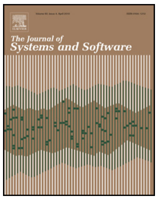
The Journal of Systems and Software 86 (2013) 1978–2001

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss



An orchestrated survey of methodologies for automated software test case generation

Saswat Anand[a], Edmund K. Burke[b], Tsong Yueh Chen[c], John Clark[d], Myra B. Cohen[e], Wolfgang Grieskamp[f], Mark Harman[g], Mary Jean Harrold[h], Phil McMinn[i]

[a] *Stanford University, USA*
[b] *University of Stirling, Scotland, UK*
[c] *Swinburne University of Technology, Australia*
[d] *University of York, UK*
[e] *University of Nebraska-Lincoln, USA*
[f] *Microsoft Research, Redmond, USA*
[g] *University College London, UK*
[h] *Georgia Institute of Technology, USA*
[i] *University of Sheffield, UK*

Orchestrators and Editors,
Antonia Bertolino[j,1], J. Jenny Li[k,2], Hong Zhu[l,3]

[j] *ISTI-CNR, Italy*
[k] *Avaya Labs Research, USA*
[l] *Oxford Brookes University, UK*

ARTICLE INFO

Article history:
Received 11 February 2013
Accepted 11 February 2013
Available online 17 April 2013

Keywords:
Adaptive random testing
Combinatorial testing
Model-based testing
Orchestrated survey
Search-based software testing
Software testing
Symbolic execution
Test automation
Test case generation

ABSTRACT

Test case generation is among the most labour-intensive tasks in software testing. It also has a strong impact on the effectiveness and efficiency of software testing. For these reasons, it has been one of the most active research topics in software testing for several decades, resulting in many different approaches and tools. This paper presents an orchestrated survey of the most prominent techniques for automatic generation of software test cases, reviewed in self-standing sections. The techniques presented include: (a) structural testing using symbolic execution, (b) model-based testing, (c) combinatorial testing, (d) random testing and its variant of adaptive random testing, and (e) search-based testing. Each section is contributed by world-renowned active researchers on the technique, and briefly covers the basic ideas underlying the method, the current state of the art, a discussion of the open research problems, and a perspective of the future development of the approach. As a whole, the paper aims at giving an introductory, up-to-date and (relatively) short overview of research in automatic test case generation, while ensuring a comprehensive and authoritative treatment.

E-mail addresses: saswat@cs.stanford.edu (S. Anand), e.k.burke@stir.ac.uk (E.K. Burke), tychen@swin.edu.au (T.Y. Chen), John.Clark@cs.york.ac.uk (J. Clark), myra@cse.unl.edu (M.B. Cohen), wgrieskamp@gmail.com (W. Grieskamp), mark.harman@ucl.ac.uk (M. Harman), harrold@cc.gatech.edu (M.J. Harrold), p.mcminn@sheffield.ac.uk (P. McMinn).

[1] **Antonia Bertolino** (http://www.isti.cnr.it/People/A.Bertolino) is a Research Director of the Italian National Research Council (CNR), in Pisa. She investigates approaches for model-based and security testing, for service-oriented and component-based test methodologies, as well as for monitoring of non-functional properties of composite services. Currently she is involved in the European Projects CHOReOS, NESSOS and Presto4u, and serves as the Area Editor for Software Testing for the Elsevier Journal of Systems and Software, and as an Associate Editor of Springer Empirical Software Engineering and IEEE Transactions on Software Engineering. In the next couple of years she will be busy organizing the 2015 edition of the International Conference of Software Engineering in Florence (Italy). She has (co)authored over 100 papers in international journals and conferences.

[2] **Dr. Jenny Li** is a research scientist at Avaya Labs, formerly part of Bell Labs. She is an experienced industrial researcher with more than 70 papers published in technical journals and conferences, and holder of over 20 patents with five pending applications. She led a software immunization project at Avaya for software reliability and security improvement. Her specialties are in automatic failure detection, with particular emphasis on reliability, security, performance and testing. Prior to joining Avaya, she worked at Bellcore (formerly Telcordia and now Applied Communication Science) for 5 years. She received her Ph.D. from University of Waterloo in 1996.

[3] **Hong Zhu** is a professor of computer science at Oxford Brookes University, where he chairs the Applied Formal Methods Research Group. He is a senior member of IEEE Computer Society, a member of British Computer Society, ACM, and China Computer Federation. His research interests are in the area of cloud computing and software engineering including software testing, software development methodology, agent technology, automated software development tools, etc. He is a co-founder and the chair of the Steering Committee of the IEEE/ACM International Workshops on Automation of Software Test (AST). He has published more than 160 research papers and 2 books.

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.02.061

1. Introduction

Software testing is indispensable for all software development. It is an integral part of the software engineering discipline. However, testing is labour-intensive and expensive. It often accounts for more than 50% of total development costs. Thus, there are clear benefits in reducing the cost and improving the effectiveness of software testing by automating the process. In fact, there has been a rapid growth of practices in using automated software testing tools. Currently, a large number of software test automation tools have been developed and have become available on the market.

Among a range of testing activities, test case generation is one of the most intellectually demanding tasks and it is also of the most critical challenges, since it can have a strong impact on the effectiveness and efficiency of the whole testing process (Zhu et al., 1997; Bertolino, 2007; Pezzè and Young, 2007). It is no surprise that a great amount of research effort in the past few decades has been spent on automatic test case generation. As a result, a significant number of different techniques for test case generation have been advanced and rigorously investigated.[4] In addition, software systems have become more and more complicated, for example, with components developed by different vendors, using different techniques in different programming languages and even running on different platforms. Although automation techniques for test case generation have started to be gradually adopted by the IT industry in software testing practice, there still exists a big gap between real software application systems and the practical usability of the test case generation techniques proposed by the research community. We believe that for researchers in software test automation it is highly desirable to critically review the existing techniques, to recognize the open problems and to put forward a perspective on the future of test case generation.

With this aim in mind, this paper offers a critical review of a number of prominent test case generation techniques and it does so by taking a novel approach that we call an *orchestrated survey*. This consists of a collaborative work collecting self-standing sections, each focusing on a key surveyed topic, in our case a test generation technique. Each section is independently authored by a world-renowned active researcher (or researchers) on the topic. The surveyed topics have been selected (and orchestrated) by the editors.

Generally speaking, test cases, as an important software artifact, must be generated from some information, that is some other types of software artifacts. The types of artifacts that have been used as the reference input to the generation of test cases include: the program structure and/or source code; the software specifications and/or design models; information about the input/output data space, and information dynamically obtained from program execution. Thus, the techniques we consider in this paper include:

1. symbolic execution and program structural coverage testing;
2. model-based test case generation;
3. combinatorial testing;
4. adaptive random testing as a variant of random testing;
5. search-based testing.

Of course, automatic test case generation techniques may exploit more than one type of software artifact as the input, thus combining the above techniques to achieve better effectiveness and efficiency. It is worth noting that there are many other automatic or semi-automatic test case generation techniques not covered in this paper, for example, mutation testing, fuzzing and data mutation testing, specification-based testing and metamorphic testing. In order to keep this paper to a reasonable size, we limited our selection to five of the most prominent approaches; future endeavors could be devoted to complement this set with further reviews: orchestrated surveys can in fact be easily augmented with more sections.

Hence, after a brief report in the next section about the process that we followed in conducting the survey, the paper is organized as follows. In Section 3, Saswat Anand and Mary Jean Harrold review typical program-based test case generation techniques using symbolic execution. In Section 4, Wolfgang Grieskamp focuses on model-based test case generation, which is closely related to the currently active research area of model-driven software development methodology. Sections 5 and 6 focus on data-centric test case generation techniques, i.e., combinatorial testing reviewed by Myra B. Cohen, and random testing and its variant of adaptive random testing are reviewed by Tsong Yueh Chen, respectively. Finally, in Section 7, Mark Harman, Phil McMinn, John Clark, and Edmund Burke review search-based approaches to test case generation.

