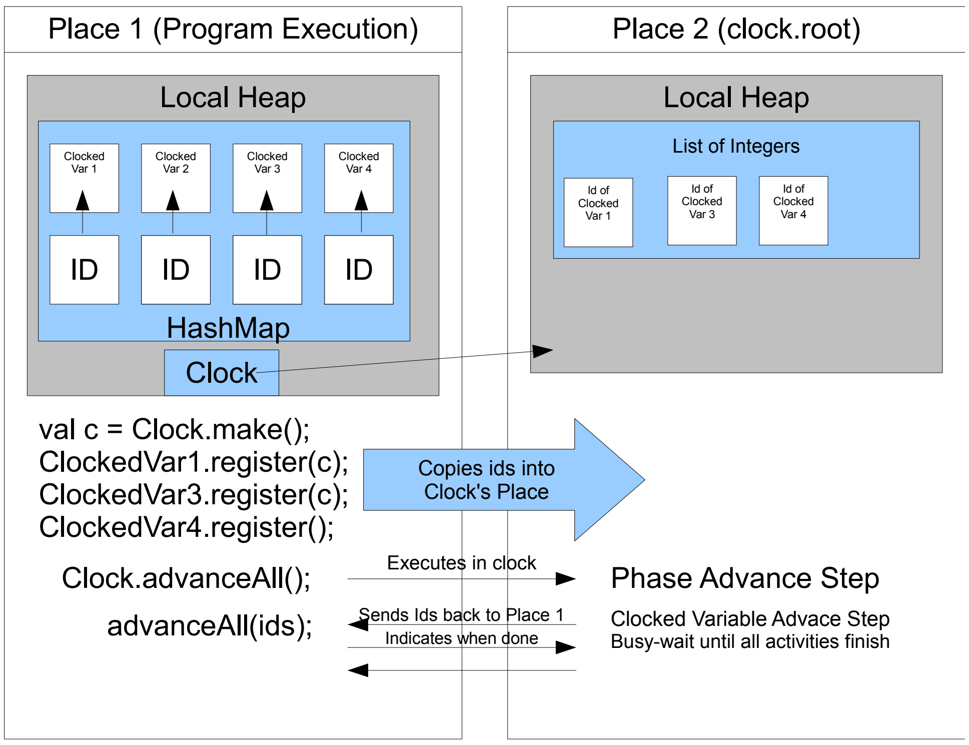


Figure 4: An activity in Place 1 using 4 ClockedVars under the Maps method

Model	Dell Optiplex GX780
CPU	Intel(R) Core(TM)2 Duo @ 3.00GHz
RAM	4GB
OS	ArchLinux (3.2.4-1-ARCH)
HDD	250GB Serial ATA 7200rpm
Ethernet	Intel On-Board 1 Gigabit

Table 1: Hardware specifications for the benchmark applications

It was decided that the Map method would be used to implement Clocked Variables, as the cost of switching places is quite high in terms of efficiency. The Map method does this much less often than the Global Ref method (once per calling activity, rather than once per clocked variable).

4 Benchmarks and Evaluation

The performance of both types of clocked variable was measured through the use of four benchmarks, each of which tested a different form of reference type. Each benchmark was implemented in two different ways; using the clocked references described in Section 3, and not using clocked references. For the "unclocked" case, synchronization and state updates were handled manually—the term refers to the absence of clocked variables, not the absence of clocks themselves. Care was taken to ensure that all versions of the benchmark programs operated correctly, and that the use of clocked/unclocked references was the *only* difference between the two versions of each benchmark. Each benchmark was executed 100 times, on a range of different values. The results shown here give the average values of those executions. Table 1 details the specifications for the hardware these benchmarks were executed on.

4.1 Conway's Game of Life

Conway's Game of Life is a fairly simple cellular automaton, originally described by the mathematician John Conway (Gardner 1970). The automaton consists of a two-dimensional grid-based world, with each cell of the grid having two states (dead or alive). Cells live or die according to fixed rules that are only reliant on the current state of the board. At each step, the rules are applied *simultaneously* to each cell in the grid. This is done in X10 by using the `async` structure to parallelise the application of the rules to each cell. Each cell is given its own thread, the state of each cell is calculated concurrently with the state of each other cell.

This was implemented in X10 using an array of integers to represent the grid. The clocked version used a single array of clocked integers, and the unclocked version used two arrays of normal integers (one to represent the current state, and one to represent the next state). The update mechanism for the unclocked version is essentially the same as for the clocked version (but coded manually): a loop copies the value from the next board state to the current board state.

Figure 5 gives the results for Conway's Game of Life for boards of various sizes. There is no significant difference between the clocked version and the unclocked version. This outcome was expected, as clocked and unclocked primitives are both updated via the same mechanism—directly copying the new value over the old value.

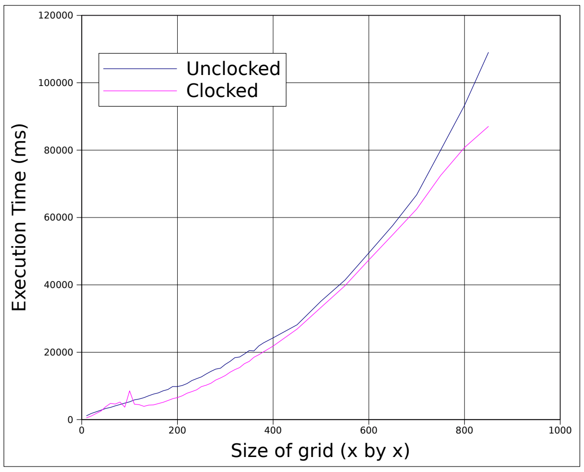


Figure 5: Conway's Game of Life: Clocked vs Unclocked execution times

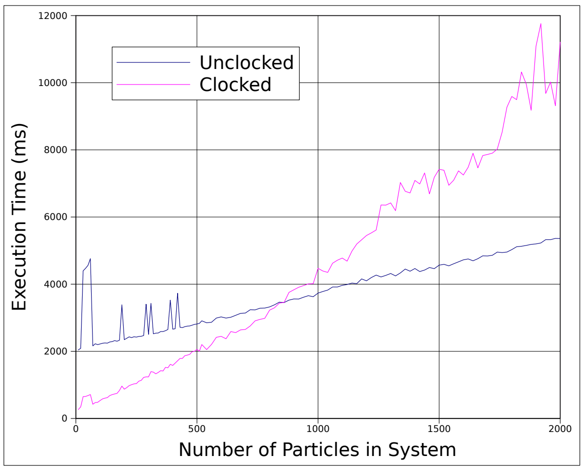


Figure 6: N-Body Simulation: Clocked vs Unclocked execution times

4.2 N-Body Simulation

An N-Body simulation is a physical simulation of a system of many interacting *particles*. N-Body problems are computationally intensive, as calculating the next state of a particle involves determining its interactions with every other particle in the system. Generally, these interactions take the form of forces exerted between the particles—usually gravitational (in the case of uncharged particles or large bodies, like planets) or electrostatic (in the case of charged particles) or both.

This benchmark was implemented as a N-Body system of uncharged particles (i.e. the only interaction between the particles was gravitational). The particles were represented as a simple object with several primitive fields and an update method. In the clocked version of this benchmark, these particles were clocked. The update method executed on the *next* state of the object, and wrote directly to the fields. In the unclocked version, two additional fields had to be added to hold the information required to update the particle, and a new method, `next()` was added to the Particle class so this update could be performed. Similar to Conway's Game of Life, this was done after ensuring that all of the next states had been calculated.

Figure 6 shows the results for the N-Body Benchmark, for various numbers of particles. As one would expect, execution time scales with the number of par-

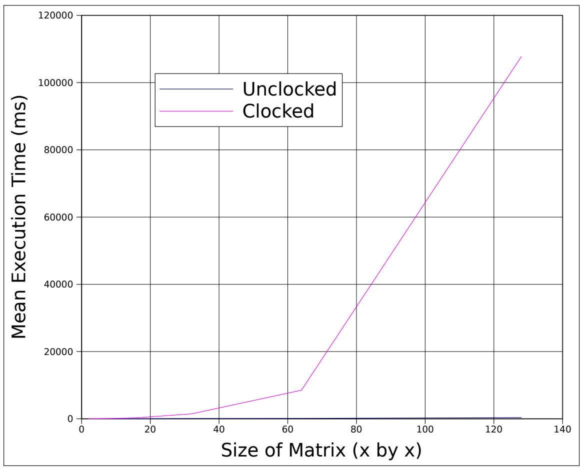


Figure 7: Sparse Matrix: Clocked vs Unclocked execution times

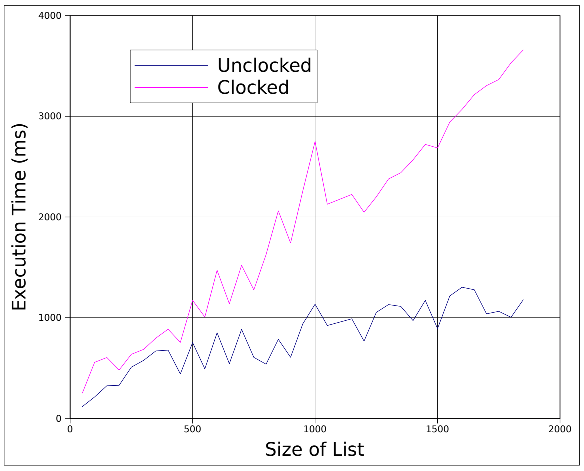


Figure 8: Linked List: Clocked vs Unclocked execution times

ticles present in the system (as this is an $O(n^2)$ algorithm). Interestingly, however, the clocked and unclocked versions clearly have a very different gradient. The update mechanism for the clocked version is a simple deep clone of the object (which only has primitive fields—essentially a struct), whereas the update mechanism for the unclocked version was method call on the object that performed two minor calculations and updated the fields. For smaller numbers of objects, the cloning method is much faster, but the time cost increases at a faster rate than the method-call update. The two methods are equal at around 800-850 objects, and the method-call update is faster for object numbers above that. From the graph, it appears that the method-call update has a constant cost associated with it (hence it starting at a much higher y value). This may be due to the update threads having to synchronize between the calculation phase and the update phase—something that doesn't need to happen in the clocked version.

4.3 Sparse Matrix Convolution

A Sparse Matrix allows more compact storage by storing only the *non-zero* values within the matrix. We used a linked-list style structure, in which each row of the matrix is represented by a single list. Rows are then linked by their first node. This allows access to any cell within the matrix by following the links from the root node.

In this benchmark, a sparse matrix was used to represent an image which then had three filters applied to it via convolution. Much like Conway's Game of Life, the "next" (in this case "filtered") state of a given pixel in the image is calculated from the value of the pixel and its immediate neighbours, and this must be done "simultaneously" for each pixel. The difference here is one of representation; whereas Conway's Game of Life was an array of primitive integers, the images used in this benchmark are represented by a complex linked object structure. In the clocked version, the entire object graph is clocked via the reference to the root node of the matrix. In the unclocked version, it is necessary to update the current image state by replacing the reference with a reference to the next image state, and then re-initialising the next state to be an empty matrix.

Figure 7 gives the results for Sparse Matrix Convolution. Only a small number of points were sampled due to the very long execution time of this benchmark—but enough data were gathered to show that the clocked version of the Sparse Matrix is *vastly* slower than the unclocked version. Due to the single-write-per-phase nature of clocked variables, the Sparse Matrix was very slow to update. Each thread had to calculate the next value of its cells **before** any other thread could actually write to the matrix (as each cell insert necessitated a phase advancement, which—if peformed while threads were reading—breaks the convolution algorithm). After the values were calculated, each thread then inserts the new cell, advancing the phase after each insertion. Obviously this reduces the behaviour of the matrix to exactly that of the unclocked version—but with the high overhead of having to deep-copy the matrix at every clock advancement! Under a single-write-per-phase scheme, complex objects seem to perform quite slowly.

4.4 Linked List Microbenchmarks

For this benchmark, for linked lists of various sizes, the add and remove methods were executed a number of times. This benchmark mostly tests the overhead introduced by forcing the clocked list to be updated after *every* method call, as both were implemented in the exact same fashion, and both required the clock to advance after every method called on the list.

Figure 8 gives the results for clocked and unclocked Linked Lists. We can see that the clocked version of this data structure is much slower than the unclocked version. Every add, every remove—every operation that changes anything about the list—requires that the clock phase be advanced. This overhead simply does not exist in the unclocked version!

While clocked variables seem to offer some sort of benefit when used with primitives and objects with only primitive fields, they incur performance penalties with more complex data structures—at least, if we're restricted to one write per phase. Allowing multiple writes per clock phase might offer some performance improvements.

5 Alternate Approaches

It is obvious from the results presented in Section 5 that the performance of certain applications (i.e. Linked Lists) is heavily impacted by the inability to write to a clocked reference multiple times per phase. Why is this a restriction? If it can be shown that a given write is "safe", then what good reason is there

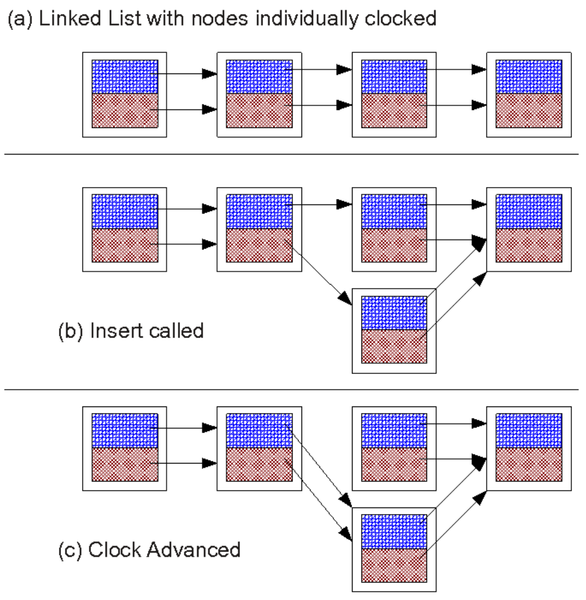


Figure 9: Approach 1: Clocking objects individually

for not allowing it? But before we can discuss that, we should look at what it means to be "safe". A "safe" write is any write to any part of a clocked object graph that (1) does not change the **structure** of the graph, and (2) does not involve a value that has been written to already during this phase. For example, it would be *unsafe* to add or remove a node from a linked list of integers, but it would be *safe* to alter the integer value store within a node—provided that value has not already been changed this phase. From this, we can immediately see that the Sparse Matrix benchmark is **not** safe, as some operations change the structure of the object graph (setting the value of a previously zero entry to a non-zero value). This was taken into account in the benchmark, and all updates to the object graph are performed atomically and are immediately visible to all threads—but this will not always be the case.

Once this difference in safety has been established, we can amend the requirement of a clocked reference to only allowing one *unsafe* write per clock phase. The issue then becomes determining what is a safe update, and what is not. Ideally, this would be done automatically by the compiler with no extra work required on the part of the programmer—but this would be require a means of determining every possible interaction that could occur with an object. Certainly possible for very simple objects, but the difficulty escalates quite rapidly.

5.1 Two Possible Approaches

Consider Figure 9. Under this approach, each object is individually clocked, allowing multiple updates to occur to the list—provided the updates don't affect the same object twice. Consider the example shown in the Figure: adding an item into a linked list cannot be safely done more than once per clock phase, as the second add operation simply **cannot** know about the previous addition, as it uses the readable versions of the objects to determine the current state of the list—these versions of the objects do not have any links to the new node! Thus the add operation replaces the next pointer of the *old* last node with a pointer to the second new node, erasing the *first* new node from the list. Multiple writes are unsafe under this approach.

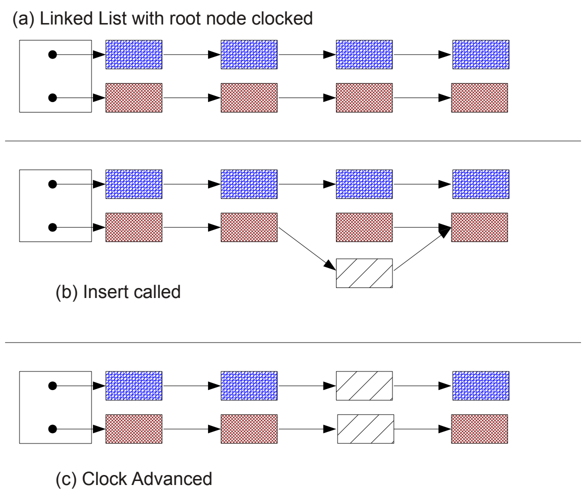


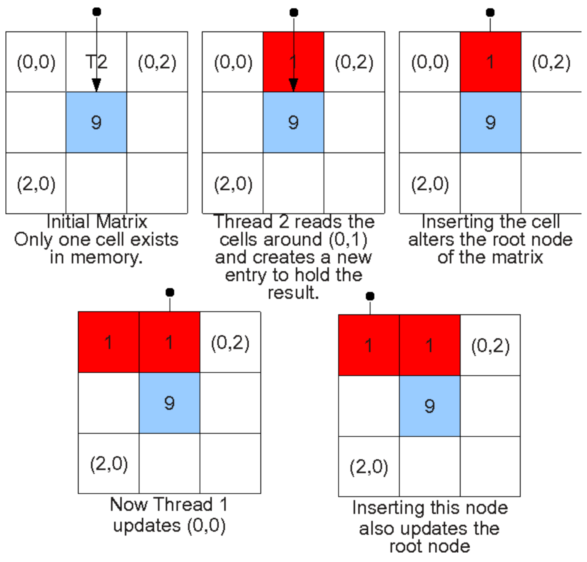
Figure 10: Approach 2: Single point-of-entry, clocking entire object graph

A second approach (Figure 10) attempts to solve this problem by splitting the object graph into two disparate graphs: a writable graph and a readable graph. This is the approach to Clocked References used earlier in this paper. We can see that doing this solves the issue of data loss, as each write operation is performed on the writable object graph, which is always the most up-to-date version of the object. The add operation is safe here, as the entire operation uses the writable object graph. But what about other operations? If we were maintaining a *sorted* list, adding a new node may not be safe, especially if the location that a node must be inserted is determined prior to calling any methods on the writable graph—instead, the location would be determined by the readable object graph, and so multiple additions—while no data would be *lost*—may result in the list no longer being sorted. We also see that this approach is not thread-safe, as multiple threads attempting to add nodes to the list would be prone to the usual issues of concurrent lists. To eliminate this, we must then state that every thread that wishes to use the writable object graph must obtain a lock on the root node in order to proceed. Thus every write is atomic and uninterruptable—but we have sacrificed parallelisation. This becomes a large issue with problems like the Game of Life, or image convolution: if each thread is only updating one node, and no node is being updated by more than one thread, then why *shouldn't* the threads be able to do this concurrently?

5.2 Two Better approaches

We can build on the first approach outlined above in order to make it slightly safer: we require that each thread lock the objects it needs to update. While these objects are locked, the thread uses the **writable** version of the object for **all** operations. This ensures that no data is lost, but brings new difficulties in ascertaining which objects a thread needs to lock in order to perform the operation successfully. It also raises concurrency issues: deadlock needs to be avoided, as it could be caused by two threads needing the same two objects, and locking them in different orders.

Our final approach attempts to solve this deadlocking problem by providing a single point of entry,



(0,0)
T2
(0,2)
9
(2,0)
Initial Matrix Only one cell exists in memory.
(0,0)
1
(0,2)
9
(2,0)
Thread 2 reads the cells around (0,1) and creates a new entry to hold the result.
(0,0)
1
(0,2)
9
(2,0)
Inserting the cell alters the root node of the matrix
1
1
(0,2)
9
(2,0)
Now Thread 1 updates (0,0)
1
1
(0,2)
9
(2,0)
Inserting this node also updates the root node

Figure 11: A Sparse Matrix Convolution

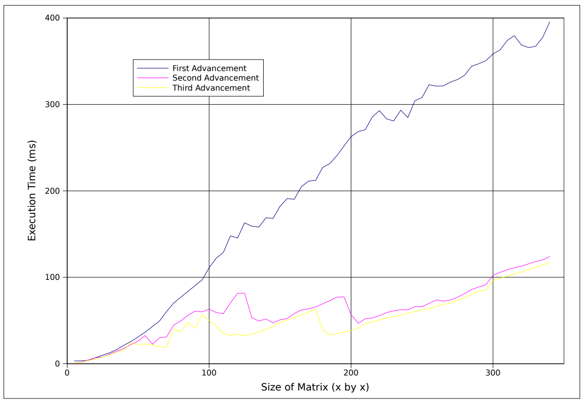


Figure 12: Clock Advancement Performance for a Sparse Matrix

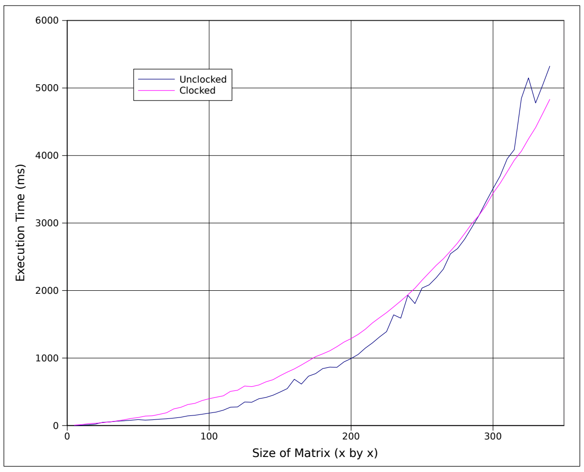


Figure 13: Using Approach 3 to solve the Sparse Matrix Performance Issue

similar to the second approach outlined above. When a thread needs to lock objects, it first locks the root object; thus any thread that needs to lock objects within the graph can do so without interfering with any other threads. This doesn't solve the issue of multiple threads needing to write to the same object. In this case, such a thread must wait until the next clock phase. So, what do these approaches look like in practice? A working implementation has not yet been developed, but Approach 3 lends itself well to *simulation*.

Figure 11 shows, we still cannot perform sparse matrix convolution—at least, not with this representation of sparse matricies. To understand this, we need to take a closer look at what is happening when a cell is updated. In the example shown, thread 2 is tasked with updating cell (0,1). To do this, it must first *read* cells (0,0), (0,1), (0,2), (1,0), (1,1), and (1,2). The result (1.00) is then written into cell (0,1)—but cell (0,1) does not currently exist in memory. So, the root of the matrix must be written to so that the cell can be inserted. The root holds a reference to the first non-zero cell in the first non-zero row, so currently it is pointing to cell (1,1). This reference needs to be updated, so we acquire a write-lock on the root node and insert the new cell.

Then, thread 1 attempts to update cell (0,0) via the same process. As this cell is before (0,1) in the row, the root needs to be updated again. Note that we have not yet advanced the clock. This requires obtaining a write-lock on the root, which throws an exception as the root has already been written to during this phase. We **cannot** solve this problem by advancing the clock before inserting (0,0), as this breaks the convolution algorithm. Cell (0,1) would be inserted into the matrix, and would thus affect any threads that have not yet read the old value of that location.

A solution could be to require all threads to perform their reads before **any** thread can write to the matrix. This would break each clock phase into two sub-phases—a read phase and a write phase. During this write phase, the clock can be advanced any number of times, as the old values are no longer required by the updating threads. However, this would result in the same level of performance as shown in Figure 7, as clock advancement is costly for a sparse matrix (Figure 12). It also renders the object unsafe to read from during the write phase, so any threads external to this process would be forced to wait until the update process had finished in its entirety.

As we can see from Figure 13, the performance of Sparse Matrix Convolution is much improved—but this relies on a safe way to update the matrix.

Another solution, perhaps, is to implement things in a safer way. If a Sparse Matrix were implemented such that the first cell in each row was *always* present, even if zero-valued, a lock could be acquired on an entire row of the matrix, making structural changes safer. This would require that each row be updated strictly by one thread, so we have lost some concurrency here—but the performance would surely be better.

5.3 Related Work

The basic concept underlying clocked variables is not a new one. Software Transactional Memory (STM) (Shavit & Touitou 1995) provides database-like transactions for operations on shared memory. A transaction consists of one or more write operations performed on an object, which is then *committed* once the transaction is complete, causing an atomic update on the object to be performed. Transactions can be *aborted* at any time, and the pending changes are lost. This is similar, in many respects, to how Clocked Variables work—with some key differences. Both operate in a "phased" fashion; for STM, these phases are transactions, and for Clocked References, the phases are literal clock phases. Both maintain the old version of the memory location for reading purposes during these phases, and both "commit" changes to memory at the end of each phase.

For X10 specifically, there has been work to develop Phasor Accumulators (Shirako et al. 2009). These accumulators provide support for the accumulation of multiple values during a single clock phase (thus allowing multiple updates to a value) while maintaining the value from the previous state for reading purposes—similar to the Clocked Variables described in this paper. However, Phasor Accumulators were only designed with Number types in mind, and do not address Reference Types.

The use of revisions and isolation types (Burckhardt et al. 2010) offers a similar functionality to the scheme presented in this paper. Programmers can declare data they wish to be shared between tasks by using *isolation types*. Tasks are then executed and merged using *revisions*: isolated instances that can only read and modify the shared data locally. When the tasks are finished, the runtime merges the results, automatically resolving any conflicts that occur. The result is a concurrent programming model that can distribute and share data without concern over concurrent modifications, and successfully merge this shared data back into a coherent whole. Under such a scheme, it would be possible to split an array (such as in the Game Of Life case study) across multiple tasks, have each task read and update their assigned cells, and have the array merged successfully back into a consistent board state. Such a process may be used to provide high-performance concurrent programs (Burckhardt et al. 2011) that greatly improve upon the expected performance gained by parallelization alone. However, it is unclear how Revisions handle complex object graphs, as this has not been specifically addressed; nor does it seem to address the case where objects have reference types as fields.