2. About the process of orchestrated survey

The idea behind this "orchestrated" survey, as we call it, originated from the desire to produce a comprehensive survey paper on automatic test generation within the short timeframe of this special section devoted to the Automation of Software Test (AST). There are already several outstanding textbooks and survey papers on software testing. However, the field of software testing is today so vast and specialized that no single author could harness the expertise required for all different approaches and could be informed of the latest advances in every technique. So, typically, surveys necessarily focus on some specific kind of approach. We wanted a review that could somehow stand out from the existing literature by offering a broad and up-to-date coverage of techniques, but without renouncing the depth and authoritaty that are required for each addressed technique. Thus we came up with this idea of selecting a set of techniques and invited renowned experts of each technique to contribute to an independent section.

For each of the included sections, the review consists of a brief description of the basic ideas underlying the technique, a survey of the current state of the art in the research and practical use of the technique, a discussion of the remaining problems for further research, and a perspective of the future development of the approach. Whilst these reviews are assembled together to form a coherent paper, each section remains an independently readable and referable article.

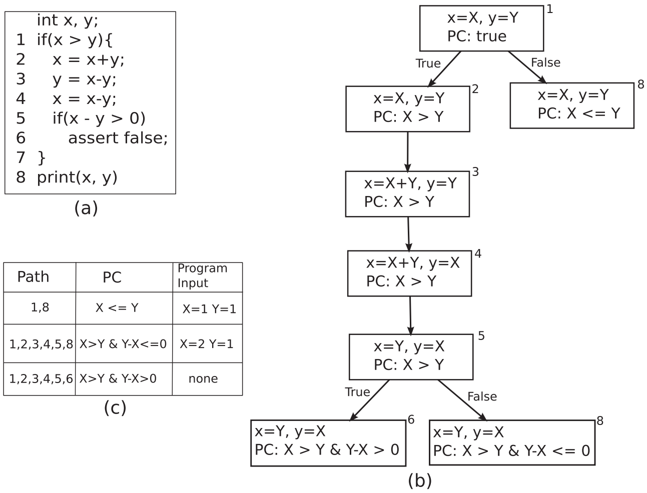
For those authors who accepted our invitation, the submitted sections have not been automatically accepted. Each section underwent a separate peer-review process by at least two (often three) reviewers, following the normal standards of this journal's reviewing process. Some of them were subject to extensive revision and a second review round before being accepted. The five finally accepted sections were then edited in their format and collated by us into this survey paper, which we proudly offer as an authoritative source both to get a quick introduction to research in automatic test case generation and as a starting point for researchers willing to pursue some further direction.

3. Test data generation by symbolic execution

By *Saswat Anand* and *Mary Jean Harrold*[5]

[4] See, for example, the Proceedings of IEEE/ACM Workshops on Automation of Software Test (AST'06–AST'12). URL for AST'13: http://tech.brookes.ac.uk/AST2013/.

[5] *Acknowledgements*: This research was supported in part by NSF CCF-0541049, CCF-0725202, and CCF-1116210, and IBM Software Quality Innovation Award to Georgia Tech.



```
    int x, y;
1  if(x > y){
2     x = x+y;
3     y = x-y;
4     x = x-y;
5     if(x - y > 0)
6        assert false;
7  }
8  print(x, y)
```

(a)

Path	PC	Program Input
1,8	X <= Y	X=1 Y=1
1,2,3,4,5,8	X>Y & Y-X<=0	X=2 Y=1
1,2,3,4,5,6	X>Y & Y-X>0	none

(c)

Fig. 1. (a) Code that swaps two integers, (b) the corresponding symbolic execution tree, and (c) test data and path constraints corresponding to different program paths.

Symbolic execution is a program analysis technique that analyzes a program's code to automatically generate test data for the program. A large body of work exists that demonstrates the technique's usefulness in a wide range of software engineering problems, including test data generation. However, the technique suffers from at least three fundamental problems that limit its effectiveness on real world software. This section provides a brief introduction to symbolic execution, a description of the three fundamental problems, and a summary of existing well known techniques that address those problems.

3.1. Introduction to symbolic execution

In contrast to black box test data generation approaches, which generate test data for a program without considering the program itself, white box approaches analyze a program's source or binary code to generate test data. One such white box approach, which has received much attention from researchers in recent years, uses a program analysis technique called symbolic execution. *Symbolic execution* (King, 1975) uses symbolic values, instead of concrete values, as program inputs, and represents the values of program variables as symbolic expressions of those inputs. At any point during symbolic execution, the state of a symbolically executed program includes the symbolic values of program variables at that point, a path constraint on the symbolic values to reach that point, and a program counter. The *path constraint* (PC) is a Boolean formula over the symbolic inputs, which is an accumulation of the constraints that the inputs must satisfy for an execution to follow that path. At each branch point during symbolic execution, the PC is updated with constraints on the inputs such that (1) if the PC becomes unsatisfiable, the corresponding program path is infeasible, and symbolic execution does not continue further along that path and (2) if the PC is satisfiable, any solution of the PC is a program input that executes the corresponding path. The *program counter* identifies the next statement to be executed.

To illustrate, consider the code fragment[6] in Fig. 1(a) that swaps the values of integer variables `x` and `y`, when the initial value of `x` is greater than the initial value of `y`; we reference statements in the figure by their line numbers. Fig. 1(b) shows the symbolic execution tree for the code fragment. A *symbolic execution tree* is a compact representation of the execution paths followed during the symbolic execution of a program. In the tree, nodes represent program states, and edges represent transitions between states. The numbers shown at the upper right corners of nodes represent values of program counters. Before execution of statement 1, the PC is initialized to true because statement 1 is executed for any program input, and `x` and `y` are given symbolic values X and Y, respectively. The PC is updated appropriately after execution of `if` statements 1 and 5. The table in Fig. 1(c) shows the PC's and their solutions (if they exist) that correspond to three program paths through the code fragment. For example, the PC of path (1,2,3,4,5,8) is $X > Y \& Y - X <= 0$. Thus, a program input that causes the program to take that path is obtained by solving the PC. One such program input is $X = 2$, $Y = 1$. For another example, the PC of path (1,2,3,4,5,6) is an unsatisfiable constraint $X > Y \& Y - X > 0$, which means that there is no program input for which the program will take that (infeasible) path.

Although the symbolic execution technique was first proposed in the mid seventies, the technique has received much attention recently from researchers for two reasons. First, the application of symbolic execution on large, real world programs requires solving complex and large constraints. During the last decade, many powerful constraint solvers (e.g., Z3 (de Moura and Bjorner, 2008), Yices (Dutertre and de Moura, 2006), STP (Ganesh and Dill, 2007)) have been developed. Use of those constraint solvers has enabled the application of symbolic execution to a larger and a wider range of programs. Second, symbolic execution is computationally more expensive than other program analyses. The limited computational capability of older generation computers made it impossible to symbolically execute large programs. However, today's commodity computers are arguably more powerful than the supercomputers (e.g., Cray) of the eighties. Thus, today, the barrier to applying symbolic execution to large, real world programs is significantly lower than it was a decade ago. However, the effectiveness of symbolic execution on real world programs is still limited because the technique suffers from three fundamental problems—path explosion, path divergence, and complex constraints—as described in Section 3.2. Those three problems need to be addressed before the technique can be useful in real world software development and testing.

Although symbolic execution has been used to generate test data for many different goals, the most well known use of this approach is to generate test data to improve code coverage and expose software bugs (e.g., Cadar et al., 2008; Godefroid et al., 2008; Khurshid et al., 2003). Other uses of this approach include privacy preserving error reporting (e.g., Castro et al., 2008), automatic generation of security exploits (e.g., Brumley et al., 2008), load testing (e.g., Zhang et al., 2011), fault localization (e.g., Qi et al., 2009), regression testing (e.g., Santelices et al., 2008), robustness testing (e.g., Majumdar and Saha, 2009), data anonymization for testing of database-based applications (e.g., Grechanik et al., 2010), and testing of graphical user interfaces (e.g., Ganov et al., 2008).

For example, Castro et al. (2008) use symbolic execution to generate test data that can reproduce, at the developer's site, a software failure that occurs at a user's site, without compromising the privacy of the user. Zhang et al. (2011) generate test data that leads to a significant increase in program's response time and resource usage. Qi et al. (2009) generate test data that are similar to a given program input that causes the software to fail, but that do not cause failure. Such newly generated test data are then used to localize the cause of the failure. Santelices et al. (2008) generate test data that exposes difference in program's behaviors between two versions of an evolving software. Majumdar and Saha (2009) use symbolic execution to generate test data whose slight perturbation causes a significant difference in a program's output.

A number of tools for symbolic execution are publicly available. For Java, available tools include Symbolic Pathfinder (Pasareanu and Rungta, 2010), JCUTE (Sen and Agha, 2006), JFuzz (Jayaraman

[6] This example, which we use to illustrate symbolic execution, is taken from reference (Khurshid et al., 2003).

et al., 2009), and LCT (Kähkönen et al., 2011). CUTE (Sen et al., 2005), Klee (Cadar et al., 2008), S2E (Chipounov et al., 2011), and Crest[7] target C language. Finally, Pex (Tillmann and de Halleux, 2008) is a symbolic execution tool for.NET languages. Some tools that are not currently publicly available, but have been shown to be effective on real world programs include SAGE (Godefroid et al., 2008) and Cinger (Anand and Harrold, 2011).

3.2. Fundamental open problems

A symbolic execution system can be effectively applied to large, real world programs if it has at least the two features: (1) efficiency and (2) automation. First, in most applications of symbolic execution based test data generation (e.g., to improve code coverage or expose bugs), ideally, the goal is to discover all feasible program paths. For example, to expose bugs effectively, it is necessary to discover a large subset of all feasible paths, and show that either each of those paths is bug free or many of them expose bugs. Thus, the system must be able discover as many distinct feasible program paths as possible in the available time limit. Second, the required manual effort for applying symbolic execution to any program should be acceptable to the user.

To build a symbolic execution system that is both efficient and automatic, three fundamental problems of the technique must be addressed. Other problems[8] arise in specific applications of symbolic execution, but they are not fundamental to the technique. Although a rich body of prior research[9] on symbolic execution exists, these three problems have been only partially addressed.

3.2.1. Path explosion

It is difficult to symbolically execute a significantly large subset of all program paths because (1) most real world software have an extremely large number of paths, and (2) symbolic execution of each program path can incur high computational overhead. Thus, in reasonable time, only a small subset of all paths can be symbolically executed. The goal of discovering a large number of feasible program paths is further jeopardized because the typical ratio of the number of infeasible paths to the number of feasible paths is high (Ngo and Tan, 2007). This problem needs to be addressed for efficiency of a symbolic execution system.

3.2.2. Path divergence

Real world programs frequently use multiple programming languages or parts of them may be available only in binary form. Computing precise constraints for those programs either requires an overwhelming amount of effort in implementing and engineering a large and complex infrastructure or models for the problematic parts provided by the user. The inability to compute precise path constraints leads to path divergence: the path that the program takes for the generated test data diverges from the path for which test data is generated. Because of the path divergence problem, a symbolic execution system either may fail to discover a significant number of feasible program paths or, if the user is required to provide models, will be less automated.

3.2.3. Complex constraints

It may not always be possible to solve path constraints because solving the general class of constraints is undecidable. Thus, it is possible that the computed path constraints become too complex (e.g., constraints involving nonlinear operations such as multiplication and division and mathematical functions such as `sin` and `log`), and thus, cannot be solved using available constraint solvers. The inability to solve path constraints reduces the number of distinct feasible paths a symbolic execution system can discover.

3.3. Existing solutions

Although the three problems described in the previous section have not been fully solved, many techniques have been proposed to partially address them. In the following, we briefly describe some of those techniques.

3.3.1. Techniques for path explosion problem

Many techniques have been proposed to alleviate the path explosion problem, and they can be classified into five broad classes. Techniques in the first class avoid exploring paths through certain parts of a program by using a specification of how those parts affect symbolic execution. Some techniques (Anand et al., 2008; Godefroid, 2007) automatically compute the specification, referred to as the *summary* in terms of pre- and post-conditions by symbolic execution through all paths of a function. Then, instead of repeatedly analyzing a function for each of its call sites, the summary is used. Other techniques (Bjorner et al., 2009; Khurshid and Suen, 2005; Veanes et al., 2010) use specifications that are manually created. In some of those techniques (Bjorner et al., 2009; Veanes et al., 2010), specifications of commonly used abstract data types such as strings and regular expressions are encoded internally in the constraint solver. One of the main limitations of techniques in this class is that their generated path constraints can be too large and complex to solve.

Techniques (Boonstoppel et al., 2008; Majumdar and Xu, 2009; Ma et al., 2011; Santelices and Harrold, 2010) in the second class are goal driven: they avoid exploring a large number of paths that are not relevant to the goal of generating test data to cover a specific program entity (e.g., program statement). Ma et al. (2011) explore only those paths that lead to the goal. Other techniques (Boonstoppel et al., 2008; Majumdar and Xu, 2009; Ma et al., 2011; Santelices and Harrold, 2010) use a program's data or control dependencies to choose and symbolically execute only a subset of relevant paths. However, these techniques do not address the path explosion problem fully because their effectiveness depends on the structure of data or control dependencies, which can differ significantly between programs.

Techniques in the third class are specialized with respect to a program construct or characteristics of a class of programs. Some techniques (Godefroid and Luchaup, 2011; Saxena et al., 2009) in the first category focus on analyzing loops in a program in a smarter way because loops cause dramatic growth in the number of paths. Other techniques (Godefroid et al., 2008; Majumdar and Xu, 2007) in this class aim to efficiently generate test data for programs that take highly structured inputs (e.g., parser) by leveraging the program's input grammar. Techniques in this class are limited because they are specialized.

Techniques in the fourth class use specific heuristics to choose a subset of all paths to symbolically execute, but while still satisfying the purpose (e.g., obtain high code coverage) of using symbolic execution. Anand et al. (2009) proposed to store and match abstract program states during symbolic execution to explore a subset of paths that are distinguishable under a given state abstraction function. The technique presented by Tomb et al. (2007), which uses symbolic execution to expose bugs, does not symbolically execute

[7] Burnim and Sen, CREST: auomatic test generation tool for C, URL: http://code.google.com/p/crest/.

[8] For example, when symbolic execution is applied to open programs (i.e., parts of the program are missing), a problem arises in maintaining symbolic values corresponding to reference type variables that store abstract data types.

[9] A partial bibliography of papers published in the last decade on symbolic execution and its applications can be found at http://sites.google.com/site/symexbib/; Cadar et al. (2011) and Pasareanu and Visser (2009) provide an overview of prior research on symbolic execution.

paths that span through many methods. Finally, other techniques (Chipounov et al., 2011; Godefroid et al., 2008; Majumdar and Sen, 2007; Pasareanu et al., 2008) in this class use specific path exploration strategies that enable generation of test data that cover deep internal parts of a program, which are difficult to cover otherwise. Techniques in this class can fail to discover program behavior that a systematic (but inefficient) technique can discover because of their use of heuristics.

The fifth class consists of a technique presented by Anand et al. (2007). Unlike all aforementioned techniques that aim to reduce the number of program paths to symbolically execute, this technique aims to reduce the overhead incurred in symbolic execution of each path. The technique uses static analysis to identify only those parts of a program that affect computation of path constraints. Based on that information the technique instruments the program such that the overhead of symbolic execution is not incurred for the other parts of the program.

3.3.2. Techniques for path divergence problem

It is not possible to devise a technique that can entirely eliminate the manual effort required to implement a symbolic execution system that can compute precise path constraints for large, real world programs. Some manual effort is indispensable. However, two existing techniques aim to reduce the manual effort. Godefroid and Taly (2012) proposed a technique that automatically synthesizes functions corresponding to each instruction that is to be symbolically executed such that those functions update the program's symbolic state as per the semantics of corresponding instructions. Anand and Harrold (2011) proposed a technique that can reduce the manual effort that is required to model parts of a program that cannot be symbolically executed (e.g., native methods in Java). Instead of asking the user to provide models for all problematic parts of a program, their technique automatically identifies only those parts that in fact introduce imprecision during symbolic execution, and then asks the user to specify models for only those parts.

3.3.3. Techniques for complex constraints problem

Techniques that address the problem of solving complex constraints can be classified into two classes. The first class consists of two techniques. The first technique, referred to as *dynamic symbolic execution* (Godefroid et al., 2005) or *concolic execution* (Sen et al., 2005). Using this technique, the program is executed normally along the path for which the path constraint is to be computed with some program inputs that cause the program to take that path. That path is also symbolically executed, and if the path constraint becomes too complex to solve, it is simplified by replacing symbolic values with concrete values from normal execution. In the second technique, Pasareanu et al. (2011) also proposed to use concrete values to simplify complex constraints. However, unlike Dynamic Symbolic Execution, they do not use concrete values obtained from normal execution, but instead they identify a solvable part of the complex constraint that can be solved, and use concrete solutions of the solvable part to simplify the complex constraint.