6 Conclusion

Clocked Primitives are the most viable form of clocked variable presented in this paper, and offer no significant change in performance. The benefit gained from using them is a cleaner way of updating dual-state variables often found inside concurrent code.

Clocked References, however, were the main focus of this paper. While our initial attempts at solving this problem were not entirely successful, we have presented our results and offered insights into what could be done to solve this issue. There are many options for future work with Clocked References, and many new avenues to explore.

The implementation presented in this paper is available from `http://ecs.vuw.ac.nz/~atkinsdani1/x10-clocked.tar.gz`.

References

Burckhardt, S., Baldassin, A. & Leijen, D. (2010), Concurrent programming with revisions and isolation types, *in* 'OOPSLA', ACM, New York, NY, USA, pp. 691–707.

Burckhardt, S., Leijen, D., Sadowski, C., Yi, J. & Ball, T. (2011), Two for the price of one: A model for parallel and incremental computation, *in* 'OOPSLA', Portland, Oregon.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. & Sarkar, V. (2005), X10: an object-oriented approach to non-uniform cluster computing, *in* 'OOPSLA', ACM, New York, NY, USA, pp. 519–538.

Ebcioglu, K., Saraswat, V. & Sarkar, V. (2005), X10: an experimental language for high productivity programming of scalable systems (extended abstract), *in* 'P-PHEC'.

Ebcioglu, K., Saraswat, V. & Sarkar, V. (n.d.), 'X10: Programming for hierarchical parallelism and non-uniform data access (extended)'.

Gardner, M. (1970), 'The fantastic combinations of John Conway's new solitaire game 'Life'', *Scientific American* **223**, 120–123.

Lamport, L. (1978), 'Ti clocks, and the ordering of events in a distributed system', *Commun. ACM* **21**, 558–565.

Murthy, P. (2008), Parallel computing with X10, *in* 'Proceedings of the 1st international workshop on Multicore software engineering', IWMSE '08, ACM, New York, NY, USA, pp. 5–6.

Nystrom, N., Clarkson, M. R. & Myers, A. C. (2003), Polyglot: an extensible compiler framework for Java, *in* 'CC', Springer-Verlag, Berlin, Heidelberg, pp. 138–152.

Saraswat, V. (2011), 'X10 design notes', `https://x10.svn.sf.net/svnroot/x10/documentation/trunk/x10.man/v2.2/design-notes/design-v08.txt`.

Saraswat, V. A., Sarkar, V. & von Praun, C. (2007), X10: concurrent programming for modern architectures, *in* 'PPoPP', ACM, New York, NY, USA, pp. 271–271.

Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. & Grove, D. (2011), 'X10 language specification', `http://dist.codehaus.org/x10/documentation/languagespec/x10-221.pdf`.

Shavit, N. & Touitou, D. (1995), Software transactional memory, *in* 'PODC', ACM, New York, NY, USA, pp. 204–213.

Shirako, J., Peixotto, D., Sarkar, V. & Scherer, W. (2009), Phaser accumulators: A new reduction construct for dynamic parallelism, *in* 'IEEE International Parallel and Distributed Processing Symposium'.

Hardware Trojan Resistant Computation using Heterogeneous COTS Processors

Mark Beaumont **Bradley Hopkins** **Tristan Newby**

Defence Science and Technology Organisation
Adelaide, Australia
Email: {mark.beaumont, bradley.hopkins, tristan.newby}@dsto.defence.gov.au

Abstract

Hardware Trojans pose a credible and increasing threat to computer security, with the potential to compromise the very electronics that ostensibly provide the security primitives underpinning various computer architectures.

The discovery of stealthy Hardware Trojans within Integrated Circuits by current state-of-the-art pre- and post-manufacturing test and verification techniques cannot be guaranteed. Therefore electronic systems, especially those controlling safety or security critical systems should be designed to operate with integrity in the presence of any Hardware Trojans, and regardless of any Trojan activity.

We present an architecture that fragments and replicates computation over a pool of Commercial-Off-The-Shelf processors with widely heterogeneous architectures. Processors are loosely synchronised through their use of a voted, architecture-independent message box mechanism to access a common memory space. A minimal Trusted Computing Base abstracts the processors as a single computational entity that can tolerate the effects of arbitrary Hardware Trojans within individual processors. The architecture provides integrity, data confidentiality, and availability for executing applications.

1 Introduction

Hardware Trojans are malicious modifications to Integrated Circuits (ICs) that can compromise the security of a hardware platform or any software running on it. They are persistent in nature and can operate continuously or be triggered into one or more actions, including modification of functionality, modification of specification, leaking of sensitive information, or Denial of Service (DoS) (Rajendran et al. 2010). The severity of Hardware Trojan action can range from minor through to catastrophic, such as the disabling of a major financial system causing economic loss or affecting a critical electro-mechanical system (Tsang 2009), leading to potential loss of human life.

Many different types of Hardware Trojans have been demonstrated, (Lin, Burleson & Paar 2009)(Baumgarten et al. 2011)(Jin et al. 2009), including malicious modifications to CPUs that have enabled privilege elevation and password stealing attacks (King et al. 2008). Whilst not a malicious Trojan, the Intel Pentium `f00f` bug (Collins 1998) demonstrated how a small design flaw in an IC could render a system vulnerable to a DoS attack.

The Hardware Trojan threat is increasing, especially amongst Commercial-Off-The-Shelf (COTS) components, where much of the IC development chain has been outsourced, relinquishing control over many potential Hardware Trojan insertion vectors. The past six years have seen increased research into methods for detecting Hardware Trojans. The primary methods involve self-checking systems, side-channel analysis and destructive reverse-engineering (Chakraborty et al. 2009). Even with the most recent advances in detection techniques, there are no guarantees that an IC is free of Hardware Trojans (Abramovici & Bradley 2009).

Economic and political rationale are driving increased globalisation, pushing *untrusted* COTS components into many electronic devices, including safety critical systems and sensitive military equipment (Young 2011). The cost of developing a Trojan-free IC is immense, requiring trust in many areas including design tools and teams, fabrication facilities, supply chains and anti-tamper technology. This approach is currently both technologically and economically infeasible, especially in an Australian Military context. To track technological advances, especially in relation to the latest processor architectures, accreditation of all components is not practicable, thus the use of COTS elements cannot be avoided. Instead, we advocate coupling the latest COTS technology with some small, accreditable trusted logic to form a Trojan-hardened system.

In previous work, the SAFER PATH architecture (Beaumont et al. 2012), a Hardware Trojan-resistant general computing platform, was proposed as a trusted drop-in replacement for a potentially compromised processor. The architecture combines many similar, cycle-accurate processors with a small Trusted Computing Base (TCB) to achieve replicated and fragmented execution. The architecture provides integrity and availability through majority voting of execution and protects data confidentiality by limiting any individual processor's access to program code and data. It relies on obtaining variations of the same processor for protection against identical Trojans.

In this paper, we introduce a modified version of the SAFER PATH architecture that abstracts a single computational entity from the collective behaviour of a pool of COTS Processing Elements (PEs) with *widely heterogeneous* architectures. Computation across multiple PEs is loosely synchronised via an architecture-independent message box (mbox) mechanism, allowing voted execution of an application. This execution is also fragmented in time across many different sets of PEs, limiting access to sensitive information for any individual PE. A software interpreter is demonstrated that provides a software abstraction, allowing a single, architecture-independent application to be collectively executed and fragmented across a pool of architecturally different processors.

The replication and fragmentation logic are part of a minimal TCB, providing the root of trust for the architecture. As such, effort must be put into ensuring that this logic is free of Hardware Trojans. Development and accreditation of the TCB logic is far more economically feasible than pursuing a complete trusted processor.

Trustworthiness is targeted at the expense of cost, performance, power usage and size. This is a design decision, but one that we believe needs to be made, especially for critical systems.

Our focus is on Hardware Trojans present within PEs, e.g. a CPU with local memory, and we aim to provide a broad spectrum defence against the effects of any Trojans present within these circuits. While we assume that any given PE may be infected, the likelihood of having identically functioning, or collaborative Hardware Trojans across many processors is considered very low, decreasing as the number of different processors is increased.

External to our abstracted computational entity, we provide no protection against Hardware Trojans in other ICs such as system-wide memory, or Input/Output (IO) circuitry. The architecture ensures that any given software is executed correctly as determined by the collective behaviour of multiple PEs. This architecture can be used as a trusted replacement processing element, with other defences able to be incorporated to protect the system as a whole, e.g., Bloom et al. (2009) protect against Trojans residing in memory using a double guard on the memory bus.

The paper is organised as follows: Section 2 discusses related work, Section 3 details our proposed solution and Section 4 describes our experimental implementation and results. Section 5 discusses some potential extensions while Section 6 summarises our work.

2 Related Work

There are existing commercial and industrial systems that provide availability, and protect functional integrity and data confidentiality. They often incorporate one or more of the following techniques: heterogeneous processors, redundant processing, software dissimilarity, voting, and data fragmentation. Hardware Trojan research has also incorporated some of these mechanisms.

Recently, the SAFER PATH architecture was developed incorporating fragmented execution and replication as a defence mechanism. SAFER PATH relies on obtaining variability between operationally equivalent processors to combat Hardware Trojans. Ensuring there is enough variability between processors to prevent the same Hardware Trojan appearing is difficult, requiring sufficient orthogonality between the design teams, design software, and fabrication facilities. It is also difficult to obtain this variability off-the-shelf, meaning that processors would need to be customised. Utilising a new type of processor would require significant effort. The same Hardware Trojan might also be more easily inserted into variants post-manufacture, given that all processors must adhere to the same operational interface.

In contrast, the architecture presented here uses truly heterogeneous, unmodified COTS processors, allowing new types of processors to be easily added. Every processor in the architecture can be different, increasing the barrier for any collaborative Hardware Trojan insertion.

Yeh (1996) describes the use of triple modular redundancy using heterogeneous PEs, majority voting and N-version dissimilar software to achieve high levels of reliability in the Boeing 777 primary flight computer. A low-level communications bus is used for synchronisation between varying processor channels. The system is only used to process simple inputs and outputs, and only outputs are voted on. Saxena and McClusty (1998) use redundant simultaneous multithreading to achieve fault detection and recovery at a software level. In more recent work, Reis et al. 2005 employ compiler-based transforms that duplicate instructions and insert checkpoints for fault detection. These systems provide protection against transient faults, as opposed to Hardware Trojans which may not manifest as a fault, but rather a subtle change to a processor's behaviour, or the leaking of sensitive information.

McIntyre et al. (2010) propose a software fault-tolerant technique, the Trojan Aware Distributed Scheduling (TADS) system. TADS operates on a multi-core compute platform potentially containing one or more Hardware Trojans. A scheduler is used to execute functionally equivalent subtask variants on different cores within the processor. Results are evaluated for equivalence and any disparity is used as an indicator of Hardware Trojan presence. This process is repeated and the scheduler is able to progressively establish trust in the circuitry of each core. Software variants provide course-grained protection and require program diversity through recompilation. This architecture protects against simple Hardware Trojans, but is vulnerable to more sophisticated Trojans (e.g., King et al. 2008) that may be replicated across processing cores.

Other research has proposed Data Guards (Bloom et al. 2009) (Waksman & Sethumadhavan 2011) and reconfigurable logic (Baumgarten et al. 2010) to counter the presence of Hardware Trojans within ICs. These solutions are focused on protecting against specific Hardware Trojan triggers or actions. We make no such assumptions about the type of triggers that may exist or the actions that may result, and presume Hardware Trojans may be active within all our processors.

3 Architecture

Modern computing systems typically entrust one or more COTS Processing Elements (PEs) to reliably execute programs. Our assumption is that any of these individual PEs may be infected by one or more Hardware Trojans, consequently compromising security by modifying the behaviour of the program or leaking data.

Our architecture uses a pool of many architecturally different PEs together with a small Trusted Computing Base (TCB), to collectively execute a given application. The architecture enables the external behaviours of simultaneously executing PEs to be supervised, making no assumptions about the internal operation of individual PEs. All PEs run independently of each other, executing their own code from a locally attached memory. Low-level computation is not replicated, instead, some of the external behaviours of the PEs are replicated and arbitrated by the TCB to coordinate a collective behaviour across the pool of PEs.

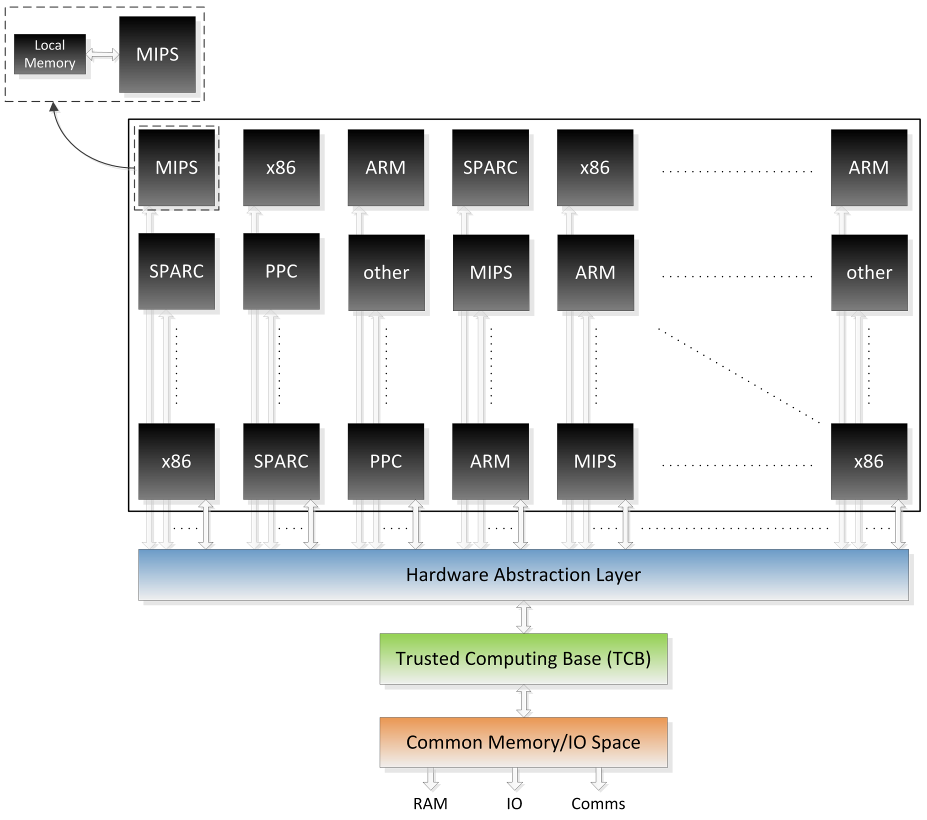


Figure 1: A set of PEs collectively execute a given application.

To support this behavioural replication across different architectures, a Hardware Abstraction Layer (HAL) provides independent access to a common memory and IO space. The HAL unifies accesses using a message box (*mbox*) associated with each PE. Mboxes provide a register-style interface to access the common resources. The architecture is shown in Figure 1.

Software applications are independently compiled for each PE architecture, and all accesses to the common memory and IO space are made through architecture-specific mbox routines. The compiled applications execute concurrently across a set of PEs, but execute different machine code and have different mbox access timing. To obtain collective behaviour from a subset requires the mbox accesses to be strictly ordered. At any instance in time, execution is loosely synchronised across a set of PEs, enforced by the TCB which arbitrates access to the common memory space using a simple voting mechanism.

The TCB also facilitates time-domain fragmentation of program behaviour across multiple independent sets of PEs from the pool, protecting against side-channel data leakage attacks (Lin, Burleson & Paar 2009) (Lin, Kasper, Paar & Burleson 2009). Application synchronisation between sets is maintained by storing selected elements of program state in the common memory.

3.1 Processing Elements

Our architecture supports the use of almost any type of COTS processing element, e.g., ARM, MIPS, x86, SPARC, to form a large resource pool. This architectural diversity minimises the probability of colluding or replicated Hardware Trojans existing within different PEs.

Independently infected PEs alone cannot compromise the integrity of the computation; an adversary would need to influence multiple designs, fabrication facilities, or supply chains to bypass this diversity. New types of PEs can always be incorporated into the architecture to maintain this diversity and track technological developments.

3.2 Message Boxes

Heterogeneous PEs have different bus interfaces, timing characteristics and byte orderings, making it difficult to combine their behaviours, especially at a low level. The mboxes ensure each architecture can access the common memory and IO space, facilitating synchronisation and voting. The mboxes form a trust boundary between a PE and the TCB. A simple interface enables easy system integration, promotes simple TCB design and allows the architecture to scale to a large number of PEs.

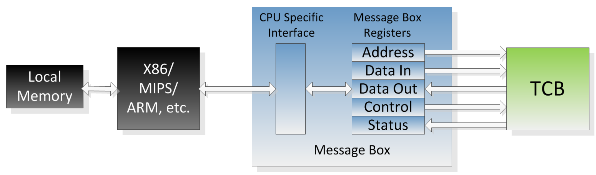


Figure 2: mbox register interface.

Access to common memory and IO is abstracted through a transaction style approach using registers for address, data, control and status as shown in Figure 2. When a PE wishes to read or write to the common memory, it writes the address (together with the data for a write) and then signals through the control register for the operation to be performed. The TCB decodes the address and forwards the request to the appropriate memory or IO resource. Once the operation is complete (and data returned for a read) a status register bit is set.

Individual PEs could potentially communicate with an mbox using any available communication interface, for example a memory bus, GPIO port, or USB or PCIe interface. However, the implementation must consider potential Hardware Trojan interference. Strict separation between mbox communication channels must be maintained, and, where mboxes are not part of the TCB, they also require diversity in their design and manufacture to counter replicated or colluding Hardware Trojans.

3.3 Loosely Synchronised Execution

The TCB interfaces with the mboxes and combines the multiple access requests into a single collective request to the common memory and IO space. Read accesses result in the same data being returned to all PEs. Figure 3 shows a set of PEs writing data to a common memory address; specifically shown is the asynchronous nature of the requests from the different PEs.

The same, single application is independently compiled for and subsequently run on each PE. Although compilation is generally from the same source code, there may be significant differences in the respective machine code representations. Loose synchronisation between the executing programs is maintained by ensuring each PE attempts the same mbox accesses in the same order. This strict ordering is enforced during the software development for the architecture, with consideration given to different architectures, programming languages, and compilers. The TCB generally blocks on these access requests until all PEs have updated their respective mboxes. The TCB then votes on the requests and performs the actual memory access to the common memory space. This synchronises execution of the application on all mbox operations and ensures voting occurs on the same accesses.

Voted output guards against Hardware Trojans attempting to modify program behaviour. It also prevents leakage of confidential data through logical channels, either common memory or IO. While we implement a majority voting mechanism in our concept demonstrators (Section 4), different access aggregation policies may be used. Individual PE accesses may be weighted or even ignored by the TCB when generating the voted output. Such voting schemes can help protect against potential DoS attacks. The voting algorithm can be adaptive and designed with potential risk profiles for different PE architectures taken into account. Changes to the TCB need to balance performance and complexity.

3.4 Fragmentation

Even when the direct outputs of a collective program are protected, Hardware Trojans may still be able to leak data through side-channels. To combat this, the execution of the collective program, i.e. its behaviour, is fragmented in time across many different sets of PEs. A given set runs a *fragment* of the collective program before execution of that program is switched to another set of PEs. Fragmentation of program execution is achieved by transferring the currently executing program context from one set of PEs to a different set of PEs.

In contrast to the SAFER PATH architecture, an individual PE's program code itself is not fragmented. Instead, all PEs in the architecture run continuously, however only a subset have access to the common memory and IO space at any given time. The TCB assembles these subsets of PEs and enables or disables their access to the common memory.

Application instances, on any particular PE, are informed when they become connected to the common memory space. Application support for fragmentation involves the unique instances running on

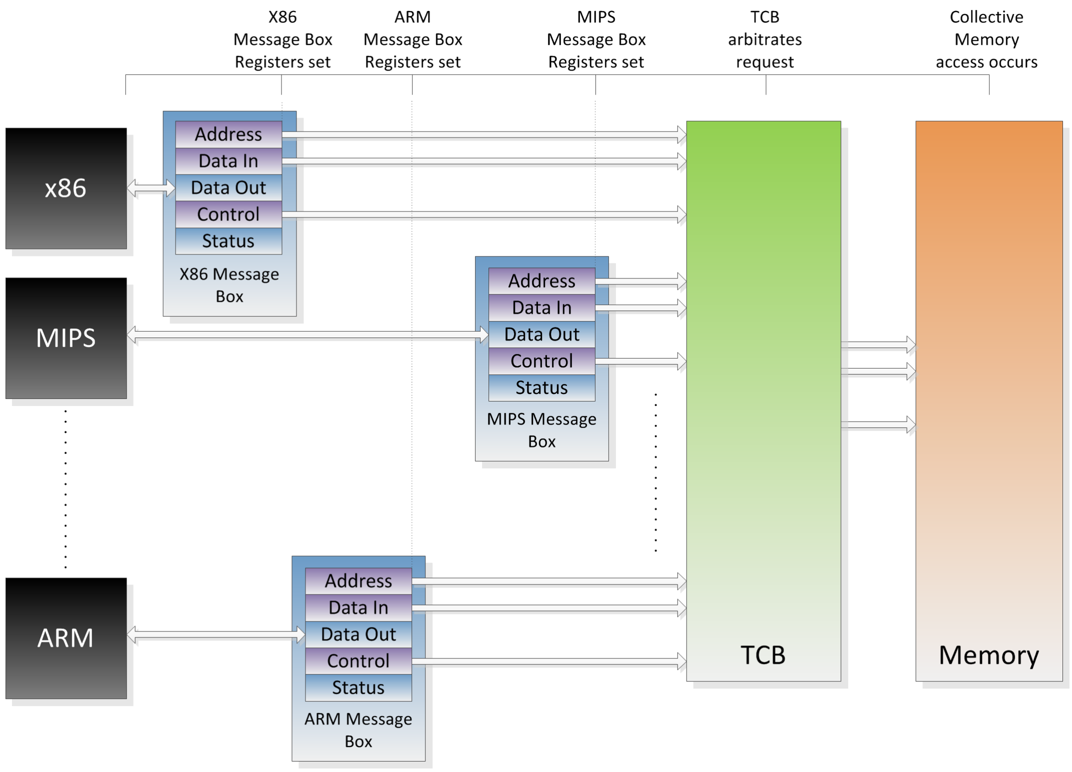


Figure 3: The TCB arbitrates a write access to common memory.

the currently active set of PEs saving collective program context to the common memory. The TCB can then switch its mbox interfacing to a new set of PEs. The application instances running on the new set of PEs load the saved context from the common memory and continue the collective execution. Saving and loading of this program context is implicit if all necessary program variables are permanently stored and accessed through the TCB protected common memory. A context switch can either be initiated from the software application, or mandated at periodic intervals by the TCB.

This form of context switching restricts an individual PE's access to sensitive data, limiting what information a potentially infected PE may leak. It also restricts any Hardware Trojans from understanding the broader scope of executing applications, making it difficult to interpret what data is currently being processed by a PE, and thereby increasing the complexity requirements of such Trojans. For example, execution may be fragmented to restrict individual PE access to an encryption key or sensitive report.

3.5 Trusted Computing Base

The TCB contains minimal logic to enforce the collective behaviour. A benefit of our architecture is that the TCB is a small, generic design that can be used with many different PEs, providing a more tractable and flexible solution than developing a custom trusted processor. While the TCB provides integrity, availability and confidentiality, the outputs are not necessarily correct; rather they reflect collective behaviour. As the number of PEs is increased, the outputs become probabilistically correct and more resistant to Hardware Trojans.

The design consists primarily of voting and switching logic plus additional ancillary circuits for the purpose of enforcing time-windows on mbox accesses. The simplicity and small size of the TCB relative to an individual PE assists both accreditation and subsequent design and fabrication free of Hardware Trojans. The TCB may include the mboxes or just an interface to the mboxes. This decision is dependent on obtaining variant mboxes that do not need to be trusted.

The TCB must also prevent misuse of the architecture. Rogue PEs may delay or insert additional mbox accesses in an attempt to degrade service. The heterogeneous processing nature of the architecture requires the TCB to aggregate accesses that are asynchronous. The TCB can use a time-window to ensure timely access synchronisation. If PEs violate this timing they can be blacklisted and removed from the set, or in the worst case, the TCB can reset all processors. A larger pool of PEs reduces the influence of this issue.

In our proposed architecture, the TCB arbitrates access to common memory and IO devices. Other peripherals could also be supported, such as system timers and interrupts, to enhance software application support, and enable more complete systems to be protected by the architecture. The trade-off for this convenience is the size and complexity of the TCB.

4 Experimentation and Results

The architecture was prototyped within a Xilinx Virtex 6 FPGA. A pool of embedded soft-core processors, an mbox for each processor, the TCB logic, and the common memory were all implemented within the FPGA. Three different processor architectures were used: *leon3* (Aeroflex Gaisler AB 2010), a 32-bit SPARCv8 processor; *mblite* (Kranenburg & van Leuken 2010), a 32-bit MIPS based processor; and *zpu* (Zylin Consulting 2008), a tiny, 32-bit stack based processor.