Techniques (Borges et al., 2012; Souza and Borges, 2011; Lakhotia et al., 2010) in the second class model the problem of finding solutions of a constraint over *N* variables as a search problem in a *N*-dimensional space. The goal of the search to find a point in the *N*-dimensional space such that the coordinates of the point represent one solution of the constraint. These techniques use meta-heuristic search methods (Glover and Kochenberger, 2003) to solve such search problems. The advantage of using meta-heuristic methods is that those methods can naturally handle constraints over floating point variables and constraints involving arbitrary mathematical functions such as `sin` and `log`. The limitations of such techniques arise because of the incompleteness of the meta-heuristic search methods that may fail to find solutions to a constraint, even if it is satisfiable.

3.4. Conclusion on symbolic execution

Symbolic execution differs from other techniques for automatic test-generation in its use of program analysis and constraint solvers. However, it can be used in combination with those other techniques (e.g., search-based testing). An extensive body of prior research has demonstrated the benefits of symbolic execution in automatic test-data generation. More research is needed to improve the technique's usefulness on real-world programs. The fundamental problems that the technique suffers from are a long-standing open problem. Thus, future research, in addition to devising more effective general solutions for these problems, should also leverage domain-specific (e.g., testing of smart-phone software) or problem-specific (e.g., test-data generation for fault localization) knowledge to alleviate these problems.

4. Test data generation in model-based testing

By *Wolfgang Grieskamp*

Model-based testing (MBT) is a light-weight formal method which uses models of software systems for the derivation of test suites. In contrast to traditional formal methods, which aim at verifying programs against formal models, MBT aims at gathering insights in the correctness of a program using often incomplete test approaches. The technology gained relevance in practice since around the beginning of 2000. At the point of this writing, in 2011, significant industry applications exist and commercial grade tools are available, as well as many articles are submitted to conferences and workshops. The area is diverse and difficult to navigate.

This section attempts to give a survey of the foundations, tools, and applications of MBT. Its goal is to provide the reader with inspiration to read further. Focus is put on behavioral, sometimes also called functional, black-box testing, which tests a program w.r.t. its observable input/output behavior. While there are many other approaches which can be called MBT (stochastic, structural/architectural, white-box, etc.), including them is out of scope for this survey. Historical context is tried to be preserved: even if newer work exists, we try to cite the older one first.

4.1. Introduction to model-based testing

One can identify three main schools in MBT: axiomatic approaches, finite state machine (FSM) approaches, and labeled transition system (LTS) approaches. Before digging deeper into those, some general independent notions are introduced. In *behavioral MBT*, the *system-under-test* (SUT) is given as a black box which accepts *inputs* and produces *outputs*. The SUT has an internal state which changes as it processes inputs and produces output. The model describes possible input/output sequences on a chosen level of abstraction, and is linked to the implementation by a *conformance relation*. A *test selection* algorithm derives test cases from the model by choosing a finite subset from the potentially infinite set of sequences specified by the model, using a *testing criterion* based on a *test hypothesis* justifying the adequateness of the selection. Test selection may happen by generating test suites in a suitable language ahead of test execution time, called *offline test selection*, or maybe intervened with test execution, called *online test selection*.

4.1.1. Axiomatic approaches

Axiomatic foundations of MBT are based on some form of logic calculus. Gaudel summarizes some of the earliest work going back to the 70 ties in her seminal paper from 1995 (Gaudel, 1995). Gaudel's paper also gives a framework for MBT based on *algebraic specification* (see e.g., Ehrig and Mahr, 1985), resulting from a decade of work in the area, dating back as early as 1986 (Bougé et al., 1986). In summary, given a conditional equation like $p(x) \rightarrow f(g(x), a) = h(x)$, where f, g, and h are functions of the SUT, a is a constant, p a specified predicate, and x a variable, the objective is to find assignments to x such that the given equality is sufficiently tested. Gaudel et.al developed various notions of test hypotheses, notably *regularity* and *uniformity*. Under the regularity hypothesis all possible values for x up to certain complexity n are considered to provide sufficient test coverage. Under the uniformity hypothesis, a single value per class of input is considered sufficient. In Bougé et al. (1986) the authors use logic programming techniques (see e.g., Sterling and Shapiro, 1994) to find those values. In the essence, p is broken down into its disjunctive normal form (DNF), where each member represents an atom indicating a value for x. The algebraic approach can only test a single function or sequence, and in its pure form it is not of practical relevance today; however, it was groundbreaking at its time, and the DNF method of test selection is mixed into various more recent approaches.

Dick and Faivre (1993) a test selection method for VDM (Plat and Larsen, 1992) models based on pre- and post-conditions. The basic idea of using the DNF for deriving inputs is extended for generating sequences as follows. A state machine is constructed where each state represents one of the conjunctions of the DNF of pre- and post conditions of the functions of the model. A transition is drawn between two states S_1 and S_2, labeled with a function call $f(x)$, if there exists an x such that $S_1 \Rightarrow pre(f(x))$ and $post(f(x)) \Rightarrow S_2$. On this state machine, test selection techniques can be applied as described in Section 4.1.2. A theorem prover is required to prove the above implications, which is of course undecidable for non-trivial domains of x, but heuristics can be used as well to achieve practical results.

The approach of Dick and Faivre was applied and extended in Helke et al. (1997). Also, Legeard et al. (2002) is related to it, as well as (Kuliamin et al., 2003). Today, the work of Brucker and Wolff (2012) is closest to it. These authors use the higher-order logic theorem prover system Isabelle (Paulson, 1994) in which they encoded formalisms for modeling stateless and state-based systems, and implemented numerous test derivation strategies based on according test hypotheses. In this approach, the test hypotheses are an explicit part of the proof obligations, leading to a framework in which interactive theorem proving and testing are seamlessly combined.

Models which use pre- and post-conditions can be also instrumented for MBT using random input generation, filtered via the pre-condition, instead of using some form of theorem proving/constraint resolution. Tools like QuickCheck (Claessen and Hughes, 2000) and others are used successfully in practice applying this approach.

4.1.2. FSM approaches

The finite state machine (FSM) approach to MBT was initially driven by problems arising in functional testing of hardware circuits. The theory has later been adapted to the context of communication protocols, where FSMs have been used for a long time to reason about behavior. A survey of the area has been given by Lee and Yannakakis (1996).

In the FSM approach the model is formalized by a *Mealy machine*, where inputs and outputs are paired on each transition. Test selection derives sequences from that machine using some coverage criteria. Most FSM approaches only deal with deterministic FSMs, which is considered a restriction if one has to deal with reactive or under-specified systems.

Test selection from FSMs has been extensively researched. One of the subjects of this work is discovering assumptions on the model or SUT which would make the testing exhaustive. Even though equivalence between two given FSMs is decidable, the SUT, considered itself as an FSM, is an 'unknown' black-box, only exposed by its I/O behavior. One can easily see that whatever FSM the tester supposes the SUT to be, in the next step it can behave different. However, completeness can be achieved if the number of states of the SUT FSM has a known maximum, as Chow (1978) (see also Vasilevskii, 1973; Lee and Yannakakis, 1996). A lot of work in the 80s and 90s is about optimizing the number of tests regards length, overlap, and other goals, resulting for example in the Transition-Tour method (Naito and Tsunoyama, 1981) or the Unique-Input–Output method (Aho et al., 1988).

Many practical FSM based MBT tools do not aim at completeness. They use structural coverage criteria, like transition coverage, state coverage, path coverage, etc. as a test selection strategy (Offutt and Abdurazik, 1999; Friedman et al., 2002). A good overview of the different coverage criteria for FSMs is found, among others, in Legeard's and Utting's seminal text book *practical model-based testing* (Utting and Legeard, 2007).

Various refinements have been proposed for the FSM approach, and problems have been studied based on it. Huo and Petrenko investigated the implications of the SUT using queues for buffering inputs and outputs (Huo and Petrenko, 2005). This work is important for practical applications as the assumption of input and output enabledness, which assumes that the SUT machine can accept every input at any time (resp. the model machine/test case every output), is not realistic in real-world test setups. Hierons and Ural (2008) and Hierons (2010) have investigated *distributed testing*. Hierons (2010) showed that when testing from an FSM it is undecidable whether there is a strategy for each local tester that is guaranteed to force the SUT into a particular model state.

FSMs are not expressive enough to model real software systems. Therefore, most practical approaches use *extended finite state machines* (EFSM). Those augment the control state of an FSM with data state variables and data parameters for inputs and outputs. EFSMs are described usually by state transition rules, consisting of a guard (a predicate over state variables and parameters), and an update on the state variables which is performed when the rule is taken, which happens if the guard evaluates to true in a given state. In practice, many people just say 'state machine' when in fact they mean an EFSM. A typical instance of an EFSM is a statechart. In a proper EFSM, the domains from which data is drawn are finite, and therefore the EFSM can be unfolded into an FSM, making the foundations of FSMs available for EFSMs. However, this has its practical limitations, as the size of the expanded FSM can be easily astronomic. A different approach than unfolding is using symbolic computation on the data domains, applying constraint resolution or theorem proving techniques.

4.1.3. LTS approaches

Labeled transition systems (LTS) are a common formalism for describing the operational semantics of process algebra. They have also been used for the foundations of MBT. An early annotated bibliography is found in Brinksma and Tretmans (2000).

Tretmans (1996) and consolidated the theory in Tretmans (2008). IOCO stands for input/output conformance, and defines a relation which describes conformance of a SUT w.r.t. a model. Tretmans starts from traditional LTS systems which consist of a set of states, a set of labels, a transition relation, and an initial state. He partitions the labels into inputs, outputs, and a symbol for *quiescence*, a special output. The approach assumes the SUT to be an *input enabled* LTS which accepts every input in every state. If the SUT is not naturally input enabled, it is usually made so by wrapping it in a test adapter. Quiescence on the SUT represents the situation that the system is waiting for input, not producing any output by its own, which is in practice often implemented observing timeouts.

As well known, an LTS spawns traces (sequence of labels). Because of non-determinism in the LTS, the same trace may lead to different states in the LTS. The IOCO relation essentially states that a SUT conforms to the model if for every suspension trace of the model, the union of the outputs of all reached states is a superset of the union of the outputs in the according trace of the SUT. Hereby, a suspension trace is a trace which may, in addition to regular labels, contain the quiescence label. Thus the model has 'foreseen' all the outputs the SUT can produce. IOCO is capable of describing *internal* non-determinism in model and SUT, which distinguishes it from most other approaches. There are numerous extensions of IOCO, among those real-time extensions (Nielsen and Skou, 2003; Larsen et al., 2004) and extensions for symbolic LTS (Frantzen et al., 2004; Jeannet et al., 2005) using parameterized labels and guards. The effect of distributed systems has been investigated for IOCO in Hierons et al. (2008).

An alternative approach to the IOCO conformance relation is *alternating simulation* (Alur et al., 1998) in the framework of *interface automata* (IA) (de Alfaro and Henzinger, 2001). While originally not developed for MBT, IA have been applied to this problem by Microsoft Research for the Spec Explorer MBT tool since 2004 (Campbell et al., 2005) (long version in, Veanes et al., 2008). Here, to be conforming, in a given state the SUT must accept every input from the model, and the model must accept every output from the SUT. This makes testing conformance a two-player game, where inputs are the moves of the tests generated from the model, with the objective to discover faults, and outputs are the moves of the SUT with the objective to hide faults. In contrast to IOCO, IA does not require input completeness of the SUT, and treats model and SUT symmetrically; however, it can only deal with *external non-determinism*, which can be resolved in the step it occurs (though extensions waiving this restriction are found in Aarts and Vaandrager, 2010). The IA approach to conformance has been refined for symbolic transition systems by Grieskamp et al. (2006a), which describes the underlying foundations of Microsoft's Spec Explorer tool. Veanes and Bjørner have provided a comparison between IOCO and IA in Veanes and Bjorner (2010), which essentially shows equivalence when symbolic IA's are used.

Both IOCO and IA do not prescribe test selection strategies, but only a conformance relation. Test selection has been implemented on top of those frameworks. For IOCO, test selection based on coverage criteria has been investigated in Groz et al. (1996), based on metrics in Feijs et al. (2002), and based on test purposes in Jard and Jéron (2005). For IA, test selection based on state partitioning has been described in Grieskamp et al. (2002), based on graph traversal and coverage in Nachmanson et al. (2004), and based on model slicing by model composition in Grieskamp et al. (2006a), Grieskamp and Kicillof (2006), Veanes et al. (2007). Test selection in combination with combinatorial parameter selection is described in Grieskamp et al. (2009). Most of these techniques can be equally applied to online testing (i.e. during the actual test execution) or to offline testing (ahead of test execution). Algorithms for offline test generation are generally more sophisticated as they cannot rely on feedback from the actual SUT execution, and typically use state space exploration engines, some of them off-the-shelf model checkers (e.g. Ernits et al., 2006), others specialized engines (e.g. Jard and Jéron, 2005; Grieskamp et al., 2006b).

In contrast to FSM approaches, even for a finite transition system test selection may not be able to achieve state coverage if non-determinism is present. That is because some of the transitions are controlled by the SUT, which may behave 'demonic' regarding its choices, always doing what the strategy does not expect. A 'winning strategy' for a finite LTS exists if whatever choice the SUT does, every state can be reached. However, in practice, models with non-determinism which guarantee a winning strategy are rare. If the SUT can be expected to be 'fair' regarding its non-determinism, this does not cause a problem, as the same test only needs to be repeated often enough.

4.2. Modeling notations

A variety of notations are in use for describing models for MBT. Notations can be generally partitioned into *scenario-oriented*, *state-oriented*, and *process-oriented*. Whether they are textual or graphical (as in UML) is a cross-cut concern to this. A recent standard produced by ETSI, to which various tool providers contributed, collected a number of tool-independent general requirements on notations for MBT following this taxonomy (ETSI, 2011b).

4.2.1. Scenario-oriented notations

Scenario-oriented notations, also called interaction-oriented notations, directly describe input/output sequences between the SUT and its environment as they are visible from the viewpoint of an outside observer ('gods view'). They are most commonly based on some variation of message sequence chart (Dan and Hierons, 2011), activity chart (flow chart) (Hartmann et al., 2005; Wieczorek and Stefanescu, 2010), or use case diagram (Kaplan et al., 2008), though textual variations have also been proposed (Grieskamp et al., 2004; Katara and Kervinen, 2006).

Test selection from scenario-based notations is generally simpler than from the other notational styles, because by nature the scenario is already close to a test case. However, scenarios may still need processing for test selection, as input parameters need to be concretized, and choices and loops need expansion. Most existing tools use special approaches and not any of the axiomatic, FSM, or LTS based ones. However, in Grieskamp et al. (2006a) it is shown how scenarios can be indeed broken down to an LTS-based framework in which they behave similar as other input notations and are amenable for model composition.

4.2.2. State-oriented notations

State-oriented notations describe the SUT by its reaction on an input or output in a given state. As a result the models state is evolved and, in case of Mealy machine approaches, an output maybe produced. State-oriented notations can be given in diagrammatic form (typically, statecharts) (Offutt and Abdurazik, 1999; Bouquet et al., 2007; Huima, 2007) or in textual form (guarded update rules in a programming language, or pre/post conditions on inputs, outputs and model state) (Dick and Faivre, 1993; Kuliamin et al., 2003; Grieskamp et al., 2003, 2011b). They can be mapped to axiomatic, FSM, or LTS based approaches, and can describe deterministic or non-deterministic SUTs.

4.2.3. Process-oriented notations

Process-oriented notations describe the SUT in a procedural style, where inputs and outputs are received and sent as messages on communication channels. Process-algebraic languages like LOTOS are in use (Tretmans and Brinksma, 2003), as well as programming languages which embed communication channel primitives (Huima, 2007). Process-oriented notations naturally map to the LTS approach.