Each type of processor was configured with enough local memory (*leon3*: 16kB, *mblite*: 16kB/16kB, *zpu*: 32kB) for the example programs to run natively.

The mboxes associated with each processor were connected using a GPIO port native to each architecture. Each mbox consisted of a 32-bit address port, a 32-bit data-in port, a 32-bit data-out port and a control/status port. The mapping of memory and IO peripherals in this address space is application-specific.

Developing an application for the architecture requires identifying important information or computational actions to protect. Important computations need to be replicated and voted upon. Likewise, sensitive information must be fragmented across different sets of processors. For a custom application this normally entails voting on all accesses to the system inputs and outputs and supporting the fragmentation of the application across multiple subsets of processors. To achieve this, each natively executing processor in the currently executing set must perform the same common memory accesses in the same order.

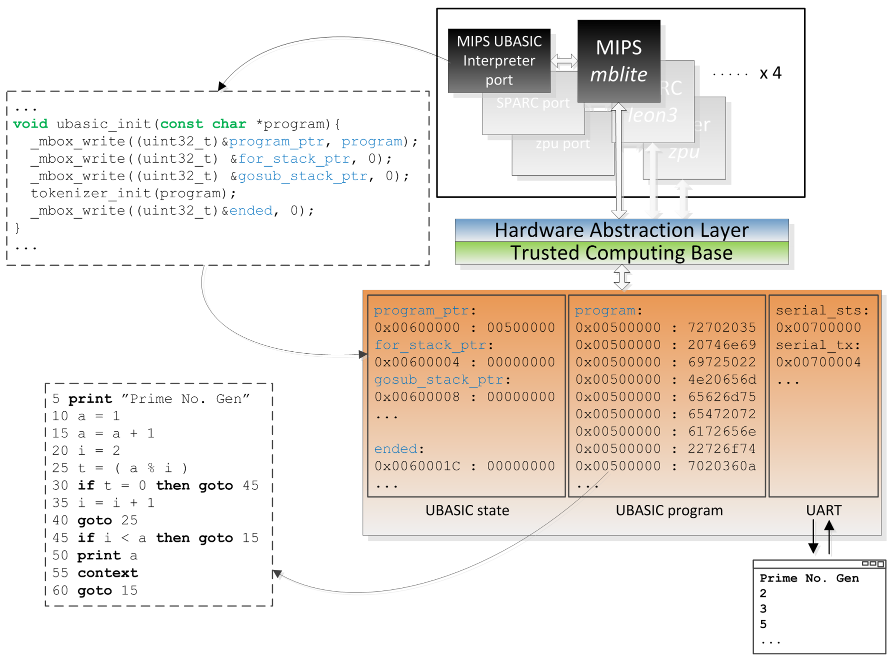
Two example programs were developed. The first is a software interpreter where the interpreted program is stored and accessed through the TCB protected common memory. The output of the interpreted program occurs through a serial port, which is also mapped into the TCB protected common memory space. The second example program is a VNC client, where the network IO, keyboard and mouse inputs, and framebuffer outputs are mapped into the common IO space and protected by the TCB.

4.1 Software Interpreter

A consequence of using many different processor architectures is that programs need to be compiled for each architecture. To alleviate this requirement, we employed a software interpreter. Though the interpreter itself runs natively on each processor and, hence, needs to be compiled for each different architecture, once this has been done, many different programs can be run on top of this interpreter without needing to be rewritten or recompiled for the underlying architectures. A secondary benefit of the interpreter is that it also allows computation to be voted upon and fragmented, in addition to just the system IO. The fragmentation is trivially achieved because the interpreted program and interpreted state is entirely stored and accessed through the TCB protected common memory.

We utilised a pool of 12 processors divided into four sets, with each set containing one *mblite*, one *leon3*, and one *zpu* processor, each running at 25MHz. Input and output to the architecture is provided by an asynchronous serial port that is mapped into the common IO space. Synchronisation and fragmentation are supported by a 16kB TCB protected common memory.

The *ubasic* (Dunkels 2007) BASIC interpreter was ported to the each native processor architecture. The native code for each interpreter is stored in and executed from the local memory attached to each processor, i.e., 12 separate instances running in our experimental set up. Collectively, the processors interpret a single, architecture-independent BASIC program. The BASIC program to be interpreted (e.g., Listing 2), is stored in the common memory space. The *ubasic* implementation was modified so that calls to



```
...
void ubasic_init(const char *program){
  _mbox_write((uint32_t)&program_ptr, program);
  _mbox_write((uint32_t) &for_stack_ptr, 0);
  _mbox_write((uint32_t) &gosub_stack_ptr, 0);
  tokenizer_init(program);
  _mbox_write((uint32_t)&ended, 0);
}
...
```

```
program_ptr:
0x00600000 : 00500000
for_stack_ptr:
0x00600004 : 00000000
gosub_stack_ptr:
0x00600008 : 00000000
...

ended:
0x0060001C : 00000000
...
```

```
program:
0x00500000 : 72702035
0x00500000 : 20746e69
0x00500000 : 69725022
0x00500000 : 4e20656d
0x00500000 : 65626d75
0x00500000 : 65472072
0x00500000 : 6172656e
0x00500000 : 22726f74
0x00500000 : 7020360a
...
```

```
serial_sts:
0x00700000
serial_tx:
0x00700004
...
```

```
5 print "Prime No. Gen"
10 a = 1
15 a = a + 1
20 i = 2
25 t = ( a % i )
30 if t = 0 then goto 45
35 i = i + 1
40 goto 25
45 if i < a then goto 15
50 print a
55 context
60 goto 15
```

```
Prime No. Gen
2
3
5
...
```

Figure 4: Operation of the software interpreter.

memory accesses associated with the BASIC program were replaced with accesses through the mboxes.

Simple asynchronous serial output was also provided through the common memory space. The BASIC program was able to write out this serial port using the *print* keyword. An instructive example of accessing the serial port via an mbox is given in Listing 1. When the *print* keyword is interpreted, the `outbytem` function is called for each character to be printed. The `_mbox_read` and `_mbox_write` calls ensure that reads from the serial port status register, and writes to the data (output) register are synchronised and voted on across the currently active set of heterogeneous processors.

```
void outbytem (char c) {
  do {
  } while ((_mbox_read(STATUS_REG) &
               BIT_SET(SERIAL_XMIT)) == 0);
  _mbox_write(SERIAL_REG, c);
}
```

Listing 1: Mbox access to a collectively controlled serial port.

In the demonstrator, all processors concurrently execute the *ubasic* interpreter, however, only a selected set of processors are connected to the common memory and thus interpret the instructions of the BASIC program at any one time. The *ubasic* software running on processors that are not connected remains in a waiting state until their connection to the common memory is restored. A read of a status bit through the processor's mbox indicates whether the common memory is connected. Once connected, the interpreters exit their waiting state and continue interpreting the BASIC program.

Enabling fragmentation of the BASIC program required further modification of *ubasic* to allow the execution context of the interpreter to be passed from the current subset to the next subset of processors. To support saving and loading of this execution context, some of the interpreter's state, including the program counter (a pointer into the BASIC program), stack pointers and global variables, is stored in and accessed through the common memory. A new keyword, *context*, was added to the *ubasic* instruction set that allows software to initiate fragmentation to a different set of processors. This is achieved via a write to an mbox control register bit. As with all mbox accesses, this write is voted on before being performed by the TCB, ensuring the context switch is a collective request.

The operation of the software interpreter is shown in Figure 4. A subset of processors connected to the common memory space accesses and interprets the BASIC program. The keyword *context* initiates a context switch to a new subset of processors and finally the new processors continue interpretation of the BASIC program. In this example, the program counter stored in the common memory is utilised to pick-up execution where the previous processors finished.

4.1.1 Performance

An example program (Listing 2) calculates prime numbers in a simple manner, performing an execution switch (line number 70) after printing the value of each successive prime.

```
5  print "Prime No. Gen"
10 a = 1
15 print "2"
20 a = a + 2
25 i = 2
30 t = (a % i)
35 if t = 0 then goto 60
40 b = a / i
45 if i > b then goto 65
50 i = i + 1
55 goto 30
60 if i < a then goto 20
65 print a
70 context
75 goto 20
```

Listing 2: BASIC prime number generator.

The performance of the *ubasic* interpreter on our architecture was analysed, using the BASIC program in Listing 2. The blocking nature of the mbox calls means a side-effect of loose synchronisation is to reduce the performance of the architecture to that of the slowest processor, the *zpu* processor in our experiments. This is not a problem if all processors used in the architecture have sufficient performance. Accessing the common memory also incurs a performance cost as a result of the overheads involved with mbox indirection.

Different types of interpreted programs will require different numbers of mbox calls. Highly algorithmic programs would spend more time executing native calculations and would see less of a performance hit compared with memory access intensive programs.

The benchmark was to find all the prime numbers less than 5000 and was run on three architectural variants: a single *zpu* processor; a set containing all three different processors without any context switching, and the full architecture of four sets of all three processors. The results are shown in Figure 5.

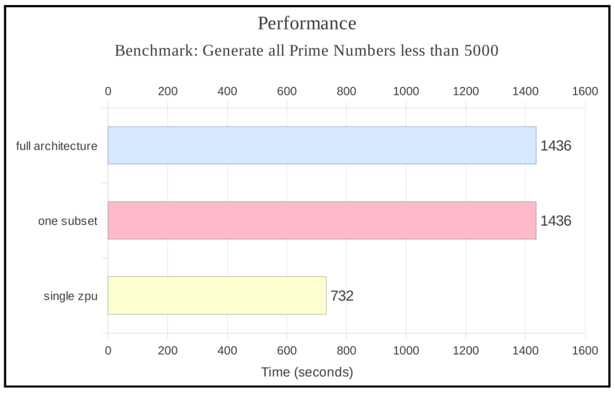


Figure 5: Performance benchmarking of the software interpreter.

The overhead of the mbox accesses increases run time for the example program from 732 seconds to 1436 seconds, this equates to a performance decrease of around 49%. However, even this kind of performance decrease would be acceptable in many safety or security critical applications. The addition of fragmentation through context switches adds no discernible run-time to the application. This is due to the context switch requiring no explicit state saving or restoring to occur.

Also of interest is the total number of read and write accesses through the mbox interface for the benchmark; this information is shown in Table 1.

Lines of Interpreted Code	264697
Number of *mbox* Read Accesses	42678143
Number of *mbox* Write Accesses	10278655
Avg. *mbox* Read Accesses per line	161.23
Avg. *mbox* Write Accesses per line	38.83

Table 1: Analysis of *mbox* accesses.

The *zpu* implemented in our experiment is a very poor performing processor. With the *zpu* executing natively at 25MHz, the average mbox access takes approximately 330 clock cycles. A typical `_mbox_read()` call as shown in Listing 3 expands to over 70 instructions on the *zpu*, which are executed at between four and five clock cycles per instruction.

The `_mbox_read()` call comprises four fixed mbox register writes (three to `mbox_ctrl`, one to `mbox_addr`), one fixed mbox register read (`mbox_datai`), and potentially multiple reads from the mbox status register (`mbox_sts`). In our experimental architecture once the address (`mbox_addr`) and control (`mbox_ctrl`) registers have been written it takes 3 clock cycles for the TCB to return the data, hence for the slowest performing *zpu* processor it will only need to read the status register (`mbox_sts`) once.

```
unsigned long _mbox_read(unsigned long addr)
{
  unsigned long returnValue;

  *mbox_addr = addr;
  *mbox_ctrl = 0x0;
  *mbox_ctrl = MBOX_READ | MBOX_ENABLE;

  while ((*mbox_sts & MBOX_RDY) != MBOX_RDY);

  returnValue = *mbox_datai;
  *mbox_ctrl = 0x0;

  return(returnValue);
}
```

Listing 3: `_mbox_read()` call.

Increasing the amount of data and state stored in common memory, and hence the required number of mbox accesses, facilitates easy fragmentation and better restricts an individual processor's access to program code and data. Decreasing what is stored in the common memory improves native performance, at the expense of more complex software support for fragmentation and lower data confidentiality.

No effort was placed into optimising the *ubasic* interpreter code to reduce the number of mbox calls, or increase the general efficiency of the program. Opportunity exists to perform multiple computations with global variables without having to read and write them back to the common memory, thereby reducing the number of mbox calls, and increasing performance.

4.1.2 Trojan Resistance

The example program fragmented the generation of prime numbers, with the TCB randomly selecting a new subset of processors to calculate every new prime number. On average, each processor only had access to one in every four prime numbers. This approach demonstrates that as the number of processors is increased, and with the use of judicious context switching, this architecture is capable of successfully partitioning sensitive information across different processors.

We developed several other BASIC programs to run on our architecture. These included a *pi* estimation program and a simple "access" type program that asked for a password and granted or denied access based on whether a hash of the supplied password matched a stored hash value. Context switches occurred after each character was read from the keyboard and the generated hash updated. Again, in a simplistic manner this demonstrates how individual processors, and hence any associated Hardware Trojans might be prevented from having access to sensitive data in its entirety.

Using the common memory to store shared program code and data enables execution state to be switched between different sets of processors. The protection against data leakage then depends on the frequency and granularity of fragmentation. Performance is affected by the amount of data accessed through the common memory and there is a trade-off between security and efficiency. This trade-off is able to be managed by the software developer. The mbox architecture features can be used to protect only the most important data structures, thereby minimising the performance hit, or they can be broadly used as is the case for the software interpreter.

The replication of execution prevents any minority set of processors from modifying the behaviour of the program (either outputs or execution of the BASIC program) or from leaking data through a logical system interface. A unanimous voting mechanism was implemented in the TCB for our experiments. Whenever any one of the three processors in the currently active subset was reset, or attempted an incorrect (unordered) mbox access, the system halted. For protection against DoS attacks, and functional and behavioural modification, different voting mechanisms would need to be prototyped.

4.1.3 TCB Analysis

The TCB for this architecture consists of a multiplexer/demultiplexer and voting arrangement operating on the mbox interfaces. The architecture of the TCB exhibits similar properties to that of SAFER PATH. Table 4.1.3 compares synthesised resource usage within the Xilinx Virtex 6 FPGA for a minimal processor core (Leon3) against the TCB for one (b = 1) and four (b = 4) subsets (or banks) respectively. Each subset contains 3 different processors.

	LUT6s	Registers
Single Leon3 core	2516	1221
TCB, 3 PEs (b=1)	125	334
TCB, 12 PEs (b=4)	557	1158
TCB[2] 3 PEs (b=1)	213	71
TCB[2] 12 PEs (b=4)	708	76

[1] Results obtained using Xilinx ISE Release 14.2
[2] SAFER PATH TCB

Table 2: TCB size analysis.

The TCB logic remains smaller than a single processor up to a threshold number of processors. The TCB is made up of simple, replicated circuitry that can more easily be checked for correctness than for example a CPU Arithmetic Logic Unit (ALU). As the number of processors increases, the TCB scales linearly due to the increasing size of multiplexers and demultiplexers. In contrast to SAFER PATH, the current design implementation registers external memory and message box inputs and outputs thus resulting in a high register count. With further optimisation, register usage could be reduced with minimal impacts on design performance.

The mboxes have not been optimised for performance, however they could be tailored either for specific applications or processor architectures. Mboxes with deep data registers and increased voted block size could also be considered.

4.2 VNC Client

The software interpreter shows how the architecture can be used, and even abstracted from a software point of view. This second experiment was performed to demonstrate how a larger, more complex application could be ported to the architecture. The rationale behind choosing a VNC Client is to provide a simple thin-client, where Hardware Trojans residing within the processing elements could not compromise the session.

A VNC client was implemented on a version of the architecture that included only three processors. To support this, a serial port, PS2 keyboard, PS2 mouse, and framebuffer memory were mapped into the common memory and IO space, with access supervised by the TCB. The VNC client communicates with a server through the serial interface, reads in keyboard and mouse events through the PS2 interfaces, and writes to a display via a double-buffered 800x600 framebuffer memory. The mbox connected peripheral hardware is shown in Figure 6.

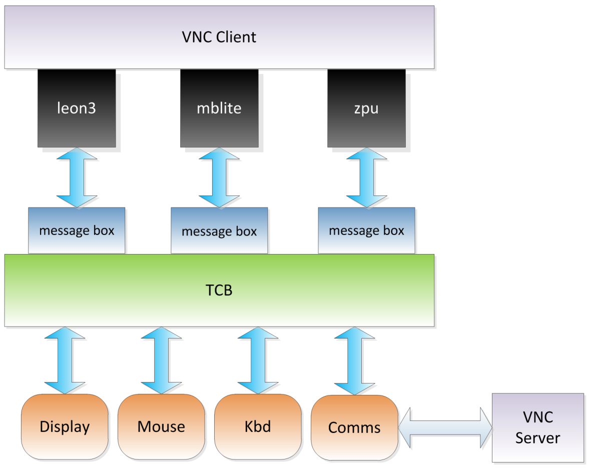


Figure 6: VNC client architecture.

The *fbvnc* (Weidner 2000) VNC client was modified and ported to each of our three implemented architectures (*leon3*, *mblite* and *zpu*). Minor architecture-specific code differences are required and the generated machine code is vastly different for each processor, but the ordering of mbox accesses to the common memory space is maintained. For example, as the program executes on each architecture, each processor reads from the keyboard, mouse or serial ports, and writes to the display or serial ports in the same order.

The VNC client communicates over a 1Mbps serial link, proxied via a network connection to a VNC server. Using hextile encoding, acceptable performance is obtained running the processors at 100MHz.

In arranging our architecture in this manner, we are able to ensure that each VNC client is provided with identical inputs, and that those inputs generate identical outputs. The synchronised execution and voting provides protection from malicious modification via the untrusted processors. However, in this instance, sensitive information that is being processed by the VNC client may be able to be leaked by an infected processor. The VNC communications protocol is modular, so fragmentation could be added to improve data confidentiality. Access to keyboard and mouse inputs, and data destined for the framebuffer would then be limited to small windows for each processor, helping to mitigate the damage of any data leakage.

The VNC client shows a how a more complex application can be implemented. Here the architecture is usefully applied to protect the inputs and outputs of a system from Hardware Trojan interference.

4.3 Summary

The two demonstrators show how applications can be protected against the threats of Hardware Trojans.

The software interpreter maintains all the protection properties of the earlier SAFER PATH architecture; a single program code, in this instance a BASIC program, can have its execution replicated and fragmented over many different processors. Sufficient performance remains in the architecture for successful application within a security critical system, while the TCB remains simple enough for accreditation.

Interpreting a program is inherently slower than native execution; this is true for all platforms. However, interpretation brings us the benefits of programming language abstraction and allows us to write a program once and run it anywhere. This is especially true for this architecture where the overheads of writing a custom application are high. The utility of a generic interpreter was demonstrated when newly written BASIC programs were able to immediately take advantage of our architecture's replication and fragmentation properties, with minimal to no work required of the programmer.

Although the use of the *ubasic* interpreter has merit for our experimentation, a different interpreter, for example a Java Virtual Machine (JVM), that has a more efficient byte code representation and better execution efficiency may provide improved performance. This improved performance comes at the cost of a larger initial effort to port the code to multiple architectures and to add support for fragmentation. The complexity of the TCB also increases if support is required for real-time features, e.g., timers and interrupts. However, a JVM would also provide the opportunity to access the large existing Java byte-code application base.

Increasing the barrier for successful Hardware Trojan operation forces Hardware Trojans to become more complex, usually translating into a larger implementation footprint. This makes them more easily detected through current Hardware Trojan detection mechanisms.

5 Further work

Our experimentation ran native applications on bare metal processors. The architecture works equally as well for more complex processors running multi-threaded operating systems. This holds as long as strict ordering is maintained through a dedicated mbox interface for any specific application that is to be protected on the architecture. Hence protected applications can run along-side less trustworthy applications on the same processor. The multi-threading nature of the underlying operating system also ensures a processor can still be usefully occupied while blocking on mbox accesses of the protected application. Further, whilst our experimentation was focused around FPGA development, the architecture is not limited to FPGA instantiation. Discrete processors could be combined, either at a macro level or together on a PCB like substrate to form a Hardware Trojan resistant computing platform.

Improving application design, mbox design and link speed, or enabling concurrent use of all available PEs could improve performance. Mboxes could be extended to distribute interrupts via register style interfaces, with consideration given to the impact on synchronised execution.

Algorithms for tuning fragmentation to achieve optimal data confidentiality properties should be investigated. These may be instrumented through the software build process or by source to source transforms enabled through formal methods.

6 Conclusion

The architecture presented allows computation to be replicated and fragmented across a pool of widely heterogeneous processors. Unlike SAFER PATH, there is no longer a requirement to obtain variants in manufacturing or design. Our updated architecture can be implemented using entirely COTS processors.

A minimal TCB, amenable to accreditation, votes on loosely synchronised, but replicated behaviour. This collective behaviour is probabilistically correct, providing integrity and availability in the presence of active Hardware Trojans. Further, fragmenting this behaviour limits individual processor access to data and defends against data leakage attacks.

A prototype implementation within an FPGA and two software applications were developed to demonstrate the utility of the architecture. The first was a software interpreter executing arbitrary programs, with an acceptable performance decrease for intended security critical applications. Extending the software interpreter from BASIC to a more sophisticated platform, such as a JVM would dramatically increase the utility of the system. The second application was designed to protect a VNC session executing on a thin client with untrusted COTS processors.

These applications demonstrate the use of the architecture as a replacement for an embedded or desktop processor, especially in circumstances where system operation needs to be guaranteed, or where sensitive data is being processed.

References

Abramovici, M. & Bradley, P. (2009), Integrated Circuit Security: New Threats and Solutions, *in* 'Workshop on Cyber Security and Information Intelligence Research', CSIIRW'09, ACM, New York, NY, USA, pp. 55:1–55:3.

Aeroflex Gaisler AB (2010), 'Leon3 Multiprocessing CPU Core Product Sheet'. http://www.gaisler.com /doc/leon3_product_sheet.pdf.

Baumgarten, A., Steffen, M., Clausman, M. & Zambreno, J. (2011), 'A case study in hardware trojan design and implementation', *Int. J. Inf. Secur.* **10**, 1–14.

Baumgarten, A., Tyagi, A. & Zambreno, J. (2010), 'Preventing IC Piracy Using Reconfigurable Logic Barriers', *IEEE Des. Test. Comput.* **27**(1), 66–75.

Beaumont, M., Hopkins, B. & Newby, T. (2012), SAFER PATH: Security Architecture using Fragmented Execution and Replication for Protection Against Trojaned Hardware, *in* 'Design Automation and Test in Europe (DATE)', pp. 1000 –1005.

Bloom, G., Narahari, B., Simha, R. & Zambreno, J. (2009), 'Providing secure execution environments with a last line of defense against Trojan circuit attacks', *Computers & Security* **28**(7), 660 – 669.

Chakraborty, R., Narasimhan, S. & Bhunia, S. (2009), Hardware trojan: Threats and emerging solutions, *in* 'IEEE High Level Design Validation and Test Workshop', pp. 166 –171.

Collins, R. R. (1998), 'The Pentium F00F Bug'. accessed at http://www.rcollins.org/ddj/May98/F00 FBug.html, 19 October 2012.

Dunkels, A. (2007), 'uBASIC - A really tiny BASIC interpreter'. accessed at http://www.sics.se/~adam /ubasic, 24 November 2011.

Jin, Y., Kupp, N. & Makris, Y. (2009), Experiences in hardware trojan design and implementation, *in* 'Hardware-Oriented Security and Trust,

2009. HOST '09. IEEE International Workshop on', pp. 50 –57.

King, S. T., Tucek, J., Cozzie, A., Grier, C., Jiang, W. & Zhou, Y. (2008), Designing and implementing malicious hardware, *in* 'Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats', USENIX Association, Berkeley, CA, USA, pp. 5:1–5:8.

Kranenburg, T. & van Leuken, R. (2010), Mb-lite: A robust, light-weight soft-core implementation of the microblaze architecture, *in* 'Design, Automation Test in Europe Conference Exhibition (DATE), 2010', pp. 997 –1000.

Lin, L., Burleson, W. & Paar, C. (2009), MOLES: Malicious Off-Chip Leakage Enabled by Side-Channels, *in* 'Proceedings of the 2009 International Conference on Computer-Aided Design', ICCAD '09, ACM, New York, NY, USA, pp. 117–122.

Lin, L., Kasper, M., Paar, C. & Burleson, W. (2009), Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering, *in* 'In Cryptographic Hardware and Embedded Systems - CHES 2009, volume 5747 of LNCS', Springer, pp. 382–395.

McIntyre, D., Wolff, F., Papachristou, C. & Bhunia, S. (2010), Trustworthy Computing in a Multi-Core System Using Distributed Scheduling, *in* 'On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International', pp. 211 –213.

Rajendran, J., Gavas, E., Jimenez, J., Padman, V. & Karri, R. (2010), Towards a comprehensive and systematic classification of hardware trojans, *in* 'Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on', pp. 1871 – 1874.

Reis, G., Chang, J., Vachharajani, N., Rangan, R. & August, D. (2005), SWIFT: Software Implemented Fault Tolerance, *in* 'Code Generation and Optimization, 2005. CGO 2005. International Symposium on', pp. 243 – 254.

Saxena, N. & McCluskey, E. (1998), Dependable Adaptive Computing Systems- The ROAR Project, *in* 'Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on', Vol. 3, pp. 2172 –2177 vol.3.

Tsang, R. (2009), Cyberthreats, Vulnerabilities and Attacks on SCADA Networks. University of California, Goldman School of Public Policy, working paper, accessed 19 December 2011, http://gspp.berkeley.edu/iths/Tsang_SCADA%20 Attacks.pdf.