4.3. Tools

There are many MBT tools around, some of them result of research experiments, some of them used internally by an enterprise and not available to the public, and others which are commercially available. Hartman gave an early overview in 2002 (Hartman, 2002), and Legeard and Utting included one in their book from 2007 (Utting and Legeard, 2007). Since then, the landscape has changed, and new tools are on the market, whereas others are not longer actively developed. It would be impossible to capture the entire market given more than a short-lived temporary snapshot. Here, three commercial grade tools are sketched which are each on the market for nearly ten years and are actively developed. The reader is encouraged to do own research on tools to evaluate which fit for a given application; the selection given here is not fully representative. For an alternative to commercial tools, one might also check out Binder's recent overview (Binder, 2012) of open-source or open-binary tools.

4.3.1. Conformiq Designer

The Conformiq Designer[10] (Huima, 2007) (formerly called QTronic) has been around since 2006. Developed originally for the purpose of protocol testing, the tool can be used to model a variety of systems. It is based on UML statecharts as a modeling notation, with Java as the action language. Models can also be written entirely in Java. The tool supports composition of models from multiple components, and timeouts for dealing with real-time. It does not support non-determinism.

Conformiq Designer has its own internal foundational approach, which is probably closest to LTS. A symbolic exploration algorithm is at the heart of the test selection procedure. The tool can be fed with a desired coverage goal (in terms of requirements, diagram structure, or others) and will continue exploration until this goal is reached. Requirements are annotated in the model and represent execution points or transitions which have been reached.

Conformiq Designer can generate test suites in various formats, including common programming languages, TTCN-3, and manual test instructions. The generated test cases can be previewed as message sequence charts by the tool.

Conformiq Designer has so far been used in industrial projects in telecommunication, enterprise IT, automotive, industrial automation, banking, defense and medical application domains.

4.3.2. Smartesting CertifyIt

The Smartesting CertifyIt tool[11] (Legeard and Utting, 2010) (formerly called Smartesting Test Designer) is around since 2002. Coming out of Legeard's, Utting's, and others work around testing from B specifications (Abrial, 1996), the current instance of the tool is based on UML statecharts, but also supports BPMN scenario-oriented models, and pre/post-condition style models using UML's constraint language OCL.

Smartesting CertifyIt uses a combination of constraint solving, proof and symbolic execution technologies for test generation. Test selection can be based on numerous criteria, including requirements coverage and structural coverage like transition coverage. The tools also supports test selection based on scenarios (in BPMN), similar as Spec Explorer does. The tool generates test suites for offline testing in numerous industry standard formats, and supports traceability back to the model. Non-determinism is not supported.

CertifyIt is dedicated to IT applications, secure electronic transactions and packaged applications such as SAP or Oracle E-Business Suite.

4.3.3. Spec Explorer

Microsoft Spec Explorer is around since 2002. The current major version, called Spec Explorer 2010,[12] is the third incarnation of this tool family. Developed in 2006 and described in (Grieskamp, 2006; Grieskamp et al., 2011b) it should not be confused with the older version which is described in Veanes et al. (2008). Spec Explorer was developed at Microsoft Research, which makes it in contrast to the other commercial tools highly documented via research papers, and moved in 2007 into a production environment mainly for its application in Microsoft's Protocol Documentation Program. The tool is integrated into Visual Studio and shipped as a free extension for VS.

The tool is intentionally language agnostic but based on the.Net framework. However, the main notations used for modeling are a combination of guarded-update rules written in C# and scenarios written in a language called Cord (Grieskamp and Kicillof, 2006). The tool supports annotation of requirements, and (via Cord) ways for composing models. Composing a state-based model written in C# with a scenario expressing a test purpose defines a slice of the potentially infinite state model, and is one of the ways how engineers can influence test selection.

The underlying approach of Spec Explorer are interface automata (IA), thus it is an LTS approach supporting (external) non-determinism. Spec Explorer uses a symbolic exploration engine (Grieskamp et al., 2006b) which postpones expansion of parameters until the end of rule execution, allowing to select parameters dependent on path conditions. The tool supports online and offline testing, with offline testing generating C# unit tests. Offline test selection is split into two phases: first the model is mapped into a finite IA, then traversal techniques are run on that IA to achieve a form of transition coverage.

Spec Explorer has been applied, amongst various internal Microsoft projects, in arguably the largest industry application for MBT up to now, a 350 person year project to test the Microsoft protocol documentation against the protocol implementations (Grieskamp et al., 2011b). In course of this project, the efficiency of MBT could be systematically compared to traditional test automation, measuring an improvement of around 40%, in terms of the effort of testing a requirement end-to-end (i.e. from the initial test planning to test execution). Details are found in Grieskamp et al. (2011b).

4.4. Conclusion on model-based testing

At the ETSI MBT user conference in Berlin in October 2011,[13] over 100 participants from 40 different companies came together, discussing application experience and tool support. Many of the academic conferences where general test automation work is published (like ICST, ICTSS (formerly TestCom/FATES), ASE, ISSTA, ISSRE, AST, etc.) regularly see a significant share of papers around MBT. Two Dagstuhl seminars have been conducted around the subject since 2004 (Brinksma et al., 2005; Grieskamp et al., 2011); the report from the last event lists some of the open problems in the area. These all document a lively research community and very promising application area.

5. Test data generation in combinatorial testing

By *Myra B. Cohen*[14]

Combinatorial testing has become a common technique in the software tester's toolbox. In combinatorial testing, the focus is on

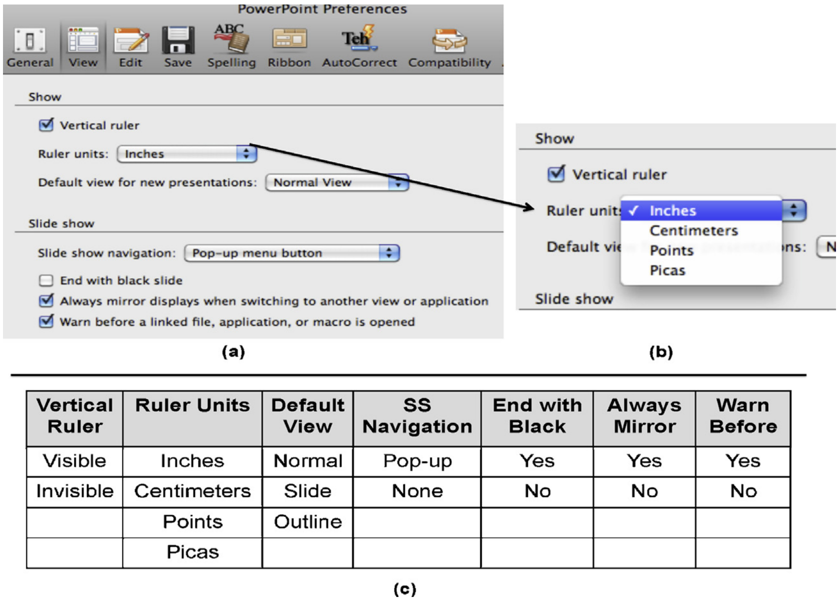
[10] http://www.conformiq.com.

[11] http://www.smartesting.com.

[12] http://www.specexplorer.net.

[13] http://www.model-based-testing.de/mbtuc11/.

[14] *Acknowledgements*: The authors would like to thank the anonymous reviewers for their helpful comments. This work is supported in part by the National Science Foundation through award CCF-0747009 and by the Air Force Office of Scientific Research through awards FA9550-09-1-0129 and FA9550-10-1-0406. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the position or policy of NSF or the AFOSR.



Show

☑ Vertical ruler

Ruler units: Inches

Default view for new presentations: Normal View

Slide show

Slide show navigation: Pop-up menu button

□ End with black slide

☑ Always mirror displays when switching to another view or application

☑ Warn before a linked file, application, or macro is opened

(a)

Show

☑ Vertical ruler

Ruler units ✓ Inches
Centimeters
Points
Picas

Default vi… ns: N

Slide show

(b)

Vertical Ruler	Ruler Units	Default View	SS Navigation	End with Black	Always Mirror	Warn Before
Visible	Inches	Normal	Pop-up	Yes	Yes	Yes
Invisible	Centimeters	Slide	None	No	No	No
	Points	Outline				
	Picas					

(c)

Fig. 2. Modeling configurations for testing.

selecting a sample of input parameters (or configuration settings), that cover a prescribed subset of combinations of the elements to be tested. The most common manifestation of this sampling is combinatorial interaction testing (CIT), where all t-way combinations of parameter values (or configuration settings) are contained in the sample. In the past few years, the literature on this area of testing has grown considerably, including new techniques to generate CIT samples and applications to novel domains. In this section we present an overview of combinatorial testing, starting at its roots, and provide a summary of the two main directions in which research on CIT has focused – sample generation and its application to different domains of software systems.