Waksman, A. & Sethumadhavan, S. (2011), Silencing Hardware Backdoors, *in* 'Proceedings of the 32nd IEEE Symposium on Security and Privacy, May 2011'.

Weidner, K. (2000), 'fbvnc - a framebuffer-based VNC client'. accessed at http://pocketworkstation.org/ fbvnc.html, 15 December 2011.

Yeh, Y. (1996), Triple-Triple Redundant 777 Primary Flight Computer, *in* 'Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE', Vol. 1, pp. 293 –307 vol.1.

Young, D. (2011), 'COTS technologies ready for UAV deployment', *Military Embedded Systems* **7**, 12.

Zylin Consulting (2008), 'Zylin CPU'. http://opensource.zylin.com/zpu.htm.

Computational Complexity for Uniform Orientation Steiner Tree Problems

M. Brazil[1] **M. Zachariasen**[2]

[1] Department of Electrical and Electronic Engineering,
The University of Melbourne,
Victoria 3010, Australia,
Email: brazil@unimelb.edu.au

[2] Department of Computer Science,
University of Copenhagen,
DK-2100 Copenhagen Ø, Denmark,
Email: martinz@diku.dk

Abstract

We present a straighforward proof that the uniform orientation Steiner tree problem, also known as the λ-geometry Steiner tree problem, is NP-hard whenever the number of orientations, λ, is a multiple of 3. We also briefly outline how this result can be generalised to every $\lambda > 2$.

Keywords: Steiner tree problem, λ-geometry, computational complexity, NP-hard.

1 Introduction

Given a set of points N and set of $\lambda \geq 2$ uniformly distributed (legal) orientations in the plane, we consider the problem of constructing a minimum-length tree that interconnects N with the restriction that the tree is composed of line segments using legal orientations only. The focus of this paper is mainly on the case where λ is a multiple of 3, however we will also discuss the problem for the more general $\lambda \geq 2$ case. This so-called *uniform orientation* (or λ-geometry) Steiner tree problem is equivalent to computing a minimum Steiner tree for N under a metric where the unit circle is a regular 2λ sided polygon. In the Steiner problem the interconnection network may contain nodes other than the points in N. Computing the optimal locations of these nodes and the topology of the network makes this a computationally challenging problem.

The uniform orientation Steiner tree problem has important applications in micro-chip design, where millions of nets need to be routed on a (small) number of chip layers. On each routing layer, all wires generally use the same orientation in order to make joint routing of multiple nets feasible. In optimising the routing, the design of each net is usually treated as a planar geometric optimisation problem in λ-geometry, where the cost of transition between layers is treated as negligable. Today, most chip design technologies use only two perpendicular routing orientations (the so-called Manhattan routing where $\lambda = 2$), but the increasing number of available routing layers has made the use of multiple orientations relevant in practice (Chen et al. 2005, Teig 2002).

One of the great challenges in the design of integrated circuits for micro-chips is the continuing increase in density of these circuits, where the number of transistors on a chip tends to double approximately every two years (an observation known as Moore's law). This means that it is essential not only to devise optimisation algorithms that allow the nets to be designed as compactly as possible, but also to understand the computational complexity of such algorithms, since the scaling of these problems is a major issue. It is this question of computational complexity that this paper addresses.

It is well-known that the λ-geometry Steiner tree problem is NP-hard for the rectilinear metric ($\lambda = 2$) (Garey & Johnson 1977) and the Euclidean metric ($\lambda \to \infty$) (Garey, Graham & Johnson 1977). More recently, an NP-hardness proof was given for the λ-geometry Steiner tree problem for $\lambda = 4$ (Müller-Hannemann et al. 2007); this proof adapts the proof for the Euclidean case to the $\lambda = 4$ case.

Rubinstein et al. (1997) have given an elegant proof of the NP-hardness of a special case of the Euclidean Steiner tree problem — where the terminals are restricted to lying on two parallel lines. This approach was adapted by Brazil et al. (1998) to show that the gradient constrained Steiner tree problem is NP-hard, and the arguments have been simplified and further generalised in a later paper (Brazil et al. 2000).

Our Contribution.

We show that a method similar to that pioneered by Rubinstein et al. (1997) can be applied to the λ-geometry Steiner tree problem, to show that the λ-geometry Steiner tree problem is NP-hard whenever λ is a multiple of 3. We also briefly outline the generalisation of this result to every $\lambda > 2$. All of these NP-hardness results are new, apart from the result for $\lambda = 4$ (the octilinear norm), and even in that case the proof is significantly simpler than the very technical proof given by Müller-Hannemann et al. (2007).

Organisation of the Paper.

In Section 2 we summarise a number of structural results for Euclidean and λ-geometry Steiner trees that are relevant for the NP-hardness proof. The main result follows in Section 3, with a focus on the case where λ is a multiple of 3. We conclude with a brief discussion of generalisations of these results.

2 Preliminaries

The well-known Euclidean Steiner tree problem is defined (as a decision problem) as follows:

EUCLIDEAN STEINER TREE PROBLEM
Instance: A finite set of points N lying in the Euclidean plane and a positive integer L.
Question: Is there a tree T interconnecting the set N such that the length of T is at most L?

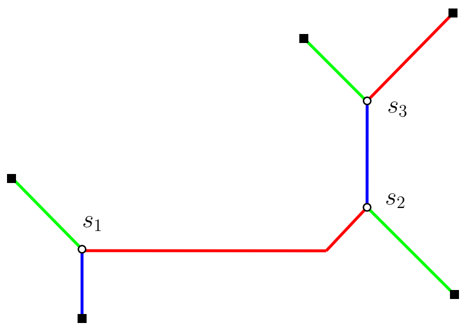


Figure 1: Full Steiner tree for five terminals and $\lambda = 4$. Black vertices are terminals and white vertices (s_1, s_2 and s_2) are Steiner points. The tree has six straight edges and one bent edge (s_1, s_2). Edges are colored by their colors in the corresponding direction set.

By the *length* of a tree T, we mean the sum of the lengths of the edges of T. A tree T with length L satisfying the Euclidean Steiner tree problem, where L is as small as possible for a given set N, is called a minimum Steiner tree for N. The given points N are called the *terminals* in T, and other vertices of T (of degree at least 3) are called *Steiner points*.

Let $\lambda \geq 2$ be a given positive integer. Given λ orientations $j\omega$ $(j = 1, 2, ..., \lambda)$ in the Euclidean plane, where $\omega = \pi/\lambda$ is a unit angle, we represent these orientations by the angles with the x-axis of corresponding straight lines. A line or line segment with one of these orientations is said to be in a *legal* direction. Objects composed of line segments in legal directions are said to belong to a λ-geometry.

For any given λ-geometry we define the following Steiner tree problem:

λ-GEOMETRY STEINER TREE PROBLEM
Instance: A finite set of points N in the Euclidean plane and a positive integer L.
Question: Is there a λ-geometry Steiner tree T with terminal set N such that the length of T is at most L?

Again, a minimum length tree T satisfying this problem for a given set N is known as a λ-geometry minimum Steiner tree for N.

The main part of the NP-hardness proof makes use of some key properties of Euclidean Steiner tree problem (Gilbert & Pollak 1968) and λ-geometry Steiner tree problem (Brazil et al. 2006, 2009). These properties are summarised below.

2.1 Direction Sets in λ-Geometry

Consider a λ-Geometry minimum Steiner tree T for a given set N. The Steiner points in T necessarily each have degree 3 or 4. In our NP-hardness proof only Steiner points that have degree 3 are relevant; degree 4 Steiner points (which only exist in very restricted cases) cannot occur as part of the minimum Steiner trees for the instances we construct.

Edges in T consist of line segments that use either a single legal orientation (*straight* edge) or two neighbouring legal orientations (*bent* edge); in the latter case we may assume that the edge consists of exactly two line segments (separated by angle $\omega = \pi/\lambda$ and called half-edges) that meet at a corner point (Figure 1).

Consider the set of legal orientations of the line segments of the edges that are adjacent to some Steiner point s of degree 3 in T; more precisely, consider each line segment as being oriented away from s, and let D be the corresponding set of directions. The set D is denoted a *direction set* if it is maximal under inclusion, i.e., there exists no minimum Steiner tree with some Steiner point that has a set of directions that is a superset of D. Local optimality conditions at Steiner point imply that direction sets can be characterized precisely (Brazil et al. 2006, 2009) (see Figure 2); when λ is a multiple of 3, the direction set has 6 directions and otherwise it has has 4 directions. The first pair of directions forms the so-called red edge, and the other directions are part of the remaining green and blue edges. The red, green and blue edges are separated by angles that are as close to 120° as possible (Figure 1). The total number of possible direction sets is 2λ — one for each pair of possible assignment of neighbouring red directions.

λ	Directions	
$3m$	Red:	$0,\ \omega$
	Green:	$2m\omega,\ (2m+1)\omega$
	Blue:	$4m\omega,\ (4m+1)\omega$
$3m+1$	Red:	$0,\ \omega$
	Green:	$(2m+1)\omega$
	Blue:	$(4m+2)\omega$
$3m+2$	Red:	$0,\ \omega$
	Green:	$(2m+2)\omega$
	Blue:	$(4m+3)\omega$

Figure 2: Feasible directions in a direction set (up to rotation by a multiple of ω).

A minimum Steiner tree can be decomposed into components in which every terminal is a leaf, known as *full components*, or *full minimum Steiner trees*. This decomposition is unique for a given minimum Steiner tree, but is not unique for a given terminal set. A minimum Steiner tree is said to be *fulsome* if it has the maximum possible number of full components for the given terminal set. Hence, a minimum Steiner tree is full and fulsome if there is no minimum Steiner tree on the same set of terminals with two or more full components.

Our interest in direction sets stems from the fact that all Steiner points in a full minimum Steiner tree use the *same* direction set; more precisely, we have the following theorem (Brazil et al. 2006) — a generalisation of this theorem to general weighted fixed orientation metrics has been given by Brazil et al. (2009):

Theorem 2.1 *(Brazil et al. 2006) Given a fulsome full minimum Steiner tree in λ-geometry, there exists a* single *direction set that is used by* every *Steiner point in the tree (where direction sets that can be obtained from each other by reflecting all directions through the Steiner point are considered to be equivalent).*

2.2 Zero-Shifts in λ-Geometry

A consequence of Theorem 2.1 is that the edges in a full minimum Steiner tree can be colored red, green and blue in such a way that all edges with the same colour use the same orientations (either a single orientation or two neighbouring orientations). Let e be a straight edge or half-edge in a full minimum Steiner tree T, oriented in direction $j\omega$. Then e is said to be *primary* if $(j-1)\omega$ is not a feasible direction with the same colour as e. Similarly, e is said to be *secondary* if $(j+1)\omega$ is not a feasible direction with the same colour as e. If $\lambda \neq 3m$ then it is possible for an edge to be both primary and secondary. We say that e is *exclusively primary* (or *exclusively secondary*) if it is primary, but not secondary (or, respectively, secondary, but not primary).

A minimum Steiner tree T is usually not unique in λ-geometry since the metric is not strictly convex. We define

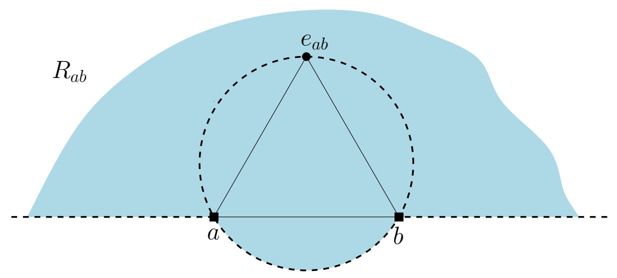


Figure 3: One of the two possible regions R_{ab} for two given points a and b. The other region is obtained by reflecting the diagram through the line through a and b.

a *zero-shift* as a perturbation of one or more Steiner points in T such that the perturbation does not increase the length of T. The identification of primary/secondary edges plays an important role for zero-shifts:

Theorem 2.2 *(Brazil et al. 2009) Let e_1 and e_2 be two edges in a full and fulsome minimum Steiner tree T such that e_1 has an exclusively secondary component and e_2 has an exclusively primary component. Then there exists a zero-shift acting on the Steiner points on the path from e_1 to e_2 in T, such that the shift can continue to be performed until either e_1 has no exclusively secondary component or e_2 has no exclusively primary component. Furthermore, this shift preserves the direction of all straight edges except (possibly) e_1 and e_2.*

A straightforward corollary of this theorem is that if, for a given set of terminals N, there exists a full and fulsome minimum Steiner tree, then there exists a minimum Steiner tree for N that has *at most one bent edge*.

2.3 Empty Regions for Euclidean Minimum Steiner Trees

Given two distinct points a and b in the Euclidean plane, let e_{ab} be the third vertex of an equilateral triangle with vertices a and b, let C_{ab} be the open finite region bounded by the circumcircle of $\triangle abe_{ab}$, and let R_{ab} be the union of C_{ab} and the open half plane defined by the line through a and b and containing e_{ab}, as illustrated in Figure 3. Note that R_{ab} is not uniquely defined; there are two possibilities for e_{ab} resulting in two possible choices for the region R_{ab}.

Proposition 2.3 *Let a and b be terminals of a Euclidean minimum Steiner tree T. If there exists a region R_{ab}, as defined above, containing no terminals of T then that region also contains no Steiner points of T.*

Proof. This is a simple extension of the "wedge property" introduced and proved by Gilbert & Pollak (1968). The wedge property states that any open wedge-shaped region having an angle of $2\pi/3$ and containing no terminals of T also contains no Steiner point of T. Region R_{ab} is an infinite union of such wedges, all with a and b on their boundary. ∎

3 NP-Hardness of the λ-Geometry Steiner Tree Problem

In this section we prove that the λ-geometry minimum Steiner tree problem is NP-complete for the cases where $\lambda = 3m$. We establish this result, in Corollary 3.2 below, by first proving a strictly stronger theorem, namely that the following class of problems is NP-complete for $\lambda = 3m$.

PARALLEL LINES λ-GEOMETRY STEINER TREE PROBLEM
Instance: A finite set of points N lying on two parallel lines in the Euclidean plane and a positive integer L.
Question: Is there a λ-geometry Steiner tree T with terminal set N such that the length of T is at most L?

In order to avoid issues related to the theoretical complexity of computing with exact real arithmetic, we in fact consider a *discretised* version of the problem as described later; in the construction below we initially ignore this technical difficulty.

We will show that for any given integer λ which is a multiple of 3 the PARALLEL LINES λ-GEOMETRY STEINER TREE PROBLEM is NP-complete. The main idea is to show that the problem can be used to polynomially encode an instance of the SUBSET SUM PROBLEM, which is well-known to be NP-complete (Garey & Johnson 1979):

SUBSET SUM PROBLEM
Instance: A set $S = \{d_1, \ldots, d_n\}$ of integers and an integer d.
Question: Is there a set $J \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in J} d_i = d$?

The main result is as follows. In the proof we let $\mathbf{d}(a, b)$ represent the Euclidean distance between the points or parallel lines a and b.

Theorem 3.1 *The parallel lines λ-geometry Steiner tree problem is NP-complete for any given $\lambda = 3m$ (where m is a positive integer).*

Proof. Let $S = \{d_1, \ldots, d_n\}$ and $d < \sum_{i=1}^{n} d_i := D$ be a given instance of the SUBSET SUM PROBLEM. We first show how to use this instance to construct (in polynomial time) an instance of the PARALLEL LINES λ-GEOMETRY STEINER TREE PROBLEM, and then show that the instance for the SUBSET SUM PROBLEM is a "yes" instance if and only if the corresponding instance for the PARALLEL LINES λ-GEOMETRY STEINER TREE PROBLEM is a "yes" instance. The statement of the theorem then follows.

The construction of the instance for the PARALLEL LINES λ-GEOMETRY STEINER TREE PROBLEM is as follows. We describe the construction in four stages:

1: Let V_1, V_1', V_2', V_2 be four vertical lines ordered from left to right such that

$$\mathbf{d}(V_1, V_2) \gg \mathbf{d}(V_1, V_1') = \mathbf{d}(V_2', V_2) \gg D. \qquad (1)$$

Let u_0 be a fixed point on V_2, and construct a zigzag path P between u_0 and a point on V_1 (labelled v), such that: P is composed of line segments with alternating polar angles $2\pi/3$ and $\pi/3$; P has $2n$ internal vertices (where n is the cardinality of S); and these internal vertices lie alternatively on V_1' and V_2'. See Figure 4.

2: Now from each internal vertex of P on V_1' extend a horizontal line segment to a point on V_1. Label these n points x_1 to x_n in ascending order. Similarly, from each internal vertex of P on V_2' extend a horizontal line segment to a point on V_2, and label these points u_1 to u_n in ascending order. Again, this is illustrated in Figure 4. This results in a λ-geometry Steiner tree interconnecting u_0, the u_i's, x_i's and v (where in each case i runs from 1 to n). We call this tree the *base tree* T_x.

3: The next stage of the construction is to replace each point x_i by three points on V_1 labeled, from bottom to top, a_i, b_i and c_i, satisfying: $|a_ib_i| = d_i$; $|b_ic_i| = d_i$; and x_i is the midpoint of a_ib_i (Figure 5). We also alter the Steiner tree constructed in Stage 2, so that

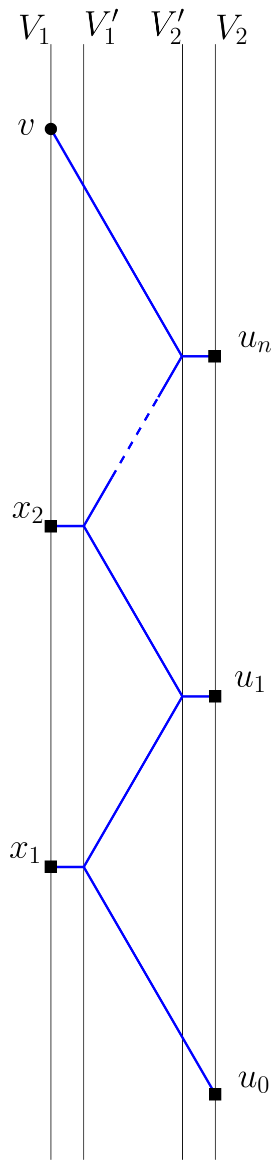


Figure 4: Construction for the case $\lambda = 3m$. The initial two stages of the construction result in the tree shown, the *base tree* T_x.

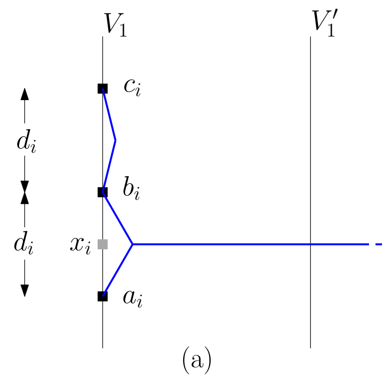
it connects to a_i, b_i and c_i, instead of x_i. This is done by shortening the horizontal edge by $d_i/(2\sqrt{3})$ on the left and creating a Steiner point at that new left endpoint with two new incident edges with polar angles $2\pi/3$ and $\pi/3$ and each with length $d_i/\sqrt{3}$ connecting to a_i and b_i. Finally we connect b_i to c_i with a single (geodesic) edge in λ-geometry, which is a vertical line segment (if m is even) or a bent edge using the two legal directions closest to vertical (if m is odd). This is illustrated in Figure 5(a), for the case where m is odd. Let N_v be the set consisting of u_0, the u_i's, a_i's, b_i's, c_i's and v. We denote the above λ-geometry Steiner tree (interconnecting the elements of N_v) by T_v. We will refer to the topology of the base tree T_x (from Stage 2) as the *base topology* of T_v.

Before completing the construction, we establish the following claim:

Claim 1. T_x and T_v are each the unique minimum Steiner λ-tree for their respective terminal sets.

Proof of Claim 1. Given the differences in scale in Inequality (1), consider the limiting case where $\mathbf{d}(V_1, V_1') = \mathbf{d}(V_2', V_2) = 0$. In that case each of T_x and T_v becomes a single zigzag path with polar angles $2\pi/3$ and $\pi/3$ between terminals $\{x_1, \ldots, x_n, v\}$ on V_1 and $\{u_0, u_1, \ldots u_n\}$ on V_2. The fact that this path is a Euclidean minimum Steiner tree (and hence a minimum Steiner λ-tree) on its vertices follows from Proposition 2.3 by constructing suitable regions: $R_{u_i u_{i+1}}$ for each $i \in \{0, \ldots, n-1\}$; $R_{x_i x_{i+1}}$ for each $i \in \{1, \ldots, n-1\}$; and $R_{x_n v}$. Taking the union of these regions, it is clear that any Steiner points must coincide with terminals, hence the minimum Steiner tree coincides with the minimum spanning tree. Furthermore, this minimum spanning tree is easily seen to be unique.

The result now follows immediately by continuity, and the fact that T_x and T_v (in the non-limiting case) are each locally minimal at every Steiner point. ∎



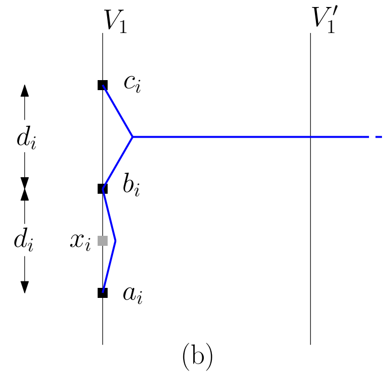


Figure 5: Stage 3 construction for the case $\lambda = 3m$. Diagram (a) shows how T_V connects to each triple a_i, b_i, c_i for the case where m is odd. Diagram (b) is the alternative connection possible in the tree T_0, used in Claim 2.

Note that it is straightforward to compute the total Euclidean length of T_v (i.e., $|T_v|$) in terms of $\mathbf{d}(V_1', V_2')$, $\mathbf{d}(V_1, V_1')$ and S. Let $L_v := |T_v|$. Also, we observe that the main full component of T_v, containing all the Steiner points, uses only three legal directions (and hence has no bent edges). We describe such a tree as a 3-*direction Steiner tree*.

The final stage of our initial construction is as follows.

4: Let v_0 be the point on V_1 below v such that $|v_0 v| = 2d$. Let N_0 be the set N_v where v has been replaced by v_0. Let T_0 be a minimum Steiner λ-tree for N_0. In other words, we can think of T_0 as being the new minimum Steiner tree obtained from T_v by moving the terminal v vertically downwards by $2d$.

We next establish some properties of the tree T_0.

Claim 2. The minimum Steiner λ-tree T_0 has the same base topology as T_v. Furthermore, for each triple, a_i, b_i and c_i, the main full component of T_0 either connects directly to a_i and b_i only, as in Figure 5(a), or to b_i and c_i only, as in Figure 5(b).

Proof of Claim 2. The first statement follows by the relative scale of the distances involved in Inequality (1), using the same argument as in the proof of Claim 1. For the second statement, it is an easy exercise to show that the configurations shown in Figure 5(a) and (b) are the only locally minimal ways of connecting the main full component of T_0 to a_i, b_i and c_i. ∎

Claim 3. The following three statements are equivalent;

1. The answer to the given instance of the SUBSET SUM PROBLEM is "yes".
2. There exists a 3-direction minimum Steiner λ-tree on N_0 with the same base topology as T_v.
3. There exists a Steiner λ-tree on N_0 with length at most $L_v - \sqrt{3}d$.

Proof of Claim 3. The equivalence of the three statements is shown in four steps.

Step 1: (1) ⇒ (2). Let T_x be the minimum Steiner λ-tree constructed in Stage 2 of the main construction. Suppose we treat v and one of the terminals x_i as 'moveable' points, able to move along V_1. Then consider the following question: If we move x_i vertically upwards by a distance δ, how does the position of v on V_1 change so that T_x remains a 3-direction tree? As Figure 6 shows, each horizontal edge incident with a terminal u_j (for j such that $i \le j \le n$) increases in length by $2\delta/\sqrt{3}$. In particular, the horizontal edge incident with u_n increases in length by $2\delta/\sqrt{3}$ which implies that v moves downwards by 2δ.

We now apply a similar argument to T_v. Again, allow v to be a 'moveable' point, and consider the effect of changing the connection of the tree at one of the triples a_i, b_i, c_i (from the original connection as shown in Figure 5(a) to the alternative connection shown in Figure 5(b)) while keeping the tree a 3-direction Steiner tree. By the symmetry of the two connection types this is equivalent in its effect on v to moving x_i upwards by d_i in T_x; that is, v moves downwards by $2d_i$. This effect is additive across all of the triples, meaning that if we change to the alternative connection scheme at each $i \in J$ where $J \subseteq \{1, \ldots, n\}$ is a set solving the given instance of the SUBSET SUM PROBLEM, then v moves downwards by $2d$ to v_0, giving the required 3-direction minimum Steiner λ-tree on N_0.

Step 2: (2) ⇒ (1). The argument here is similar to that in Step 1. This time we begin with a 3-direction minimum Steiner λ-tree on N_0 with the same base topology as T_v, and treat the terminal v_0 as being a 'moveable' point on V_1. Since $v_0 \neq v$ it follows that there must be at least one $i \in \{1, \ldots, n\}$ such that the connection of the tree to a_i, b_i, c_i uses the alternative connection scheme shown in Figure 5(b). Let $J' \subseteq \{1, \ldots, n\}$ be the set of all such i where this alternative connection scheme is used. If for any $i \in J'$ we change to the original connection scheme (as shown in Figure 5(a)) while keeping the tree as a 3-direction tree then, by the same argument as in Step 1, v_0 moves upwards by $2d_i$. Now if for every $i \in J'$ we change to the original connection scheme while keeping the tree as a 3-direction tree then it is clear that v_0 now coincides with v (since the position of v_0 is uniquely determined by the positions of the other terminals, the topology of the tree and the three directions). Since $\mathbf{d}(v_0, v) = 2d$ it follows that $\sum_{i \in J'} d_i = d$, and hence J' gives a 'yes' solution to the given instance of the SUBSET SUM PROBLEM.

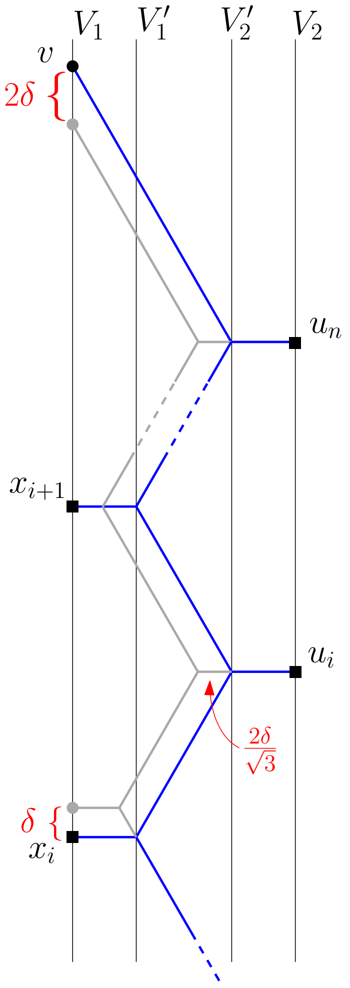


Figure 6: Construction for proof of Claim 3 (Step 1).

Step 3: (2) ⇒ (3). We first analyse the change in length to T_x under the movement of x_i by δ described in Step 1 and illustrated in Figure 6. For the horizontal edges: the edge incident with x_i decreases in length by $\delta/\sqrt{3}$; each edge incident with x_j for $i+1 \le j \le n$ decreases in length by $2\delta/\sqrt{3}$; and each edge incident with u_j for $i \le j \le n$ increases in length by $2\delta/\sqrt{3}$. Hence the total length of the horizontal edges increases by $\delta/\sqrt{3}$. For the main zigzag path: its height decreases by 2δ and hence its length decreases by $4\delta/\sqrt{3}$. Together, these result in an overall decrease in length of $3\delta/\sqrt{3} = \sqrt{3}\delta$ for the whole tree.

It follows for the tree T_v that if we treat v as a 'moveable' point, and consider the effect of changing to the alternative connection of the tree at one of the triples a_i, b_i, c_i, while keeping the tree a 3-direction Steiner tree, the tree decreases in length by $\sqrt{3}d_i$. Hence, by additivity, the 3-direction minimum Steiner λ-tree on N_0 has length $L_v - \sqrt{3}d$.

Step 4: ¬(2) ⇒ ¬(3). To prove this last statement, we argue as follows: Suppose we have a Steiner tree T_0 on N_0 with the same base topology as T_v, but which is not a 3-direction tree. Colour all edges containing a horizontal component red. As described in Section 2.1, we can assume that there is only one bent edge; furthermore, the bent edge is the red edge connecting to the triple a_i, b_i, c_i. Now, suppose we replace this bent red edge by the orthogonal projection of this edge onto the line extending the horizontal component of the edge. It is easy to see, by the same argument as in Step 3, that the length of the resulting (disconnected) 3-direction λ-network is again $L_v - \sqrt{3}d$. The tree T_0 has length strictly longer than this, giving the required conclusion.

Discretisation and scaling. Above we have presented a transformation of any instance of the SUBSET SUM PROBLEM to show that the parallel lines λ-geometry Steiner tree problem is NP-hard if one ignores arithmetic precision issues. Furthermore, by the straightforward constructive nature of this transformation it is clear that the problem belongs to NP, and hence is NP-complete, up to issues of artihmetic precision. Here we demonstrate that the result remains true when applying a discretisation and scaling that resolves the issues related to computing with exact real arithmetic. A similar discretisation and scaling step has been described in detail in a number of previous papers (Brazil et al. 2000, Garey, Graham & Johnson 1977, Rubinstein et al. 1997), and so will only be sketched here.

In the discretisised problem Euclidean distances are rounded up to the nearest integer. Also, it is assumed that terminals and Steiner points can only have integer coordinates. Thus for a given Steiner tree T, performing discretisation increases or decreases the length of every edge by at most 3. Since all trees considered have at most $7n + 1$ edges, every tree is at most length $3 \cdot (7n + 1)$ longer or shorter than before the discretisation.

We need to be able to distinguish between 'yes' and 'no' instances in the discretisised problem. More precisely, as shown in the proof of Claim 3 above, we need to be able to distinguish between 3-direction minimum Steiner trees and non 3-direction minimum Steiner trees. The last part of the proof of Claim 3 shows that non 3-direction minimum Steiner trees have a length that is at least $\epsilon_\lambda = (1 - \cos\omega)/(2\sin\omega)$ above $L_v - \sqrt{3}d$, the length of 3-direction minimum Steiner trees.

The problem is now scaled by multiplying all terminal coordinates by an integer K. One can distinguish between 'yes' and 'no' instances, if $K\epsilon_\lambda - 2 \cdot 3 \cdot (7n + 1) \geq 1$. Choosing $K \geq (42n + 7)/\epsilon_\lambda$ suffices, and results in a polynomial scaling. ∎

Finally, since every instance of the parallel lines λ-geometry Steiner tree problem is also an instance of the λ-geometry Steiner tree problem, we immediately get the following corollary.

Corollary 3.2 *The λ-geometry Steiner tree problem is NP-complete for any given $\lambda = 3m$ (where m is a positive integer).*

4 Conclusion and Generalisations

The proof of Theorem 3.1 in the previous section relies, to a large extent, on the properties of the base tree T_x constructed in the course of the proof. A key property of the base tree is that if we perturb a single terminal x_i up or down along V_1 the resulting minimum Steiner tree on the new terminal set is strictly longer than T_x. If x_i is perturbed downwards (away from v) then the only change to the tree T_x is that the edge incident with x_i becomes a bent edge via the introduction of a new secondary direction; all other edges in the tree are straight primary edges. On the other hand, if x_i is perturbed upwards (towards v) then the edge incident with x_i again becomes a bent edge but this time via the introduction of a new primary direction; all other edges in the tree are straight secondary edges. This is possible due to the symmetry in the direction set for $\lambda = 3m$, which means that in a 3-direction Steiner tree such as T_x it is ambiguous as to whether the edges are all primary or all secondary (see Figure 2).

The difficulty in generalising Theorem 3.1 to other values of λ lies in the fact that the direction sets no longer exhibit this symmetry when $\lambda \neq 3m$. If we construct a base tree for one of these other values of λ (as in the proof of Theorem 3.1) using primary directions (as in the table in Figure 2) then it is no longer true that perturbing x_i in either direction along V_1 always reduces the length of the Steiner tree; for one of the two directions an edge other than the edge incident with x_i will become bent (i.e., the colour labeling changes), and it can be shown that the new minimum Steiner tree that results is shorter than the original base tree.

This problem, however, can be successfully circumvented via a slight alteration to the construction. If instead of choosing the lines V_1, V_1', V_2', V_2 to be vertical, we choose them to have a polar slope of $\pi/2 - \pi/(3\lambda)$, then it is possible to show that any perturbation of x_i along V_1 reduces the length of the corresponding base tree. The details of this somewhat technical argument will appear in a forthcoming paper (Brazil et al. 2012). The conclusion is that the parallel lines λ-geometry Steiner tree problem is NP-complete for all $\lambda > 2$.

Acknowledgements

Our thanks to the Research Institute for Discrete Mathematics, University of Bonn, where both authors were visitors during the initial writing of this paper.

References

M. Brazil, D. A. Thomas & J. F. Weng (1998), Gradient constrained minimal Steiner trees. In *Network Design: Connectivity and Facilities Location (DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 40)*. American Mathematical Society.

M. Brazil, D. A. Thomas & J. F. Weng (2000), On the complexity of the Steiner problem. *Journal of Combinatorial Optimization*, 4:187–195.

M. Brazil, D. A. Thomas, J. F. Weng & M. Zachariasen (2006), Canonical Forms and Algorithms for Steiner Trees in Uniform Orientation Metrics. *Algorithmica*, 44:281–300.

M. Brazil & M. Zachariasen (2009), Steiner Trees for Fixed Orientation Metrics. *Journal of Global Optimization*, 43:141–169, 2009.

M. Brazil & M. Zachariasen (2012), Computational Complexity for Fixed Orientation Steiner Trees. in preparation, 2012.

H. Chen, C. K. Cheng, A. B. Kahng, I. I. Mandoiu, Q. Wang & B. Yao (2005), The Y-architecture for on-chip interconnect: Analysis and methodology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24:588–599, 2005.

M. R. Garey, R. L. Graham & D. S. Johnson (1977), The complexity of computing Steiner minimal trees. *SIAM Journal on Applied Mathematics*, 32(4):835–859, 1977.

M. R. Garey & D. S. Johnson (1977), The rectilinear Steiner tree problem is NP-complete. *SIAM Journal on Applied Mathematics*, 32:826–834, 1977.

M. R. Garey & D. S. Johnson (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco, CA, 1979.

E. N. Gilbert & H. O. Pollak (1968), Steiner Minimal Trees. *SIAM Journal on Applied Mathematics*, 16(1):1–29, 1968.

M. Müller-Hannemann, A. Schulze & L. Zhang (2007), Hardness and Approximation of Octilinear Steiner Trees. *International Journal of Computational Geometry and Applications*, 17:231 – 260, 2007.

J. H. Rubinstein, D. A. Thomas & N. C. Wormald (1997), Steiner trees for terminals constrained to curves. *SIAM Jorunal on Discrete Mathematics*, 10(1):1–17, 1997.

S. Teig (2002), The X Architecture: Not your father's diagonal wiring. In *International Workshop on System-Level Interconnect Prediction (SLIP)*, pages 33–37. ACM, 2002.

A Hybrid Image-Based Modelling Algorithm

Hoang Minh Nguyen
The University of Auckland, New Zealand
hngu039@aucklanduni.ac.nz

Burkhard Wünsche
The University of Auckland, New Zealand
burkhard@cs.auckland.ac.nz

Patrice Delmas
The University of Auckland, New Zealand
p.delmas@cs.auckland.ac.nz

Christof Lutteroth
The University of Auckland, New Zealand
lutteroth@cs.auckland.ac.nz

Abstract

This paper explores the practical aspects associated with visual-geometric reconstruction of a complex 3D scene from a sequence of unconstrained and uncalibrated 2D images. These image sequences can be acquired by a video camera or a handheld digital camera without the need for camera calibration. Once supplied with the input images, our system automatically processes and produces a 3D model. We propose a novel approach, which integrates uncalibrated Structure from Motion (SfM), shape-from-silhouette and shape-from-correspondence, to create a quasi-dense scene geometry of the observed scene. In the second stage, surface and texture are applied onto the generated scene geometry to produce the final 3D model. The advantage of combining silhouette-based and correspondence-based reconstruction approaches is that the new hybrid system is able to deal with both featureless objects and objects with concaved regions. These classes of objects usually pose great difficulty for shape-from-correspondence and shape-from-silhouette approach. As the result, our approach is capable of producing satisfactory results for a large class of objects. Our approach does not require any a priori information about camera and image acquisition parameters. We tested our algorithm using a variety of datasets of objects with different scales, textures and shapes acquired under different lighting conditions. The results indicate that our algorithm is stable and enables inexperienced users to easily create complex 3D content using a standard consumer level camera.

Keywords: image-based modelling, correspondence-based reconstruction, silhouette-based reconstruction

1 Introduction

There is an increasing amount of applications that require high quality 3D representations, e.g. for arts, commerce, virtual heritage, training, education, computer games, virtual environments, documentation, exchanging information, and social networking applications. Conventionally, 3D digital models are constructed using modelling tools such as Maya, 3D Max or Blender. Although these applications enable graphic designers to construct highly realistic and complex 3D models, they have a steep learning curve and often require a considerable amount of training and artistic skills to use. These restrictions render them unsuitable for non-professionals. The introduction of specialised hardware, such as laser scanners, has simplified the creation of models from real physical objects. However, while many of these systems deliver highly accurate results, they are usually extremely costly and often have restrictions on the size and surface properties of objects in the scene. Consequently, there is a critical need to improve on the status quo by making 3D content creation available to a wider group of users.

In recent years image-based modelling has emerged as a new approach to simplify 3D content creation. In contrast to traditional geometry-based modelling and hardware-heavy approaches, the image-based modelling techniques confront the formidable, and still unanswered, challenge of creating a comprehensive representation of 3D structure and appearance of a scene from visual information encoded in 2D still images. Image-based modelling techniques are usually less accurate, but offer very intuitive and low-cost methods for recreating 3D scenes and models.

The ultimate goal of our work is to create a low-cost system that allows users to obtain 3D reconstruction of the observed scene using a consumer-grade camera. The idea behind the system is very simple: Once supplied with the input images, our system will automatically process and produce a 3D model without any a priori information about the scene to be reconstructed.

Reconstructing 3D scenes from a collection of 2D photographic images requires knowing where each photo was taken. A common approach to obtain this information is to perform camera calibration manually. However, this method requires a setup and preparations that are usually too sophisticated for inexperienced users. Furthermore, this method places a restriction on the types of scenes that can be reconstructed since it is not always feasible to perform camera calibration for a large and complex scene. For this reason, for each input image, our algorithm needs to automatically estimate the intrinsic and extrinsic parameters of the camera being used and compute the 3D coordinates of a sparse set of points in the scene (the scene geometry or point cloud). Surfaces and texture are then applied to this point cloud to produce the final model.

Due to the sparseness of the scene geometry, problems such as surface artifacts, noise, and blurry textures might arise during the surface and texture reconstruction processes. Most previous works approached these problems by constraining the types of objects and requiring manual user inputs to aid their systems in deducing the structure of the object to be reconstructed. However, these requirements breach our goal of creating an easy-to-use system and providing the capability of reconstructing any type of object. We overcome these problems by using a hybrid approach that integrates shape-from-correspondence and shape-from-silhouette methods. The system will perform 3D reconstruction using the following steps:

1. Camera parameter estimation
2. Initial point cloud generation from extracted key features
3. Increase the density of the point cloud by exploiting silhouette information
4. Reconstruct the object's surface and texture to produce the final model

The remainder of this paper is organised as follows. After a description of the related work on image-based modelling, a discussion about the algorithms used in our system is presented in section 3. Results are discussed in section 4. Section 5 concludes and summarises the paper and gives a brief outlook on directions for future research.

2 Related Work

Although there has been much interest and study of 3D modelling techniques over the last few decades, robustly and automatically obtaining 3D models is still a difficult task. For the past 30 years, creation of artiticial 3D models using conventional graphics and animation software such as Maya and 3D Max has continued to be the most popular approach. To this end, various tools have been proposed to assist the human designer by using images of the object to be modelled [THP08]. The reason why manual creation of 3D models remains the prevalent approach despite intensive study and research is that, in fact, there is no computer vision or graphics technique that works for every object. The difficulty in selecting the most suitable 3D reconstruction technique that works for a large class of objects justifies the abundant literature that exists on this subject [Est04, REH06].

Various multiple view reconstruction techniques have been explored in recent years. Amongst them, the most well-known and successful class of techniques have been *shape from silhouette* and *shape from correspondence*.

The shape from silhouette class of algorithms exploits silhouette information to create intersected visual cones, which are subsequently used to derive the 3D structure of an object. Shape from silhouette-based methods are popular for shape estimation due to their good approximation qualities for a large number of objects, their stability with respect to the object's material properties and textures, as well as their ease and speed of implementation. Exploiting silhouette information for 3D reconstruction was first considered by Baumgart in 1974. In his pioneering work [Bau74], the author computed polyhedral shape approximations of a baby doll and a toy horse by intersecting silhouette cones. Following Baumgart's work, many different variations of the shape from silhouette paradigm have been studied and proposed.

Aggarwal *et al.* [MA83] proposed a method that used an intensity threshold-based segmentation method to separate the object foreground and background in each input image. A connected component analysis of the segmented image produces the silhouette. In order to compute the intersection of different silhouette cones, the authors used a run-length encoded, uniformly discretised volume. The idea behind this is to consider each image point as a statistical occupancy sensor. The point observations are then analysed to deduce where matter is located in the observed scene. This is achieved by discretising the scene into three dimensional voxels, which are then projected into silhouette images. The task is to mark the individual voxels as either "in" or "out". If the projection of a voxel belongs to the foreground in all silhouette images, then the voxel is labeled as "in" otherwise it is labeled as "out". If a voxel is labeled as "out", it is excluded from further computations of the object structure.

Matusik *et al.* [MBR$^+$00] improve the efficency of shape from silhouette techniques by taking advantages of epipolar geometry. In their method, 3D silhouette-intersection computation is reduced to 2D by projecting one silhoutte onto another. The intersection is then carried out in image space. Franco *et al.* [FB10] attempted to improve the effciency of Matusik's method and to produce a water-tight model. Their method involves two main steps. First, point clouds of the observed scene are generated by back-projecting viewing edges of the visual hull. Next, missing surface points are recovered by exploiting local orientation and connectivity rules. A final connection walkthrough is then carried out to construct the planar contours for each face of the polyhedron.

In recent years, various correspondence-based techniques have been explored. However, most of these methods were designed to tackle reconstruction problems related to a particular class of objects. As a result, their use is often limited.

Fruh *et al.* [FZ03] used a combination of aerial imagery, ground colour, and LIDAR scan data to create textured 3D models of an entire city. While the proposed method produces visually acceptable results, it suffers from a number of drawbacks that render it impractical for consumer-level applications. In particular, the method requires intensive use of special hardware during the data acquisition step. This includes a vehicle equipped with fast 2D laser scanners and a digital camera to acquire texture data for an entire city at the ground level and a LIDAR optical remote sensor. Additionally, the required manual selection of features and the correspondence in different views is very tedious, error-prone, and cannot be scaled up well.

Xiao *et al.* [XFT$^+$08] presented a semi-automatic image-based approach to recover 3D structure of façade models from a sequence of street view images. The method combines a systematic and automatic decomposition scheme of façades for analysis and reconstruction. The decomposition is accomplished by recursively splitting the complete façades into small segments, while still preserving the overall architectural structure. Users are required to provide feedback on façade partitioning. This method demonstrated excellent results.

Quan *et al.* [QTZ$^+$06] presented a method for modelling plants. In their method, segmentation is performed in both image space (by manually selecting areas in input images) and in 3D space. Using the segmented images and 3D data, the geometry of each leaf is recovered by setting a deformable leaf model. Users are also required to provide hints on segmentation. The main disadvantage of this method is that it requires full coverage of the observed model (360 degree capture), which may not always be possible in practice due to obstructions and space limitations. Branches are modelled through a simple user interface.

Tan *et al.* [TZW$^+$06, LQ02] introduced a method for creating 3D models of natural-looking trees from a collection of images. Due to the large leaf count, small image footprint and widespread occlusions, it is not possible to recover accurate geometric representation for each leaf. In order to overcome this problem, the authors populate the manufactured tree with leaf replicas from segmented input images to reconstruct the overall tree shape.

3 Algorithms

Our proposed approach uses a coarse-to-fine strategy where a rough model is first reconstructed and then sequentially refined through a series of steps. The approach consists of two main stages: scene geometry extraction and visualisation. Both of these stages are achieved by dividing the main problem into a number of more manageable subproblems, which can then be solved by separate modules. The entire reconstruction process is illustrated in Figure 1.

The objective of the first stage is to recover the scene geometry from the input images. This stage begins with the camera parameters for each view being estimated. This is accomplished by the automatic extraction of distinctive features and establishment of point correspondences in stereo image pairs. We then isolate all matching images, selecting those that view a common subset of 3D points. Given a set of matching images, a scene geometry (point cloud) and camera pose can be estimated simultaneously by Structure from Motion and subsequently refined by Bundle Adjustment. Next, additional points are added to the computed scene geometry by exploiting the silhouette information to produce a more complete geometry. In the final step, a resampling technique is applied onto the point clouds to produce the final scene geometry. In contrast to an initial version of this algorithm [NWDL11] we integrate silhouette information, which is used in the point cloud generation and the surface reconstruction.

In the second stage, we tackle the problem of how to transform the scene geometry recovered in the preceding stage into a realistic representation of the scene. This is accomplished by applying surfaces and texture to the resulting scene geometry. The outcome of this stage is a complete 3D representation of the observed scene.

3.1 Camera Parameter Estimation

One key challenge in extracting 3D representation of a scene from a sequence of 2D images is that the process requires knowing where each photo was taken and in what direction the camera was

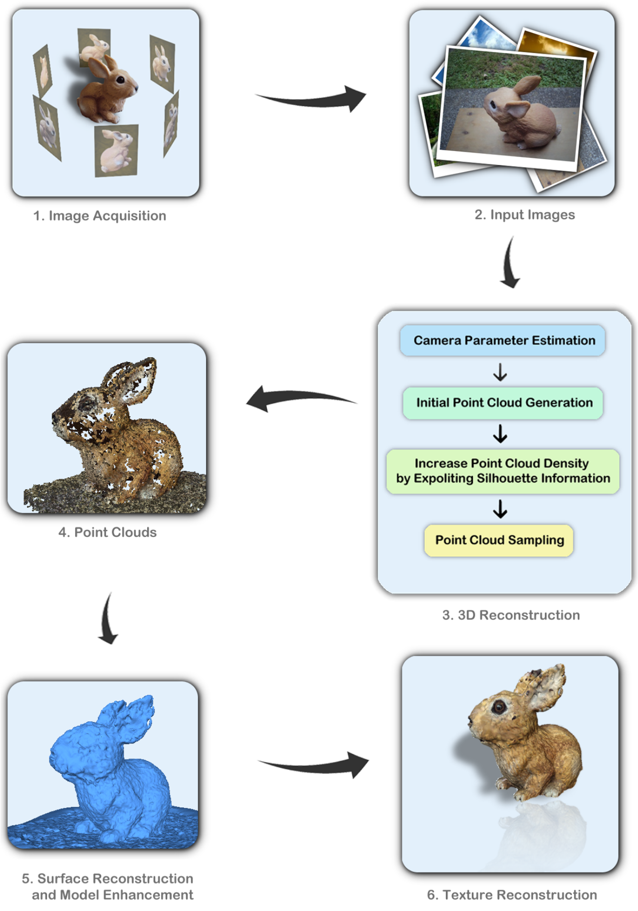

1. Image Acquisition
2. Input Images
Camera Parameter Estimation
Initial Point Cloud Generation
Increase Point Cloud Density by Expoliting Silhouette Information
Point Cloud Sampling
3. 3D Reconstruction
4. Point Clouds
5. Surface Reconstruction and Model Enhancement
6. Texture Reconstruction

Figure 1: Overview of our algorithm for reconstructing 3D models from a set of unconstrained and uncalibrated images.

pointed (extrinsic parameters), as well as the internal camera settings, such as zoom and focus (intrinsic parameters), which influence how incoming light is projected onto the retinal plane.

In order to recover such information, the system will first detect and extract points of interest such as corners (edges with gradients in multiple directions) in the input images. This is accomplished using the SIFT feature detector [Low06]. Feature points extracted by SIFT are highly distinctive, and invariant to different transformations and changes in illumination, and additionally have a high information content [HL07, BL05]. Once features have been identified and extracted from all the images, they are matched. This is known as the correspondence problem. Given a feature in an image I_1, what is the corresponding feature (the projection of the same 3D feature) in the other image I_2. This can be solved by using a Euclidean distance function to compare the two feature descriptors. All the detected features in I_2 will be tested and the one with minimum distance is selected [HQZH08].

Once all interest points have been found, we match them across views and estimate the camera parameters and 3D coordinates of the matched points simultaneously using the Structure from Motion (SfM) technique. Structure from Motion designates the computation of the camera poses and scene geometry (3D points) simultaneously from a set of images and their feature correspondences. More precisely, SfM can be formulated as an optimisation problem, where the goal is to determine the configuration of cameras and 3D points that, when related through the equations of perspective projection, best agree with the detected feature correspondences. This computation is carried out by exploiting a constraint between correspondences and the physical configuration of the two cameras. This is a powerful constraint, as two 3D rays chosen at random are very unlikely to pass close to one another. Given enough point correspondences between two images, the geometry of the system is sufficiently constrained to determine the camera poses (up to scale). The key to the success of this procedure is that successive images must not vary significantly, i.e. must contain overlapping visual features.

Our solution takes an incremental approach, in which a pair of images is selected to initialise the sequence. This initial pair should have a large number of matches, but must also have a large baseline. This is to ensure that the 3D coordinates of observed points are well-conditioned. The remaining images are added to the optimisation one at a time ordered by the number of matches [REH06, SSS06]. The Bundle Adjustment technique is followed to refine and improve the obtained solution. The accuracy of the reconstruction depends critically on the final step. Figure 2 demonstrates a scene geometry created using Structure from Motion and Bundle Adjustment.