5.1. Introduction to combinatorial testing

Throughout the various stages of testing, we rely on heuristics to approximate input coverage and outcomes. Combinatorial testing has risen from this tenet as a technique to sample, in a systematic way, some subset of the input or configuration space. In combinatorial testing, the parameters and their inputs (or configuration options and their settings) are modeled as sets of factors and values; for each factor, f_i, we define a set of values, $\{x_1, x_2, \ldots, x_j\}$, that partition the factor's space. From this model, test cases, or specific program configurations (instances) are generated, selecting a subset (based on some coverage criterion) of the Cartesian product of the values for all factors; a program with five factors, each with three values, has 3^5 or 243 program configurations in total. CIT has traditionally been used as a specification-based, system testing technique to augment other types of testing. It is meant to detect one particular type of fault; those that are due to the interactions of the combinations of inputs or configuration options. For instance, if the aim is to detect faults due to combinations of pairs of configuration options, using combinatorial testing can satisfy this test goal using only eleven configurations.

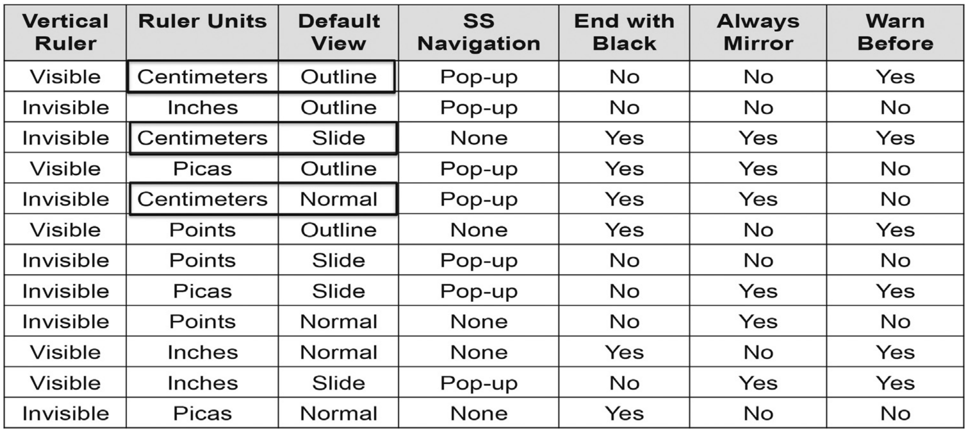
The roots of combinatorial testing come from the field of statistics called design of experiments (Fisher, 1971; Cochran and Cox, 1957). In the 1930s, Fisher described a means to lay out crop experiments that combine independent variables in a systematic way, in order to isolate their impact on the observed outcomes. In 1985, Mandl used these ideas to sample the combinations of parameter values in compiler software through the use of orthogonal latin squares (Mandl, 1985). Around the same time, Ostrand and Balcer (1988) developed the Category Partition Method, and the Test Case Specification Language (TSL), which gives us a way to model the factors and values so that they can be combined. In TSL, test inputs (or configurations) are modeled as categories and partitions, and for each a set of choices are described which are equivalence classes for testing. There are mechanisms to reduce the combinatorial space. Choices can be tagged as *single*, or *error*, or predicates can be defined that describe dependencies such as *requires* between elements. The full Cartesian product, that satisfy these constraints, is then generated.

In 1992 Brownlie et al. (1992) presented an extension of Mandl's work called the Orthogonal Array Testing System, OATS, in which they used *orthogonal arrays* to define the combinations of an AT&T email system; all pairs of factor-values are tested exactly once. The work of Cohen et al. (1997, 1996) leveraged a key insight, that all factor-values must be tested *at least* once, lifting the *exactly-once* restriction of orthogonal arrays. This led to the use of *covering arrays* and the core underpinnings of combinatorial interaction testing (or CIT) as is used today. Out of this work came the Automatic Efficient Test Case Generator (AETG), a greedy algorithm that generates covering array samples and includes both a modeling language and test process (Cohen et al., 1997, 1996).

Over the past several years we have seen a large increase in the number of algorithms for generating CIT samples and new applications that use CIT. We don't attempt to provide a complete survey of CIT (for a recent survey see (Nie and Leung, 2011)), but instead provide a short overview and highlight some key research directions.

5.1.1. Example of CIT

In Fig. 2(a) we show the `View` preferences tab from a version of Microsoft Powerpoint. The user has seven configuration options that they can customize. Some of the configuration options such as *Ruler units* have multiple settings to choose from. We provide an enlargement of this selection menu which offers the user the choice of Inches, Centimeters, Points or Picas in Fig. 2(b). Other options such as *End with black slide* are binary; the user can select or deselect this setting. In total we have seven configuration-options, (*factors*) which we have shown as columns in the table (Fig. 2c). The first factor, *Vertical ruler*, has two possible *values*, the second



Vertical Ruler	Ruler Units	Default View	SS Navigation	End with Black	Always Mirror	Warn Before
Visible	Centimeters	Outline	Pop-up	No	No	Yes
Invisible	Inches	Outline	Pop-up	No	No	No
Invisible	Centimeters	Slide	None	Yes	Yes	Yes
Visible	Picas	Outline	Pop-up	Yes	Yes	No
Invisible	Centimeters	Normal	Pop-up	Yes	Yes	No
Visible	Points	Outline	None	Yes	No	Yes
Invisible	Points	Slide	Pop-up	No	No	No
Invisible	Picas	Slide	Pop-up	No	Yes	Yes
Invisible	Points	Normal	None	No	Yes	No
Visible	Inches	Normal	None	Yes	No	Yes
Visible	Inches	Slide	Pop-up	No	Yes	Yes
Invisible	Picas	Normal	None	Yes	No	No

Fig. 3. 2-Way CIT sample for Fig. 2.

has four values, etc. In total, there are $2 \times 4 \times 3 \times 2^4$ or 384 unique configurations of `View` preferences. In practice this is only a part of the configuration space; if we combine this with the preferences from the `Save` tab we have almost 25,000 configurations, and if we model the entire preference space this becomes intractable; the literature has reported the optimization configuration space of GCC (the GNU Compiler Collection) (Free Software Foundation, 2012), an open source widely used compiler framework, is in the order of 10^{61} (Cohen et al., 2008). If instead we sample the configuration space so that we cover all pairs of combinations of each factor-value we can achieve this using only twelve configurations. We show one such sample in Fig. 3. This sample constitutes a covering array, defined next.

5.1.2. Covering arrays

A *covering array*, $CA(N; t, k, v)$, is an $N \times k$ array on v symbols such that every $N \times t$ sub-array contains all t-tuples from the v symbols *at least* once. In a covering array t is called the *strength* and N is the sample size. The parameter t tells us how strongly to test the combinations of settings. If $t = 2$ we call this *pairwise* CIT. In Fig. 3 all combinations of the factor-value *Ruler Units-Centimeters* have been combined with all values from *Default View*, but we can't guarantee any specific combinations of three (or more) factor-values are covered. We notice that the factors have different numbers of values. We define a more general structure next.

A *mixed level* covering array, $MCA(N; t, (w_1 w_2 \ldots w_k))$, is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^{k} w_i$, and each column i $(1 \leq i \leq k)$ contains only elements from a set of size w_i and the rows of each $N \times t$ subarray cover all t-tuples of values from the t columns at least once. We typically use a shorthand notation to equal consecutive entries in $(w_i : i \leq 1 \leq k)$. For example three consecutive entries each equal to 2 can be written as 2^3. Fig. 3 is a mixed level covering array, with $k = 7$ and $t = 2$; an $MCA(12; 2, 2^1 4^1 3^1 2^4)$. When we use the term covering array for CIT we usually are referring to a mixed level covering array. The minimization of the covering array sample size (i.e. N) has been a focus of much work in this domain. While it is has been shown that the upper bound on the size of a covering array grows logarithmically in k (Cohen et al., 1997), it is non-trivial to construct CIT samples. Some instances of CIT construction have been shown to be NP Hard, such as finding the minimal size given forbidden constraints (Colbourn et al., 2004; Bryce and Colbourn, 2006).