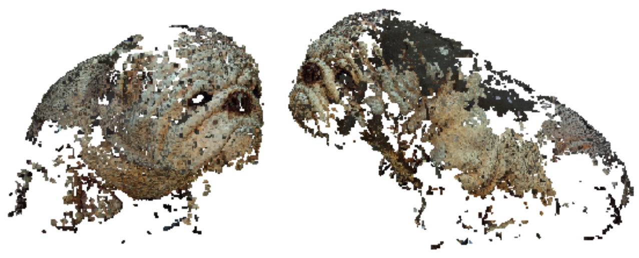


Figure 2: Two views of the scene geometry generated from 37 input images.

3.2 Scene Geometry Enhancement

At this stage, we have successfully acquired both camera parameters and scene geometry. As the scene geometry is derived from distintive features of the input images, they are often sparse. In order to produce a more complete and comprehensive model, the obtained scene geometry needs to be enhanced. This can be accomplished by exploiting the silhouette information of the observed scene to generate additional 3D points.

The process is as follows: First, silhouette information of the observed scene in each view is extracted using the Marching Squares algorithm [Lor95] producing sets of silhouette points. Each of these sets represents an exhaustive point-by-point isoline list of every pixel which constitutes a silhouette contour. To define a silhouette using an enormous contour point set will inevitably upsurge the computational expense. To avoid this, silhouette data must be preprocessed to reduce the number of silhouette contour points. We achieve this by first performing a *Delaunay triangulation* of the contour points. The output triangular mesh framework is then fed to a mesh simpification algorithm [Mel98] to decrease the number of triangles, which in turn effectively reduces the number of contour points.

Each set of contour points together with the camera centre of that view defines a *viewing cone*, whose apex is located at the camera's optical centre. Each viewing cone consists of a number of *cone lines.* A cone line represents a 3D line formed by a silhouette contour point and the camera's optical centre. The polyhedral visual hull information can be obtained by calculating the intersection of these viewing cones. However, 3D polygon intersection is non-trivial and often computationally expensive. Matusik *et al.* [MBR+00] proved that equivalent results can be obtained by the following steps:

Assume that we want to compute the intersection of silhoutte A and B.

1. Projecting each cone line of the viewing cone A onto the silhoutte B.
2. Calculate the intersection of the projected line and the silhouette B in 2D.
3. Lifting the computed intersection points from 2D to 3D yields a set of 3D points, which defines a face of the polyhedral visual hull.

Projecting a cone line onto another silhouette As each of these rays (cone lines) is defined by the camera's optical centre and the projecting ray $\overrightarrow{\mathbf{O}_A S_i}$, the projection of a ray onto the image B is computed as follows:

Projecting the camera's optical centre of A onto B (*epipole of* B)

$$\widetilde{e_B} = \mathtt{P}_B \begin{bmatrix} \mathbf{O}_A \\ 1 \end{bmatrix} \tag{1}$$

where $\mathtt{P}_B = [Q_B, \ q_B]$ is the 3×4 projection matrix of B.

Projecting the point D_i at the infinity of the projecting ray $\overrightarrow{\mathbf{O}_A S_i}$ onto B

$$\widetilde{d_B} = \mathtt{P}_B \begin{bmatrix} \mathbf{D}_A \\ 0 \end{bmatrix} = \mathtt{P}_B \begin{bmatrix} \mathtt{Q}_A^{-1} S_i \\ 0 \end{bmatrix} \tag{2}$$

These two points define a 2D line on image B, which will subsequently be used to compute intersections.

Line-Silhouette Intersection The naive approach of computing intersections of a line and a silhouette requires the traversal of each pair of silhouette contour points and performing a line-line intersection with the given line. This method, however, proves to be very inefficient. We use an alternative method [15], which is based on the observation that all projected rays intersect at one common point, the epipole. This makes it possible to subdivide the reference image into partitions so that each projected ray will intersect all the edges and only the edges in that partition. This way we only have to traverse edges in a particular partition when computing intersections. The algorithm to partition the reference image into regions is as follows:

First, the epipole of the reference image B is computed by projecting the camera's optical centre of A onto B. Each silhouette contour point along with the epipole forms a line segment. The slopes of all the line segments are stored in a sorted list in ascending order. Two consecutive line segments in the list create a partition or a bin. Each bin is defined by a minimum and maximum slope value (α_{min} and α_{max}) and a set of edges associated with it.

To determine the set of edges allocated to each bin, we traverse through the silhoutte contour points while maintaining a list of edges in the current bin. When a contour point is examined, we eliminate from the current bin all edges ending at that contour point and append all edges starting at that contour point. A start of an edge is characterised as the edge end point that has a smaller slope value.

The intersections of a given line and a silhouette can be determined by first examining the slope of the input line segment to establish the bin to which the line segment belongs. Once the corresponding bin is found, we iterate through the edges assigned to that bin and perform line-to-line intersection. The result of this stage is a collection of 2D intersection points.

2D points to 3D points Given a 2D intersection point in the reference image, we want to compute the corresponding 3D point of this point. This is accomplished by shooting a ray from the camera's optical centre of the reference image through the intersection point and compute the intersection point of the newly created 3D rays with the original 3D ray. The intersection point of the two 3D ray is the 3D point we want to find. Figure 3 shows an example of the lifting process.

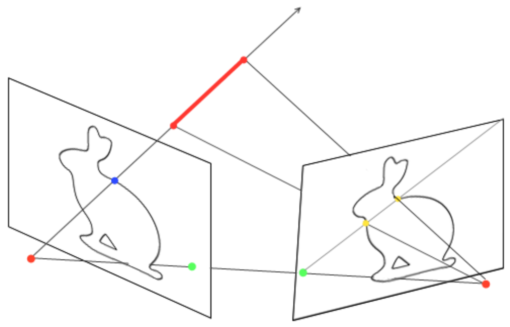


Figure 3: Lifting 2D points to 3D points.

A problem arises when the two rays do not intersect as a result of noise. We can handle this by finding the point with the smallest distance to the two rays using a *Least Square* method. Figure 4 illustrates the newly generated 3D points of the dog dataset. These point clouds together with those of the previous stage form a much more comprehensive geometric representation of the observed scene.

3.3 Surface and Texture Reconstruction

The final step is to reconstruct surfaces from the obtained point clouds. Surface reconstruction is the process of automated generation of a smooth surface that closely approximates the underlying 3D models from which the point clouds were sampled. The reconstructed surface is usually represented by a polygon mesh. Many sophisticated surface reconstructions have been proposed and extensively studied. In our system, we employ the Poisson surface reconstruction algorithm [KBH06] for remeshing the surface.

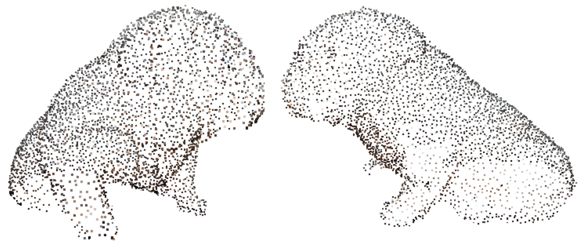


Figure 4: Additional 3D points generated by exploiting the silhouette information.

The Poisson reconstruction method [Bol10] extracts surfaces by taking advantage of the integral relationship between oriented points sampled from the surface of an unknown model and the indicator function χ of the model. The indicator function is defined as 1 at points that lie inside the model and as 0 for points that lie outside the model.

Formally, the problem can be formulated as: Given an oriented point set $P = \{s_1, s_2, ..., s_N\}$, where a point sample s_i consists of a position p and an inward facing normal n and is assumed to lie on or near the surface ∂M of an unknown model M. The aim is to create a smooth and watertight approximation to the original surface by computing the indicator function χ of the model and then extract an appropriate isosurface [Bol10].

This is accomplished by first deriving a relationship between an integral of the normal field over the surface and the gradient of the model's indicator function. The gradient of the indicator function is defined as a vector field that is zero almost everywhere as the indicator function is constant practically everywhere except at points near the surface, where it points in the direction of the inward surface normal. Hence, the input oriented point samples can be thought of as samples of the gradient of the indicator function [Bol10, Kaz05]. An example of Poisson Surface reconstruction is shown in Figure 5.

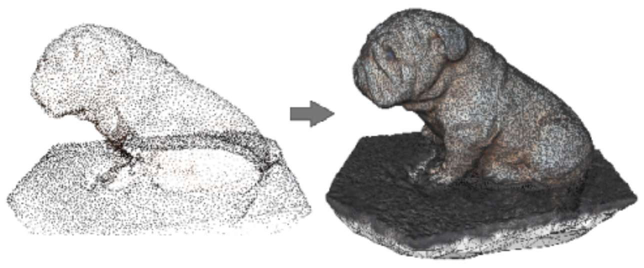


Figure 5: Poisson reconstruction. The input point samples are on the left, while the reconstructed mesh is shown on the right.

After the surface reconstruction phase, the resulting model is a shaded and textureless polygonal mesh. Such a representation is usually not an accurate reflection of the object in the input images. A more realistic representation is obtained by creating a surface texture from the colour information in the input images. In texture reconstruction, each vertex of the polygonal mesh has the RGB colour of the corresponding vertex in the attached point clouds. The resulting triangle mesh with vertex colours is rendered using Gouraud shading. Since the triangle mesh is very dense, the colour interpolation over triangles gives acceptable results.

4 Results

We evaluated our system using a wide range of data sets at different scales of both indoor and outdoor scenes. In most test cases, the system produces qualitatively good results. The geometries recovered using our system appear to retain high resemblance to that of the original models even for objects with smooth, uniform and limited-feature surfaces or concave regions. Datasets with both smooth, uniform surfaces and concave regions often resulted in an unsatisfied geometry since the silhouettes did not contain sufficient geometry information, and there were too few unambiguous features to allow full surface reconstruction. The size of our test datasets varies from as few as 6 images to hundreds of images, which were all taken with a simple handheld camera.

Owl Dataset The first dataset consists of 32 images (3648 × 2736 pixels) taken from arbitrary view directions using a normal consumer-level SONY DSC-W180 camera of a paper model of an Owl. Three of the 32 input images are shown in Figure 6. This Owl object is highly decorated with texture. On average, each image contains over 53,000 features, which would aid the reconstruction greatly.

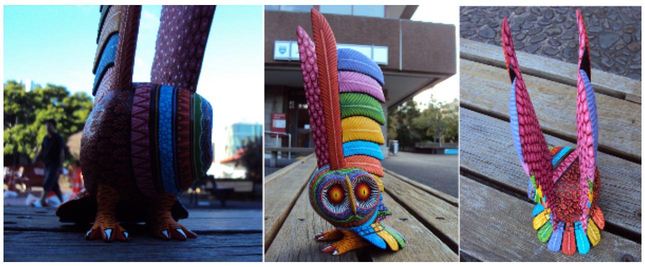


Figure 6: Owl dataset input images.

The resulting reconstructed model, illustrated in figure 7, is of excellent quality. The overall shape, along with details such as feathers, and the texture of the original model are reconstructed with no visual difference to the original model. This is due to the high number of distinct features of the original object. The resulting model has 678,210 faces in total. The process took approximately 3 hours and 30 minutes to complete on an Intel Quad Core i7 with 6GB RAM.

Lady Statue Dataset This dataset consists of 17 images taken from the frontside of a black copper statue. The backside was not accessible. The images were taken with the same camera as in the previous case and under very complex lighting conditions.

Some of the images which were used for the reconstruction are shown in Figure 8. The resolution of each image in this dataset is 3648 × 2736 pixels. Notice that the surface of the model contains few texture details.

The reconstructed model has 268,892 faces and is of moderate quality (Figure 9). This result is

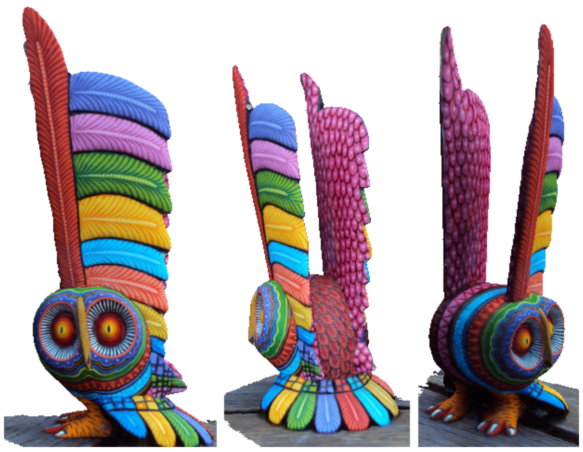


Figure 7: 3D reconstruction of the Owl model.

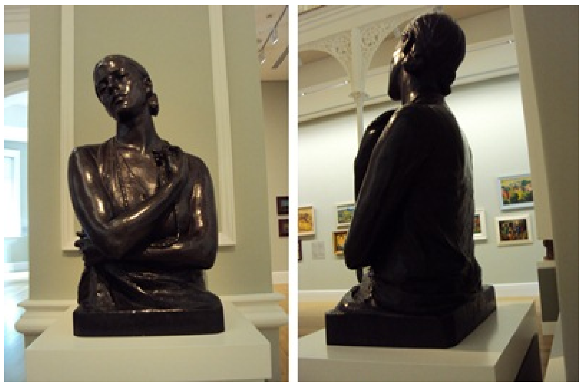


Figure 8: Two of seventeen input images of the Lady model taken at the Auckland Art Gallery.

surprisingly good considering the relatively low number of input images and the lack of distinct visual features for correspondence matching. The geometry was predominantly obtained from the silhouette information. This shows that our system is capable of producing good results for feature-poor models and few input images.

The reconstructed texture shows several significant differences to the original model. This is caused by changes in lighting conditions within the gallary. Some white patches also appear around the head region of the reconstructed model due to the lack of images from that particular direction. The current systems only computes surface colours, rather than material properties, and hence works best for diffuse surfaces and lighting conditions resulting in few isolated highlights. The reconstruction process required approximately 1 hour and 47 minutes on an Intel Quad Core i7 with 6GB RAM.

Bunny Dataset This data set comprises 49 images (2592 × 1944 pixels) taken from many different views of a bunny model using a normal consumer-level SONY DSC-W180 camera. The original model has a very bumpy surface, which is extremely difficult to reconstruct (Figure 10). The objective of this test is to determine if the system can effectively deal with rough surfaces and high illumination variations due to surface roughness and self shadowing.

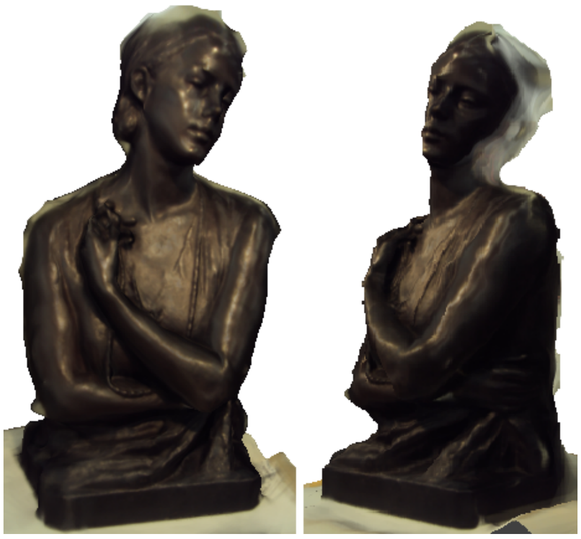


Figure 9: 3D reconstruction of the Lady Statue.

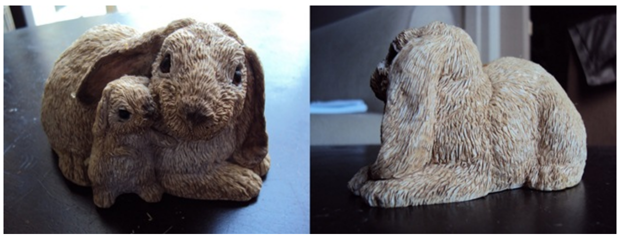


Figure 10: Two of 31 input images of a bunny model.

The reconstruction result (shown in Figure 11) is of very good quality. The final model, which is composed of 528,637 faces, bears a high resemblance to the original object.

This result demonstrates that our approach is able to recover realistic 3D models of shapes with complex surface geometries. The overall computation time is approximately 4 hours and 40 minutes on an Intel Quad Core i7 with 6GB RAM.

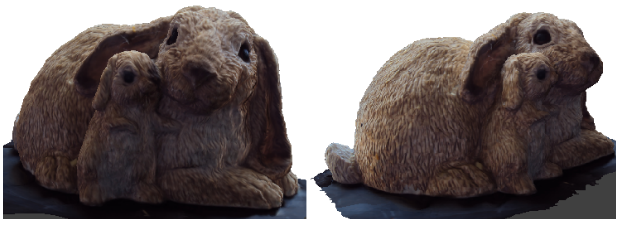


Figure 11: 3D Reconstructed model of the Bunny dataset.

Cat Dataset This data set was obtained from a ceramic statue of a cat. There are 44 images with a resolution of 2592 × 1944 pixels. This object has a very shiny, reflective, smooth and uniform surface. These surface characteristices pose a major problem for correspondence-based methods as very few features are available. Added to that, the observed object's surface also contains a number of concave regions, which also cause problems for silhouette-based methods. Three of the 44 input images are presented in Figure 12.

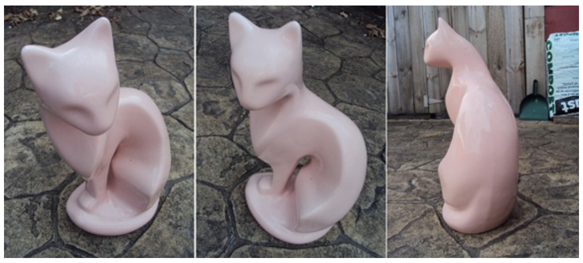


Figure 12: Input images of the Cat dataset.

The visual quality of the reconstruction is not satisfactory. While the object is recognisable, important features such as the eyes and ears are not reconstructed well, and the colour distribution varies from the original object.

This reconstruction results (Figure 13) demonstrate that our approach is able to recover 3D geometry even for models with shiny, reflective and uniform surfaces, but problems exist with concave regions. The overall computation time was approximately 2 hours and 55 minutes on an Intel Quad Core i7 with 6GB RAM.

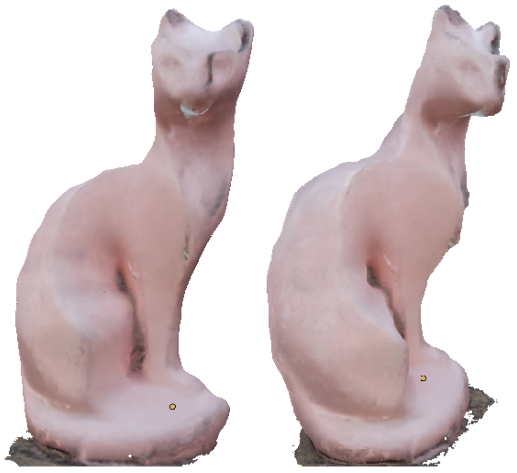


Figure 13: 3D reconstruction of the Cat model.

Comparison with Commercial Systems
Over the past 2-3 years a couple of prototypes of commercial systems have emerged, which are currently at various stages of user testing. While none of the commercial systems has published any information about the utilised approach, we performed an analysis, which suggest that most of them use predominantly a silhouette-based approach [NWDL12]. In this section we provide a short comparison of our novel hybrid approach with some of the most promising alternative image-based modelling systems (*123D Catch* and *Agisoft*) currently in development.

In order to evaluate these systems, we used a repository of over 40 objects. After initial tests using different objects we selected one object which reflected the main shortcomings of all tested algorithms. For this test, 44 input images of the above cat model were supplied to these systems. The reconstruction results from these two systems are shown in Figure 14 and 15.

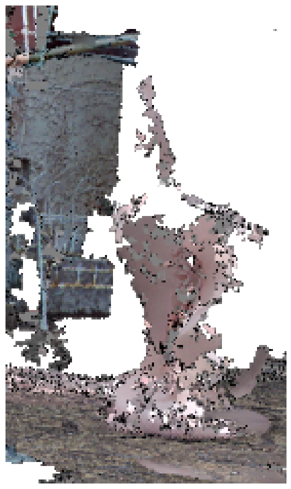


Figure 14: 3D reconstruction of the Cat model using *Agisoft* System.

Agisoft's model is almost unrecognisable. The reconstruction is very incomplete and retains no visual resemblance to the original object. Large portions of the object geometry are missing. This is mostly due to the fact that the system is not able to register views when there is only a limited number of features in the input images. Additionally, the reconstructed textures are also inadequate with many black patches covering the reconstructed surfaces.

123D Catch produces more a complete geometry of the reconstructed model. The resulting model still somewhat reflects the structure of the original object. Compared to our reconstruction result, the reconstructed geometry appears much rougher. Regions around the back, the head and the tail of the reconstructed model are mostly distorted.

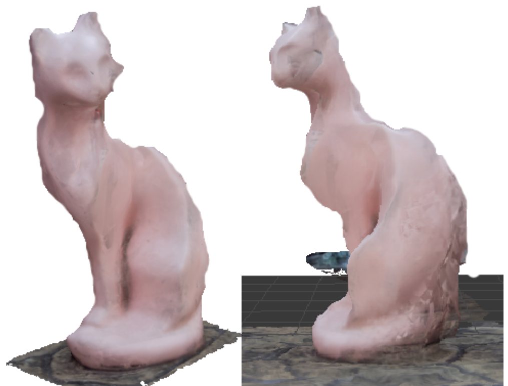


Figure 15: 3D reconstruction of the Cat model using the *123D Catch* System.

5 Conclusion and Future Work

Our research was motivated by the observation that there is an increasing demand for virtual 3D models. Existing modelling packages, such as Blender and Maya, enable the construction of realistic and complex 3D models, but have a steep learning curve and require a high level of artistic skills. Additionally, their use is very time-consuming and often involves tedious manual manipulation of meshes. The use of specialised hardware for 3D model reconstruction (laser scanners) makes it possible for inexperienced users to acquire 3D digital models. However, such hardware is very expensive and limited in its applications. Based on this, we concluded that the most promising approach for a general, affordable content creation process is to use an image-based modelling approach using images obtained with a consumer-level uncalibrated camera.

We demonstrated our system's ability to produce qualitatively good results for a wide range of objects including those with smooth, uniform, and textureless surfaces or containing concave regions. The system has also demonstrated its robustness in the case that there are illumination variations and shadows in the input images.

Problems, such as missing textures in some regions, still exist with the resulting 3D models. This is caused by insufficient input images of those regions. We aim to overcome this problem by employing image in-painting and exemplar-based texture synthesis techniques.

Although we have demonstrated that our system can create 3D content inexpensively and conveniently, the high computation cost partially offsets its advantages. For example, our system takes approximately 2 hours to process 17 images, and roughly 5 hours for 31 images on a Intel Quad Core i7 with 6GB RAM. The computational cost rises quadratically with the number of input images. For the current state-of-the-art of hardware technology, real time processing is impossible on consumer-level machines. However, we can reduce the processing time considerably by parallelising the computations as much as possible and executing them on the GPU.

References

[Bau74] Bruce Guenther Baumgart. *Geometric modeling for computer vision.* Doctoral Dissertation, Stanford University, 1974.

[BL05] Matthew Brown and David Lowe. Unsupervised 3D object recognition and reconstruction in unordered datasets. *In International Conference on 3D Digital Imaging and Modelling*, pages 56–63, 2005.

[Bol10] Matthew Grant Bolitho. *The Reconstruction of Large Three-Dimensional Meshes.* Doctoral Thesis. The Johns Hopkins University, 2010.

[Est04] Carlos Hernandez Esteban. *Geometric modeling for computer vision.* Doctoral Thesis, Ecole Nationale Superieure des Telecommunications, 2004.

[FB10] Jean Sebastien Franco and Edmond Boyer. Efficient polyhedralmodeling from silhouettes. *IEEE Transaction on Pattern Analysis and Machine Intelligences*, 31:853–861, 2010.

[FZ03] Cristian Fruh and Avideh Zakhor. Constructing 3d city models by merging ground-based and airborne views. *In IEEE International Conference on Computer Vision and Pattern Recognition*, 2:562–569, 2003.

[HL07] Shungang Hua and Ting Liu. Realistic 3D reconstruction from two uncalibrated views. *In International Journal of Computer Science and Network Security*, 7:178–183, 2007.

[HQZH08] Shaoxing Hu, Jingwei Qiao, Aiwu Zhang, and Qiaozhen Huang. 3d reconstruction from image sequence taken with a handheld camera. *The International Archives Of The Photogrammetry*, 37(91):559–563, 2008.

[Kaz05] Michael Kazhdan. Reconstruction of solid models from oriented point sets. *In SIGGRAPH*, pages 73–82, 2005.

[KBH06] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. *In Proceedings of the fourth Eurographics symposium on Geometry processing (Aire-la-Ville, Switzerland, Switzerland, 2006)*, pages 61–70, 2006.

[Lor95] William Lorensen. Marching through the visible man. *IEEE Visualization*, pages 368–373, 1995.

[Low06] David G Lowe. Distinctive image features from scale-invariant keypoints. *In International Journal of Computer Vision*, 60(91):110–2004, 2006.

[LQ02] Maxime Lhuillier and Long Quan. Quasi-dense reconstruction from image sequence. *European Conference on Computer Vision*, pages 125–139, 2002.

[MA83] Worthy Martin and J. K Aggarwal. Volumetric descriptions of objects from multiple views. *In IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(2):150–158, 1983.

[MBR$^+$00] Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven Gortler, and Leonard McMillan. Image-based visual hulls. *In Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 369–374, 2000.

[Mel98] Stan Melax. Simple, fast, and effective polygon reduction algorithm. *Game Developer Magazine*, pages 44–49, 1998.

[NWDL11] Minh Hoang Nguyen, Burkhard C Wunsche, Patrice Delmas, and Christof Lutteroth. Modelling of 3D objects using unconstrained and uncalibrated images taken with a handheld camera. *Computer Vision, Imaging and Computer Graphics - Theory and Applications*, 274:1–5, 2011.

[NWDL12] Minh Hoang Nguyen, Burkhard C Wunsche, Patrice Delmas, and Christof Lutteroth. 3D models from the black box: Investigating the current state of image-based modeling. *Communication Proceedings, Pilsen, Czech Republic, Union Agency*, pages 25–28, 2012.

[QTZ$^+$06] Long Quan, Ping Tan, Gang Zeng, Lu Yuan, JingdongWang, and Sing Bing Kang. Image-based plant modeling. *In ACM Transactions on Graphics*, 25(3):599–604, 2006.

[REH06] Fabio Remondino and Sabry El-Hakim. Image-based 3d modelling: A review. *The Photogrammetric Record*, 21:269–291, 2006.

[SSS06] Noah Snavely, Steven Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3D. *In ACM Transactions on Graphics*, 25(3):835–846, 2006.

[THP08] Thormahlen Thorsten and Seidel Hans-Peter. 3D-modeling by ortho-image generation from image sequences. *ACM Transaction Graph*, 27(3):1–5, 2008.

[TZW$^+$06] Ping Tan, Gang Zeng, Jingdong Wang, Sing Bing Kang, and Long Quan. Image-based tree modeling. *In ACM Transactions on Graphics*, 27(3):418–433, 2006.

[XFT$^+$08] Jianxiong Xiao, Tian Fang, Ping Tan, Peng Zhao, Eyal Ofek, and Long Quan. Image-based façade modeling. *In ACM Transactions on Graphics*, 27(5):26–34, 2008.

Improving Product Configuration in Software Product Line Engineering

Lei Tan **Yuqing Lin** **Huilin Ye** **Guoheng Zhang**

School of Electrical Engineering and Computer Science
University of Newcastle,
University Drive, Callaghan, NSW, 2308,
Email: {lei.tan, guoheng.zhang}@uon.edu.au,
{yuqing.lin, huilin.ye}@newcastle.edu.au

Abstract

Software Product Line Engineering (SPLE) is a emerging software reuse paradigm. SPLE focuses on systematic software reuse from requirement engineering to product derivation throughout the software development life-cycle. Feature model is one of the most important reusable assets which represents all design considerations of a software product line. Feature model will be used in the product configuration process to produce a software. The product configuration is a decision-making process, where all kinds of relationships among configurable features will be considered to select the desired features for the product. To improve the efficiency and quality of product configuration, we are proposing a new approach which aims at identifying a small set of key features. The product configuration should always start from this set of features since, based on the feature dependencies, the decisions made on these features will imply decisions on the rest of the features of the product line, thus reduce the features visited in the configuration process. We have also conducted some experiments to demonstrate how the proposed approach works and evaluate the efficiency of the approach.

Keywords: Software Product Line; Feature Model; Product Configuration; Minimum Vertex Cover.

1 Introduction

During the last decade, software product line (SPL) engineering has emerged as an effective software development methodology to promote systematic software reuse. An SPL is a collection of software products that share common characteristics as a family. The key idea of software product line engineering is to discover and exploit commonalities across a product family, thus to improve the reusability of various software engineering assets. A successful SPL based software development will improve the development productivity and the quality of software, and significantly reduce development cost and time-to-market.

This work is supported by Australian Research Council under Discovery Project DP0772799.

In addition to the commonalities shared by all the products in an SPL, individual products may somehow vary from each other. The variabilities among products in an SPL must be appropriately represented and managed. Feature oriented modelling approaches have been widely used in software product line engineering for this purpose. Features are prominent and distinctive system requirements or characteristics that are visible to various stakeholders in a product line (Lee et al. 2002). A feature model specifies the features, their relationships, and the constraints of feature selections for product configuration. A product in an SPL is defined by a unique valid combination of selected features. Product configuration is a process of selecting features for developing a product in an SPL.

A feature model is usually represented as a tree in which the variabilities of features are represented as variation points (VPs). A variation point (Pohl et al. 2005) consists of a parent feature, a group of child features, called variants, and a multiplicity specifying the minimum and maximum number of variants that can be selected from the variation point when configuring a product. The selection of variants at a variation point is not only constrained by the multiplicity but also by the dependencies between the variants at this variation point and the variants at other variation points. Dependencies are the constraints on configurations in a product line. The following two dependencies have been defined by Kang et al. (1990).

1. Requires: If a feature requires, or uses, another feature to fulfil its task, there is a Requires relationship between these two features.
2. Excludes: If a feature has conflicts with another feature, they cannot be chosen for the same product configuration, i.e. they mutually exclude each other. There is a bi-directional Excludes relationship between two features.

Many other types of dependencies have been considered as well, such as *Impact*, *Mandatory*, *Optional*, *Alternative* and *Or* (Benavides et al. 2010, Ye et al. 2008). It does not seem that there is an agreeable industry standard for the types of dependencies to be included in the feature model.

A valid feature model describes the configuration space of a system family (Czarnecki et al. 2005). During product configuration process, application engineers specify member products by selecting the desired features from a feature model based on customer requirements and constraints such as feature dependencies. However, the traditional product configuration becomes a time-consuming and error-prone task because of the large number of features and feature relationships. In literature, several approaches have been proposed to improve the traditional product configuration method. Czarnecki et al. (2005) propose staged configuration which allows incremental configuration of cardinality-based feature model by performing a step-wise specialization of feature models. White et al. (2009) provide automated support of staged configuration based on constraint satisfaction problem (CSP). Mendonca et al. (2008) develop a collaborative product configuration method which decomposes a feature model into several configuration spaces. Each type of stakeholder makes feature selection in a corresponding configuration space and finally the selected features in different configuration spaces are merged to get the final configuration. Loesch et al. (2007) simplify product configuration by reclassifying variable features based on their usage and by restructuring feature model to simplify variabilities. The above proposed approaches have improved product configuration process from different aspects.

In this paper, we propose an approach to improve the efficiency of product configuration. The idea of our approach is that, by taking into account of feature dependencies, it is possible to identify a small set of variation points from a feature model. Selecting variants from this variation point set implies the visiting of all the variation points in the feature model, thus we have reduced the number of variation points to visit during the configuration process. As a result, the number of decisions and rollbacks in the configuration process are significant reduced. We have not found any other works along the same line and we believe our method is an innovative approach.

The remainder of the paper is organized as follows. Section 2 and 3 define the basic concepts and propose our approach. Section 4 discusses the adapted simulated annealing algorithm that is the core of the proposed approach. In Section 5, we use an example to demonstrate how our approach works. In Section 6 and 7 we present experiments results which demonstrate the efficiency of our proposed approach. Section 8 concludes the paper and discusses future works.

2 Feature Model and Product Configuration

Feature Model is a key artifact of the software product line engineering. It tells the commonalities and differences among the member products. As we have already mentioned, various types of relationships among the features have been considered to be included in the feature model, many of them are not precisely defined. Indeed, the relationships among the features are complex and hard to describe.

For example, the "Requires" and "Excludes" relationships are simple and most understandable ones, however, the "Impact" relationship is somehow more complex. Feature A has impact on feature B could mean several things, e.g. the selection of feature A suggests in certain degree of selection of feature B, or the implementation of feature A depends on the implementation of feature B etc. So, the "Impact" relationship is defined less precisely and harder to deal with.

If we only consider the simple relationships, i.e. the relationships we could define precisely, then many mathematical approaches could be involved to improve the efficiency of software engineering process. Mannion (2002) uses propositional logic expressions to detect "void feature model" errors and Zhang et al. (2004) develop a propositional logic-based approach to verify partially customized feature models at any binding time. For detecting dead features and false optional features, Czarneck et al. (2005) transform a feature model into an CSP problem which includes a set of variables and a set of constraints over the variables and then uses CSP solvers to automate the identification process. Trinidad et al. (2008) further develop a CSP-based approach to explain the identified feature model errors. To improve the efficiency of CSP-based approaches, we have developed a constraint propagation-based method to identify and explain dead features and false variable features (Zhang et al. 2011). We can use the above mentioned approaches to obtain a valid feature model by detecting and correcting feature model errors and then perform product configuration in the valid feature model.

As we would like to improve the efficiency of product configuration by applying some mathematical approaches, we need to limit us to the dependencies which can be defined properly. In this paper, we only consider the "Requires" and "Excludes" dependencies. We also like to point out that our approach is extendable to cover other types of dependencies if they are defined accurately and the logical operations involving these dependencies are defined precisely.

When configuring a product, we usually need to go through the feature model and make a configuration decision at each variation point to select variant(s). Usually a depth-first traversal of a feature tree will be employed to make decision at each variation point. Assuming that there are two variation points, VP1 and VP2, we first encounter VP1 during the traversal and select a variant at VP1. If the selected variant at VP1 has "Requires" dependency with a variant at VP2, then the required variant at VP2 has to be selected based on the "Requires" dependency. Thus, we do not need to visit this variant at VP2 later. In this case a selection of variants at one variation point may already cover the selections at other variation points. If those variation points with greater coverage are processed first, obviously, there will be less number of decisions to make, thus the configuration process is more efficient. Another advantage of doing configuration this way is that we can reduce the mistakes made during configuration. For the above mentioned example, assume that we make configuration decision at VP2 first and mistakenly decide not to include this variant in the final product. We will not realize this is a wrong decision until we make configuration decision at VP1. In this case, we have to go back to VP2 again to correct the wrong selection made before. And in a worse situation, the corrections might propagate, thus could be time consuming to fix. Thus, in terms of configuration efficiency and quality, it is better to visit VP1 first in the configuration

The sequence of the variation points following which we make our configuration decisions has significant impact on the efficiency of product configuration. Getting the correct sequence is the key idea of our approach.

3 The Proposed Approach

As discussed in the precede section, the sequence of variation points follow which we select the variants is important for improving the efficiency of product configuration. The sequence of the variation points can be determined based on a parameter which we call *Configuration Coverage*. Configuration Coverage (CC) of a variation point refers to what extent a configuration decision made at a variation point covers the configuration decisions of the remainder variation points in a feature model. To improve the efficiency of configuration process, it is crucial to identify a minimum set of variation points, where the decisions made at this set of variation points cover the decisions to be made at all the variation points in a feature model. This set of variation points will be sorted based on their configuration coverage and we will start making configuration decision at the variation point with the biggest configuration coverage. However, as the feature model could be very complex, and the feature dependency model could be hard to trace, it is not always straightforward to find such a minimum set of variation points for product configuration. We propose to employ some well studied mathematical techniques to help identify the minimum variation point set from a feature model. Before we present the proposed approach, we first define some measurements used in the approach.

As mentioned, each variation point consists of a set of variants and a multiplicity. For a variant v, we define two measurements, one is called the $Positive\ Coverage\ PC(v)$, another one is called the $Negative\ Coverage\ NC(v)$. When the variant v is included in a product configuration, the positive coverage of variant v ($PC(v)$) is a set of variable features which will be automatically included or excluded based on their dependent relationships with variant v. Similarly, when the variant v is excluded in a product configuration, the negative coverage of variant v ($NC(v)$) is a set of variable features which will be automatically included or excluded based on the multiplicity constraint between these variants. To work out the positive coverage and negative coverage, we need to examine the dependency relationships among the variants in a feature model. For example, if variant v requires variant w, then we know w is in $PC(v)$, furthermore, if w requires variant u, then u is also in $PC(v)$ since if v is included in the final product, then u will be included as well. If variant t requires variant v, then we know t is in $NC(v)$, since if v is not included in the final product, then t can not be included as well.

For a variation point in a feature model, the multiplication rule restricts the selection of the variants associated with the variation point. For example, a multiplicity of $1..n$ means only up to n variants can be selected in the final product. For all the variants associated with a variation point, we call a subset of variants a *valid selection* if it obeys the multiplicity. The *complement* of a valid selection is the set of variants that are not included in the selection at the variation point. When a certain valid selection has been made at a variation point, the configuration coverage of the selection is the union of all the positive coverage of the variants in the valid selection and all the negative coverage of the variants in the complement of the selection. Below is an example to illustrate how do we calculate these parameters. Note a variation point may have different configuration coverage when different valid selections are made at the variation point.

Included in Fig. 1 is a fraction of a feature model. There are four variants $v1$, $v2$, $v3$ and $v4$ associated with the variation point VP, and the multiplicity restricts the selection, i.e. only up to two variants can be included in the final product. From the dependency relationship Fig. 2, we know that:

$PC(v1) = \{u1, u3\}$, $NC(v1) = \emptyset$, $PC(v2) = \{u2, u4, u7, u8\}$, $NC(v2) = \emptyset$, $PC(v3) = \{u5\}$, $NC(v3) = \{u6\}$, $PC(v4) = \{u4, u8\}$ and $NC(v4) = \{u6\}$.

All possible selections based on the multiplicity at the VP are listed below,

$v1$, $v2$, $v3$, $v4$, $v1 \cup v2$, $v1 \cup v3$, $v1 \cup v4$, $v2 \cup v3$,

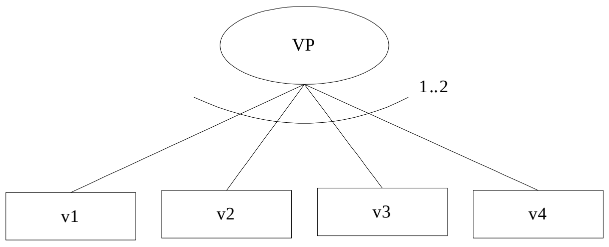


Figure 1: A Variation Point (VP) and its Variants.

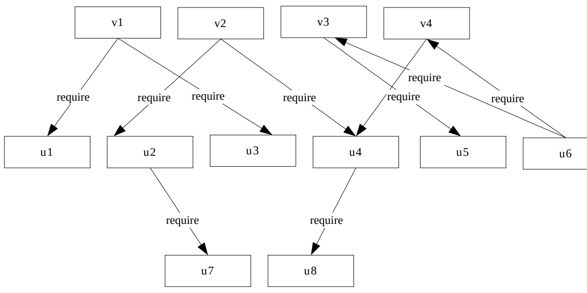


Figure 2: The Dependencies among Variants.

$v2 \cup v4$, $v3 \cup v4$.

The configuration coverage (CC) of each selection is listed as following:

$$CC(v1) = PC(v1) \cup NC(v2) \cup NC(v3) \cup NC(v4)$$
$$= \{u1, u3, u6\},$$
$$CC(v2) = \{u2, u4, u7, u8, u6\},$$
$$CC(v3) = \{u5, u6\},\ CC(v4) = \{u4, u8, u6\},$$
$$CC(v1 \cup v2) = PC(v1) \cup PC(v2) \cup NC(v3) \cup NC(v4)$$
$$= \{u1, u2, u3, u4, u7, u8, u6\},$$
$$CC(v1 \cup v3) = \{u1, u3, u5, u6\},$$
$$CC(v1 \cup v4) = \{u1, u3, u4, u6\},$$
$$CC(v2 \cup v3) = \{u2, u4, u7, u8, u5, u6\},$$
$$CC(v2 \cup v4) = \{u2, u4, u7, u8, u6\},$$
$$CC(v3 \cup v4) = \{u5, u6\}.$$

The maximum from the above sets is the CC when $v1$ and $v2$ are selected at this variation point, we call this $Maximum\ Configuration\ Coverage$ ($MAXCC$). The Configuration Coverage of a variation point is the $MAXCC$ of all the valid selections at the variation point. If a selection of variants at a variation point can result in $MAXCC$, we call the selection a $Max\ Coverage\ Selection$ (MCS). For the above example, $MCS = v1 \cup v2$, $MAXCC = CC$ ($v1 \cup v2$)$= \{u1, u2, u3, u4, u7, u8, u6\}$.

The $MAXCC$ of a variation point indicates how much a decision made at the variation point covers the decisions at the other variation points of a feature model. The bigger the coverage of a variation point, the (potentially) more variant features will be included/excluded as the result of including or excluding the variation point, therefore, it is more important to visit the variation point earlier in the configuration process. Using the $MAXCC$ of variation points, we can construct a small set of variation points from a feature model, the union of whose $MAXCC$ will cover all the variant features in the feature model. Software engineers should start with this set of variation points when configuring a product from the feature model. This set of variation points represents the key decisions for configuring a member product. Focusing on this set of variation points will reduce the configuration effort.

One thing we would like to point out is that, in the configuration process, the $MAXCC$ of a variation point might not correspond to the CC of the actual selection. i.e. the actual selection might not give the $MAXCC$. The $MAXCC$ of a variation point only indicates the potential importance of the variation point in the configuration process.

Our approach works like this, we firstly work out the $MAXCC$ of every variation point in the feature model and then calculate the smallest set of variation points that covers the whole feature model. In other words, the union of the $MAXCC$ of this set of variation points includes all the variants of the feature model. The software engineers could start from this set of variation points to configure the final product. Once a decision is made on a variation point (or a set of variation points), we then work out the coverage of the selection(s), this is a straightforward task as we already know the positive coverage and negative coverage of each variable feature. For the rest of the uncovered features in the feature model, we will then repeat this process until all the variants are covered.

To identify a minimum set of variation points which covers a feature model, there are many ways to do it. One of the simple but less optimal approaches would be using greedy algorithm, i.e. selecting the variation points with the biggest coverage until the union of the coverage covers the feature model. A more precise approach is to model the problem as the minimum vertex cover problem and use some approximation algorithm to solve the problem, minimum vertex cover problem is a well studied mathematic problem, there are many approximation algorithms we could use. In the following section, we will discuss this topic.

4 Vertex Cover Problem and Simulated Annealing Algorithm

Before we discuss the vertex cover problem, we firstly describe how do we transfer a feature model into a direct graph. The transformation is quite straightforward, every variable feature in the feature model become a vertex in the resulting graph and the dependencies between two variable features become the arcs in the resulting graph. For example, if feature A requires feature B, then there are two vertices in the resulting graph, say vertex A and vertex B. And also there is an arc goes from vertex A to vertex B in the graph. If feature A excludes feature B, then there will be two arcs between vertex A and B. Once we can model the feature model into a direct graph, we can then apply some discrete optimization techniques.

In graph theory terms, a "vertex-cover" of a directed graph (digraph) is a set of vertices such that each arc of the digraph is incident to at least one vertex of the set. A minimum vertex-cover is a vertex-cover of the smallest size. The problem of finding a minimum vertex-cover is a classical optimization problem in computer science. This problem is a typical example of a NP-hard optimization problem and an optimal solution is very hard to obtain in general. Normally randomized algorithms become the first choice. For example, Simulated Annealing and Genetic Algorithm have been used very often in solving the vertex cover problem. The algorithms are efficient and very often can produce reasonable good solutions.

The problem we are dealing with here is very similar to the classical Vertex Cover problem. The key difference is the coverage in the classical minimum vertex cover problem is the immediate neighbor of the node, where in our problem, the coverage can extend to the nodes beyond the immediate neighbor. However, this difference does not introduce much difficulties into the original problem. We believe that a Simulated Annealing algorithm would still be suitable for solving our problem, as the algorithm can produce reasonable good solutions in a given time frame.

Simulated Annealing (SA) is a well known randomized algorithm in approximating optimal solutions. The technique was proposed by Metropolis et al. (1958) and then been further developed for combinatoric optimization by Pincus (1970).

The technique was originally used as a means of finding the equilibrium configuration of a collection of atoms at a given temperature. Analogy with the physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the solutions and on a global parameter T (called the temperature), that is gradually decreased during the process. The current solution changes almost randomly when T is large, but increasingly "downhill" as T goes to zero. Occasionally, we allow the current solution to go worst, which prevents the method from becoming stuck at local minimum.

The pseudo code of the Simulated Algorithm 4.1 for solving the minimum vertex cover problem is given below. In the inner "for loop", an original vertex-cover is created at random using "Breadth-First Search" (BFS). BFS is a graph searching algorithm that begins at a root vertex (also selected at random base) and explores all the neighboring vertices. Then for each of those neighboring vertices, it explores their unexplored neighbor vertices, and so on, until it exhausts all the vertices. Once the for loop terminates, we have an vertex cover which might not be optimal. Next, we start the SA process, staring from the vertex-cover found before, we shall randomly choosing a vertex from the original vertex-cover to be replaced, in order to obtain a new vertex-cover. If a smaller vertex-cover is produced, then we continue from the new vertex-cover. Otherwise, with certain probability, say $e^{-\Delta/T}$, we still continue from the new vertex-cover, where T is a global time-varying parameter called the *Temperature* and Δ is the increase in cost (i.e., $|VC(G)| - |VC_{min}(G)|$). A limited number of iteration is accepted at each Temperature level. The optimality of the results depends on the number of iterations. The more iterations we run, the results we found are closer to the optimal results, however, the more running time it will consume.

The HSAGA algorithm is an improvement of the standard SA algorithm and was introduced by Tang et al. (2008). The algorithm combines the Genetic Algorithm (GA) and Simulate Annealing (SA) algorithm. The HSAGA algorithm have multiple iterations, in each iteration, we start from multiple instances, i.e, solutions or partial solutions, and we apply SA on these instances to produce the offsprings. The offsprings will then be crossed over in the GA algorithm and producing instances for next iteration. The key idea is to balance the effort in local optimization and multiple probing. The algorithm has demonstrated its efficiency in several classical discrete optimization problems, for more details on the HSAGA algorithm, please see works by Tang et al. (2008).

We have applied the HSAGA algorithms in finding the minimum cover of the feature model. We have modified the algorithm and the algorithm will start from a random selected variable feature in the feature model, since we know it is coverage, i.e. the set of vertices covered by the vertex, so we can remove the vertex and also the set of vertex it covers, and then we randomly select another vertex and repeat the same process. When this process finishes, we will have a cover of the feature model. We will produce multiple covers in the same way and then apply the HSAGA algorithm on the instances. We would like to note that the HSAGA do not out perform the standard SA algorithm if the instance is small, for example, if the graph has less than a few thousand nodes.

Algorithm 4.1: SIMULATED ANNEALING(G)

```
comment: $T = 1.0$, $CR = 0.75$, $VC_{min}(G) = \{\}$
while No change in $\mathrm{VC_{min}(G)}$
  do
    for $i \leftarrow 0$ to Iteration-length
      do
        Generate a VC(G)
        $\Delta \leftarrow |\mathrm{VC(G)}| - |\mathrm{VC_{min}(G)}|$
        if $\Delta < 0$
          then $\mathrm{VC_{min}(G)} \leftarrow \mathrm{VC(G)}$
          else if $random[0;1) < e^{-\Delta/T}$
          then $\mathrm{VC_{min}(G)} \leftarrow \mathrm{VC(G)}$
    T=T*CR;
return $(\mathrm{VC_{min}(G)})$
```

- $VC(G)$ - A vertex-cover of a directed graph G.
- $VC_{min}(G)$ - the minimum vertex-cover of a directed graph G.
- T - Temperature, usual the initial value of Temperature is 1.0.
- CR - Cooling-rate, typical values for Cooling-rate are in the range from 0.75 to 0.98.
- *Iteration-length* - A limited number of iteration is accepted at each Temperature level. For better results, we use $100*|V(G)|$ as the maximum number of iteration, where $V(G)$ is the total number of vertices in G.

In this paper, we use $100*|V(G)|$ as the maximum number of iteration. Once the maximum number of iteration has been reached, the temperature is lowered and a new iteration begins. If a more accurate solution is expected, then the number of iteration should be increased.

5 Case Study

In this section, we will introduce a case study to illustrate how our approach works. In Fig. 4, we present a modified version of a Library System feature model based on our previous work (Lin et al. 2010), where 19 variant points are added to make the feature model non-trivial. As we can see, there are over 60 variable features under 42 VPs. Each variation point is represented by a name, such as VP1 and VP2. A hollow circle indicates the variation point that is linked to a set of variants. Features linked to a solid circle in the figure are mandatory features. The dependencies among the variants are presented in Table 1. Each variant listed in the table is assigned with a Variant ID, called VID. The "Requires" and "Excludes" columns represent the dependencies among the variants. We use this dependency information to create a directed graph shown in Fig. 3. In this directed graph each variant is represented as a node labeled by its VID, dependencies among the variants are represented as arcs among them. For example, the "Requires" relationship between the variant "Reward Point" (VID =11) and the variant "Reward Policy" (VID=3) shown in Table 1 corresponds to an arc from the node 11 to 3 in Fig. 3.

Using this directed graph, we calculate the $MAXCC$ for each variation point. Table 2 shows the $MAXCC$ of each variation point and the corresponding MCS. The MCS column in the table means the corresponding selection of variants that results in $MAXCC$ for the variation point. For example, for $VP7$, the MCS is $4 \cup \neg 5$, it means that if we include variant 4 but do not include variant 5, this selection of variants at $VP7$ will result in the inclusion/exclusion of another 4 variants from other variation points as shown in the table. This set of variants is MCS of $VP7$.

Based on the $MAXCC$ of each variation point, we can find a set of variation points which covers the feature model, and the set is the smallest as possible. If the MCS of each variation point in this set is selected the whole feature model will be covered, i.e. we do not need to go through other variation points for the product configuration. If in the configuration process, at a variation point, the selection is not the one giving the $MAXCC$, then we will have to recalculate the covering set.

A minimum covering set for the Library software product line has been identified and sorted into a sequence in terms of the size of their CC which are shown in Table 3. We can see that the variation points $VP35$, $VP9$, $VP2$, $VP8$, $VP11$, $VP10$, $VP30$, $VP15$, $VP6$, $VP27$, $VP34$, $VP31$, $VP40$, $VP41$ and $VP39$ covers all the variants of the feature model. In this particular feature model, some of VPs do not have any dependency relationships with other variants except their own parent or children variants, such as $VP4$, $VP5$, $VP26$, $VP32$ and $VP42$, which are not listed in Table 2 and we will deal with these VPs after we processed the others.

Since $VP35$ has the biggest coverage in the sequence, so software engineers should start the configuration by examining the variants associated with the $VP35$ and making selections. Assuming that the selection of variants at $VP35$ is the same of its MCS, the configuration will continue to select variants from $VP9$ that is the second in the sequence. The configuration process is going to continue until all the variation points in the sequence have been visited, we then get a product configuration. This is an ideal situation where the MCS of each variation in the sequence is selected. However, if the selected variant set at a variation point is not its MCS then the covering set should be recalculated. Suppose that the selection of variants at $VP8$ is 6 that is not MCS of $VP8$, in this case, we will first work out the coverage of the new selection, which is $\{5, 46\}$, and remove this set of variants from the directed graph we have generated at the beginning of this section. Then we will recalculate the cover for the left over variant features. This cover is shown in Table 4.

Using $MAXCC$ of a variation point as its Configuration Coverage works for those large feature models which contain dense dependency relationships. However, for small and simple feature models, we recommend to use the *Average CC* instead. The reason is that, for small and simple feature model, very often the $MAXCC$ of a variation point significantly larger than the CC of the other selections, giving somehow false indication of how important is

Table 1: The Dependency Relationships among Variants in Feature Model.

VID	Variant	Requires	Excludes	VID	Variant	Requires	Excludes
0	Payment			31	Overdue Fee	0,2,12,17	
1	eBook	41,51	48	32	Damage Cost	0,2	
2	Fee Policy			33	Reserve Fee	0,2	
3	Reward Policy			34	Online Reserve	51,57	48
4	Registration Fee	0,2		35	Online Cancel	51,57	48
5	Issue Library Card	46		36	Overdue Notification	12,17	
6	Replace Library Card	5,46		37	Fee Notification	2,12,17	
7	Renew Fee	0,2		38	Email	10	
8	Phone Number			39	Post	9	
9	Address			40	SMS	8	
10	Email Address			41	Digital Library	51,53	48
11	Reward Point	3,12		42	On-site Explore	51,57	48
12	Borrowing History			43	Web Explore	51,57	48
13	Update Phone Number	8		44	On-site Print	50	48
14	Update Address	9		45	Download	51,57	48
15	Update Email Address	10		46	Library Card Device		
16	Loan Fee	0,2		47	Self-check Device		
17	Record Check	12		48	Non-Network		1,22,24,25,26, 28,29,34,35,41,42, 43,44,45,49,57
18	Pops-up Reminder	12		49	Network Based		48
19	Loan Restriction	12,17		50	LAN Based		
20	Front Desk	12,17,18		51	Internet Based	53	
21	Self-check	12,17,18,47		52	Wireless Device		
22	Web Search	51,57	48	53	Network Security		
23	View Account	12		54	Message Encryption		
24	Website	51,57	48	55	Use Library Card	5,46	
25	On-site Computer	50	48	56	Use Digital Certificate	59	
26	Inter-Library		48	57	User Web Interface	51,53	48
27	Onsite Loan			58	Credit Card	54	
28	Web Request	51,57	48	59	Digital Device		
29	InterLibrary Search	56	30,48	60	Firewall		
30	External Database		29	61	Proxy Server		

Table 2: Max Configuration Coverage of Each VP.

VPID	MCS	MAXCC	VPID	MCS	MAXCC
VP1	$\neg 0$	{4,7,16,31,32,33}	VP22	35	{48,51,57}
VP2	$\neg 2 \cup \neg 3$	{4,7,11,16,31,32,33,37}	VP23	$36 \cup 37$	{2,12,17}
VP3	1	{41,48,51}	VP24	$38 \cup 39 \cup 40$	{8,9,10}
VP6	11	{3,12}	VP25	23	{12}
VP7	$4 \cup \neg 5$	{0,2,6,55}	VP27	$24 \cup 25$	{48,50,51,57}
VP8	$6 \cup 7$	{0,2,5,46}	VP28	26	{48}
VP9	$\neg 12$	{11,17,18,19,20, 21,23,31,36,37}	VP29	28	{48,51,57}
VP10	$\neg 8 \cup \neg 9 \cup \neg 10$	{13,14,15,38,39,40}	VP30	29	{30,48,56}
VP11	$13 \cup 14 \cup 15$	{8,9,10}	VP31	41	{48,51,53}
VP12	16	{0,2}	VP33	$42 \cup 43 \cup 44 \cup 45$	{48,50,51,57}
VP13	$\neg 17$	{19,20,21,31,36,37}	VP34	$\neg 46 \cup \neg 47 \cup \neg 52$	{5,6,21,52,55}
VP14	19	{12,17}	VP35	48	{1,22,24,25,26,28,29,34, 35,41,42,43,44,45,49,57}
VP15	21	{12,17,18,47}	VP36	$50 \cup \neg 51$	{1,22,24,28,34,35, 41,42,43,45,57}
VP16	$\neg 18$	{20,21}	VP37	$\neg 57$	{22,24,28,34,35,42,43,45}
VP17	22	{48,51,57}	VP38	$\neg 53$	{41,51,57}
VP18	31	{0,2,12,17}	VP39	$\neg 54$	{58}
VP19	32	{0,2}	VP40	$55 \cup 56$	{5,46,59}
VP20	33	{0,2}	VP41	58	{54}
VP21	34	{48,51,57}			

Table 3: A Sequence of VPs which covers Feature Model.

VP	MCS	MAXCC
VP35	48	{1,22,24,25,26,28,29,34, 35,41,42,43,44,45,49,57}
VP9	$\neg 12$	{11,17,18,19,20,21,23,31,36,37}
VP2	$\neg 2 \cup \neg 3$	{4,7,11,16,31,32,33,37}
VP8	$6 \cup 7$	{0,2,5,46}
VP11	$13 \cup 14 \cup 15$	{8,9,10}
VP10	$\neg 8 \cup \neg 9 \cup \neg 10$	{13,14,15,38,39,40}
VP30	29	{30,48,56}
VP15	21	{12,17,18,47}
VP6	11	{3,12}
VP27	$24 \cup 25$	{48,50,51,57}
VP34	$\neg 46 \cup \neg 47 \cup \neg 52$	{5,6,21,52,55}
VP31	41	{48,51,53}
VP40	$55 \cup 56$	{5,46,59}
VP41	58	{54}
VP39	$\neg 54$	{58}

the variation point to the configuration process. So the *Average CC* gives better guide in the process. *Average CC* is defined as the set which includes the variants that appear more than once in the *CC* of all the valid selections at the variation point. For example, there are four possible selections at $VP2$, $\neg 2 \cup \neg 3 = \{4, 7, 11, 16, 31, 32, 33, 37\}$, $2 \cup 3 = \emptyset$, $\neg 2 \cup 3 = \{4, 7, 16, 31, 32, 33, 37\}$, $2 \cup \neg 3 = \{11\}$. So the average CC is $\{4, 7, 16, 31, 32, 33, 37\}$.

We have tested our approach on this specific library system feature model. We also conducted experiments on several random generated feature models. All experiments generated good results on corresponding feature models. According to the results, our approach reduces the configuration efforts spent on feature decisions significantly. In next two sections, we will introduce the processes of our experiments and explain the related results.

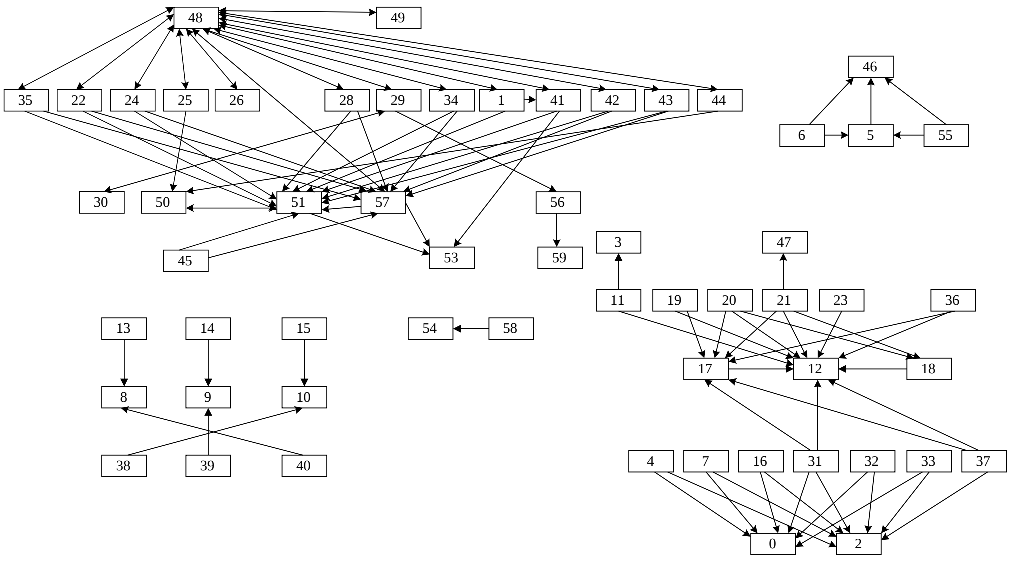


Figure 3: Dependencies among Variants in Library Systems.

Table 4: A Sequence of VPs after Decision made at VP8.

VP	MCS	MAXCC
VP35	48	{1,22,24,25,26,28,29,34, 35,41,42,43,44,45,49,57}
VP9	$\neg 12$	{11,17,18,19,20,21,23,31,36,37}
VP2	$\neg 2 \cup \neg 3$	{4,7,11,16,31,32,33,37}
VP11	$13 \cup 14 \cup 15$	{8,9,10}
VP10	$\neg 8 \cup \neg 9 \cup \neg 10$	{13,14,15,38,39,40}
VP30	29	{30,48,56}
VP15	21	{12,17,18,47}
VP6	11	{3,12}
VP27	$24 \cup 25$	{48,50,51,57}
VP34	$\neg 46 \cup \neg 47 \cup \neg 52$	{5,6,21,52,55}
VP31	41	{48,51,53}
VP40	$55 \cup 56$	{5,46,59}
VP12	16	{0,2}
VP41	58	{54}
VP39	$\neg 54$	{58}

6 Experiment Results

We had conducted an experiment by using the Library System feature model. Firstly, we developed the requirement documents for two library systems based on the feature model. One with our university library in mind, the library is not only available for students and staff, but also available for public access. The other library system is a medium sized community library for local residents. The university library is more complex than the community library, for example, the digital library access is part of the university library but not for the community library.

The requirement documents of two systems serve two purposes, 1) to give users some general ideas of what features the library system might include, 2) to control the scope of the final product. Because the final products are configured using the same feature model, we presume that, for a given library requirement, the products will be similar, i.e. with minor differences in terms of features included. If one of the final products been configured in the experiment is significantly different from other products, for instance, contains a lot more features than the average case, then it is not a valid sample and we will reject the product. The reason to do so is to have some consistency among the configured products so we could compare the result. Furthermore, as the economical concerns, such as software development cost, are not included in the feature model, the configured product could be unrealistic and having too many or too little features, thus we have reject these products in our experiment. To better simulate the real world situation, where the requirements are very often not clear and exact, some vague descriptions and even misleading information are introduced to the requirement documents.

The experiment consisted of two focus groups with 20 university students. Users were picked randomly without SPL experience to make sure they are on the same level of background. The basic idea of SPL and purpose of experiment were explained to all the users in two groups. One group used traditional configuration method (depth-first traversal to get through each variant) and the other group used our approach based on variant points sequence generated from *Average CC*. The experiment was carried out in pairs, i.e. one user from each group. Based on the functional and non-functional requirements listed in the requirement document, users made their selections on the feature model. Each user is given a computer program that could record all the decisions he/she have made, when the computer program identified the conflicts among the selected features, the computer program will remind the experiment conductor who then explains the conflict and help user to change their selections. If there was anything user feels unclear about, then he/she could ask questions during the experiments, the information was shared with the peer in the experiments. This parallel process was able to provide a direct comparison of two methods. For each group, 50% of users configured the university library system and the other half configured the community library.

Several key data sets were collected during experiment process. Including the number of Decisions (users' selection of variants), the number of Rollbacks (users has realized that they had made a mistake and modified their previous selection) and Time consumed for configuration. The number of Decisions would be the most important figure to demonstrate the efficiency of our approach.

Considering the variant features involved in the dependency relationships, they form trees/chains. In our approach, configuration decision is always made at the root of the tree/chain, thus, less number of Rollbacks. Rollback represents errors in the configuration, where the impact of one improper selection does not reflect immediately but afterwards. It seems little happened in our approach. Time is a parameter to support configuration procedure somehow. It only reflects the process to some extent because there are so many factors may influence the time consumed. For instance, users' understanding of the features.

Table 5 shows critical parameters collected for randomly selected 10 pairs of users in our experiment. Data for each pair is included in a row in Table 5 for comparison. Rollbacks did not happen at all for the user using our approach. Time consumed by users using our approach is much less than group using traditional approach when configuring the same product. From the "Decision" column of two groups, we can see the gaps between this two approaches are obvious even for this simple feature model. Figures in this experiment may vary slightly but we can still see the tendency. Our approach definitely has advantage in terms of the number of decision to make and the number of Rollbacks. Table 6 shows average value comparison of two approaches.

Table 5: Tradition Configure Group(left) and Optimization Configure Group(right).

User No.	Decision	Rollback	Time(mins)	User No.	Decision	Rollback	Time(mins)
1	42	0	30	2	34	0	23
3	43	1	25	4	34	0	21
5	42	0	23	6	34	0	20
7	42	0	28	8	33	0	22
9	43	1	30	10	34	0	26
11	42	0	28	12	32	0	24
13	42	0	24	14	33	0	16
15	42	1	20	16	33	0	15
17	44	2	28	18	32	0	22
19	42	0	30	20	33	0	24

Table 6: Average Figures Comparison.

Group	Average steps	Average time	Average decision reduced(%)
Traditional	42.4	26.6	21.7%
Optimization	33.2	21.3	

7 Experiment on Random graphs

In the literature (Segura et al. 2010), there are some works using random generated feature model as the testing environment. Here we have also conducted an experiment on random generated feature model. The random feature model is generated in three steps, first a random graphs is generated based on the simple random graph model by Erdos and Renyi (Erdos et al. 1959), where we start with a fixed set of vertices and add edges to the graph based on a edge probability parameter. In here, a vertex represents a variable feature, an edge represents a dependency relationship in a graph. In the second step, we randomly generated the relationships between the features. Except "Requires" and "Excludes" relationships, we also included parent-children relationships and the multiplicity constraints between variants in the random feature models. Of cause, it is easy to see that here, very likely, large number of conflicts will be generated, so we have to run through the third step, where we go through all the cycles in the generated graphs to check for inconsistency. As explained before, a cycle possibly represents a conflict, therefore, we have to randomly remove an edge a cycle to destroy the cycle, thus to remove the conflict. Once we complete these three steps, we then have a valid feature model for configuration.

The configuration process is automated as well, a computer program randomly selects a feature and then check if the selection conflicts with previous decisions, if yes, then the program will make a different selection or randomly change the previous decisions to remove the conflicts. The program repeats the process until all the features are visited. When using our proposed approach, there will be no rollbacks.

Obviously, the final products generated by the random configuration can be quite different. i.e. a product could contain large number of features while the others might contain significantly less number of features. To maintain the similarity of the final products, we reject those products (i.e. invalid products) which contain more than 85% of features or less than 50% of features of the feature model.

We have conducted our experiments on a series of random systems. We have generated three random systems of 800 nodes, 2000 nodes and 3000 nodes of different edge probabilities. We then configure 1000 valid products using each of the configuration methods on the three random systems. And we include the exact numbers of the decisions of our approach/the traditional approach in Table 7 below.

Table 7: Experiment Results of Random Systems.

Edge Probability	Random graph with 800 nodes	Random graph with 2000 nodes	Random graph with 3000 nodes
0.01	264/413	330/473	361/498
0.02	219/307	255/374	298/411
0.1	124/169	145/186	167/199
0.2	91/112	122/157	133/145

The result is displayed in Table 7. Columns represent three random systems of 800, 2000 and 3000 nodes. Rows indicate edge probability when creating the random graph and corresponding cells represent the exact decision numbers of both approaches (i.e our approach vs traditional). With the growth of the edge probability, we can see that the decisions made between two approaches becoming closer. This is because that the number of edges is growing, and the dependencies relationships among the variation points become dense, thus more likely the random selection has large coverage, and the gaps decrease. From Table 7, it is quite clear that using our approach is more efficient in the product configuration.

8 Conclusions and Future Works

In this paper, we have presented an approach to improve the efficiency of product configuration in software product line. Using this approach, a set of variant points (VPs) is identified from the feature model. This set only contains small number of VPs but the union of CC of these VPs will cover all the variants of the feature model. To configure a product, instead of making configuration decision at every VP of the feature model, we only go through small number of VPs to make configuration decisions. In this way, the number of decisions is reduced, thus the effort of decision making is saved. Furthermore, using this approach, it is less likely to make mistakes in the configuration, as the we have already incorporated the dependency relationships among the variants in our approach.

However, as we have pointed out, that our approach only considers two simple relationships among the variable features, therefore, how to extend our approach to cover other types of relationships is a challenge and we would like to consider this in our future works. Meanwhile, we would like to perform our experiments on some publicly available and complex feature models to further evaluate our approach, we believe that the results must be interesting to many researchers.

References

Benavides, D., Segura, S. & Ruiz-Cortes, A. (2010), Automated Anslysis of Feature Models 20 Years Later: A Literature Review, *Information Systems*, 35(6), 625–636.

Czarnecki, K., Helsen, S. & Eisenecker, U. (2005), Formalizing Cardinality-based Feature Models and Their Staged Configuration, *Software Process: Improvement and Practice*, 10(1), 7–29.

Czarneck, K. & Kim, P. (2005), Cardinality-basd Feature Modeling and Constraints: A Progress Report, *in* 'International Workshop on Software Factories at OOPSLA 2005', San Diego, California, USA.

Erdos, P. & Renyi, A. (1959), On Random Graphs, *Publicationes Mathematicae*, 6, 290–297.

Kang, K., Cohen, S., Hess, J., Nowak, W. & Peterson, S. (1990), Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, Software Engineering Institute, Carnegie Mellon University.

Lee, K., Kang, K. & Lee, J. (2002), Concepts and Guidelines of Feature Modeling for Product Line Software Engineering, *in* '7th International Conference on Software Reuse: Methods, Techniques, and Tools', pp. 62–77.

Lin, Y. Q., Ye, H. L. & Tang, T. M. (2010), An Approach to Efficient Product Configuration in Software Product Lines, *in* '14th International Conference on Software Product Lines', Jeju Island, South Korea, pp. 435–439.

Loesch, F. & Ploedereder, E. (2007), Optimization of Variability in Software Produce Lines, *in* '11th International Software Product Line Conference', pp. 151–162.

Mannion, M. (2002), Using First-Order Logic for Product Line Model Validation, *in* '2nd International Conference on Software Product Lines', p-p. 149–202.

Mendonca, M. Cowan, D. Malyk, W. & Oliveira, T. (2008), Collaborative Product Configuration: Formalization and Efficient Algorithms for Dependency Analysis, *Journal of Software*, 3(2), 69–82.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H. & Teller, E. (1958), Equations of State Calculations by Fast Computing Machines, *Journal of Chemical Physics*, 21, pp. 1087–1092.

Pincus, M. (1970), A Monte Carlo Mehtod for the Approximate Solution of Certian Types of Constrained Optimization Problems, *Operations Research*, 18(6), 1225–1228.

Pohl, K., Böckle, G. & van der Linden, F. (2005), Software Product Line Engineering: Foudations, Principles, and Techniques, *Springer*, Heidelberg, (2005).

Segura, S., Hierons, R. M., Benavides, D. & Ruiz-Cortes, A. (2010), Automated Test Data Gneration on the Analyses of Feature Models: A Metamorphic Testing Approach, *in* '3rd International Conference on Software Testing, Verification and Validation', Paris, France, pp. 35–44.

Tang, J. M., Miller, M. & Lin, Y. Q. (2008), HSAGA and its application for the construction of near-Moore digraphs, *Journal of Discrete Algorithms*, 6(1), 73–84.

Trinidad, P., Benavides, D., Duran, A., Ruiz-Cortes, A. & Toro, M. (2008), Automated Error Analysis for Agilization of Feature Modeling, *Jouranl of Systems and Software*, 81(6), 883–896.

White, J., Dougherty, B., Schmide, D. C. & Benavides, D. (2009), Automated Reasoning for Multi-step Feature Model Configuration Problems, *in* '13th International Software Product Line Conference', San Francisco, USA, pp. 11–20.

Ye, H. L. & Zhang, W. (2008), Formal Definition of Feature Models to Support Software Product Line Evolutions, *in* '2008 International Conference on Software Engineering Research Practice', Las Vegas, Nevada, pp. 349–355.

Zhang, W., Zhao, H. Y. & Mei, H. (2004), A Propositional Logic-based Method for Verification of Feature Models, *Formal Methods and Software Engineering*, 3308, 115–130.

Zhang, G. H., Ye, H. L. & Lin, Y. Q. (2011), Feature Model Validation: A Constraint Propoagation-based Approach, *in* '10th International Conference on Software Engineering Reaserch and Practice', Las Vegas, Nevada.

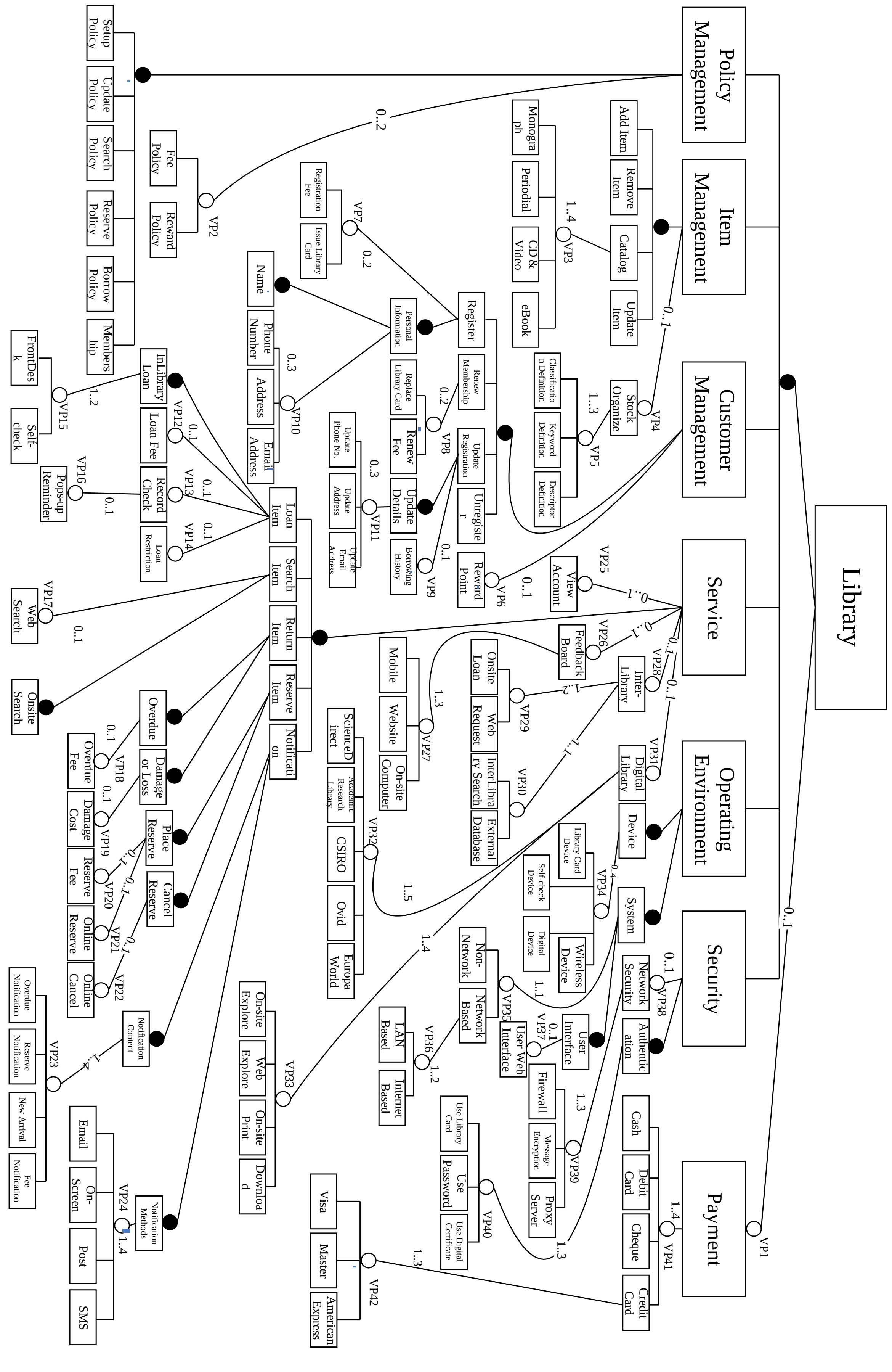


Figure 4: A Feature Model for SPL of Library Systems.

Author Index

Recent Volumes in the CRPIT Series

ISSN 1445-1336