5.2. Research directions

The research on CIT has branched in two main directions. The first direction develops methods and algorithms to generate CIT samples, while the second direction refines CIT to work in novel application domains. We highlight work in each direction next.

5.2.1. CIT generation

There is a long history of research on the mathematics of covering arrays which we do not attempt to cover, but instead refer the reader to two excellent surveys, one by Colbourn (2004) and another by Hartman and Raskin (2004). Mathematical techniques may be probabilistic (i.e. they do not construct, but prove existence of arrays), or provide direct constructions. Constructions are deterministic and produce arrays of known sizes, but are not as general as heuristic methods, known only on a limited subset of possible parameter settings for t, k and v. C. Colbourn maintains a website with references of known array sizes and associated techniques (Colbourn, 2012).

Heuristic techniques to generate CIT samples have dominated the literature on CIT. The first algorithms were greedy, of which there are two primary types. One class follows an AETG-like mechanism (Cohen et al., 1997). Examples are the Test Case Generator (TCG) (Tung and Aldiwan, 2000), Deterministic Density Algorithm (DDA) (Colbourn et al., 2004), and PICT (Czerwonka, 2006). One test case at a time is added (the test case that increases the number of combinations that are covered the most) and within each row, the algorithms greedily chose the next best factor-value for inclusion. The heuristics used to select the next best factor-value differentiate these algorithms. Some newer variants of this algorithm, use meta-heuristic search techniques (such as genetic algorithms, tabu search, or ant colony optimization) to optimize selection of the factors with-in a row (i.e. that part is no longer greedy), but still retain the one row at a time greedy step (Bryce and Colbourn, 2007). A similar type of greedy algorithm, the Constrained Array Test System (CATS) (Sherwood, 1994), was proposed around the same time as AETG. It too selects test cases one at a time, but full test cases are enumerated and then re-ordered, so that the earliest test cases provide the greatest value towards covering uncovered factor-values. A second class of greedy algorithms are of the form used in the In Parameter Order Algorithm (Tai and Lei, 2002; Lei et al., 2008). In IPO, the algorithm begins with some number of factors $k' \ll k$ and expands the size of the covering array horizontally (by increasing k') and vertically (by adding new test cases to the sample to complete coverage if needed).

Meta-heuristic search techniques have been used to generate CIT samples, working on the entire sample at once. Some size of N, is chosen as a start. Guided by a fitness function, and a stochastic process to transition through the search space, different solutions are tried and evaluated, until a covering array either is found for that N or a time-out has occurred. N is then adjusted in subsequent iterations until the smallest array is produced. Simulated annealing, has been the most widely discussed meta-heuristic algorithm for constructing covering arrays (Cohen et al., 2003, 2003; Garvin et al., 2011). Other approaches include genetic algorithms (Stardom, 2001), tabu search (Nurmela, 2004) and constraint solvers (Hnich et al., 2006).

Another primary direction for CIT research, has been generating samples that consider dependencies (or constraints) between factor-values, called Constrained CIT (CCIT). For example, in Fig. 2, suppose that the Vertical Ruler is not visible when using Picas because this functionality is unsupported. We do not want to include this combination in our samples (it will render the configuration infeasible). In CCIT satisfiability solvers have been used to aid in the evaluation of legal combinations of factor-values. In the work of Cohen et al., standard meta-heuristic search algorithms and greedy algorithms for CIT have been tightly-interwoven with SAT solvers to achieve this goal. Calvagna and Gargantini (2009, 2010) and Grieskamp et al. (2009) use solvers as the primary method to generate the samples. And more recently Binary Decision Diagrams BDD's have been employed as a way to generate CCIT samples (Segall et al., 2011).

5.2.2. CIT application domains

The original uses of CIT was for *test case* generation where the factors and their values are system inputs or parameters and each row of the covering array is a test case (Brownlie et al., 1992; Cohen et al., 1997, 1996; Dalal et al., 1998; Dunietz et al., 1997). CIT has also been applied to test protocol conformance (Grieskamp et al., 2009; Burroughs et al., 1994).

More recent work samples configurations to be tested (under which many test cases will be run) (Qu et al., 2008; Yilmaz et al., 2006; Kuhn et al., 2004; Kuhn and Okun, 2006; Fouché et al., 2009; Dumlu et al., 2011). One type of configurable system, a software product line, has been an area of active research on CIT (Cohen et al., 2006; Perrouin et al., 2010). Software product lines are systems of program families that have a well managed asset base and feature model; from which one can derive a CIT model (Cohen et al., 2006; Clements and Northrop, 2001). McGregor (2001) first suggested that products in an SPL could be sampled for testing using CIT. Cohen et al. (2006) described a mapping from a feature model to a relational mode. Perrouin et al. (2010) have more tightly integrated construction with the feature model. There has been recent work that explores the use of CIT when testing sequences (Kuhn et al., 2012; Yuan et al., 2011). In traditional CIT, there is no notion of order (any two columns of the covering array can be swapped as long as a mapping is maintained for the concrete test cases to be applied). In sequence-based CIT, each factor becomes a location within a sequence, and the values within each factor are repeated at each location. This has been used to test graphical user interfaces (GUIs) (Yuan et al., 2011) and devices (Kuhn et al., 2012).

CIT has also been used to *characterize* the configuration option combinations that are likely to be the cause failures through the use of classification trees (Yilmaz et al., 2006; Fouché et al., 2009). More recently, Colbourn and McClary (2008) present special types of covering arrays, called locating and detecting arrays for the purpose of directly isolating the causes. Another new direction for CIT is to tune the test process and coverage through the use of *variable strength covering arrays* (Cohen et al., 2003), *prioritization* (Bryce and Colbourn, 2006; Qu et al., 2008, 2007) and *incremental covering arrays* (Fouché et al., 2009), where the size of t can vary, the order of testing is prescribed, or we generate increasingly stronger t by re-using existing tests from lower strength t when moving to higher t.

5.3. Conclusion on combinatorial testing

In this section we have presented an overview of combinatorial testing, defined the primary mathematical object on which this research is based and presented some research directions that are being pursued. This is a promising area of research for automated software test generation, with opportunities to enhance new domains of its application. Fruitful future research directions for generating CIT samples includes automated model extraction, adapting to model evolution, and developing techniques that reuse or share information between different test runs. In addition to applying CIT to novel application domains, an area of potential for improvement in this direction is the combination of program analysis techniques with CIT to refine the sample space, and to target specific interactions at the code (as opposed to only the specification) level.

6. Test data generation by adaptive random testing

By *Tsong Yueh Chen*

Empirical studies have shown that failure-causing inputs tend to form contiguous failure regions: consequently, non-failure-causing inputs should also form contiguous non-failure regions. Therefore, if previously executed test cases have not revealed a failure, new test cases should be far away from the already executed non-failure-causing test cases. Hence, test cases should be evenly spread across the input domain. It is this concept of even-spreading of test cases across the input domain which forms the basic intuition for adaptive random testing, a family of test case selection methods designed to enhance the failure detection effectiveness of random testing by enforcing an even spread of randomly generated test cases across the input domain. This section provides a brief report on the state-of-the-art of adaptive random testing.

6.1. Introduction to adaptive random testing

Random testing (RT) is one of the most fundamental and most popular testing methods. It is simple in concept, easy to implement, and can be used on its own or as a component of many other testing methods. It may be the only practically feasible technique if the specifications are incomplete and the source code is unavailable. Furthermore, it is one of the few testing techniques whose fault detection capability can be theoretically analysed. Adaptive random testing (ART) (Chen et al., 2010, 2004) has been proposed as an enhancement to RT. Several empirical studies have shown that failure-causing inputs tend to form contiguous failure regions, hence non-failure-causing inputs should also form contiguous non-failure regions (White and Cohen, 1980). Therefore, if previous test cases have not revealed a failure, new test cases should be far away from the already executed non-failure-causing test cases. Hence, test cases should be evenly spread across the input domain. It is this concept of even spreading of test cases across the input domain, which forms the basic intuition of ART. Anti-random testing (Malaiya, 1995) also aims at even spreading of test cases across the input domain. However, a fundamental difference is that ART is a nondeterministic method and anti-random testing is in essence a deterministic method with the exception of the first test case which is randomly chosen. Another difference is that anti-random testing requires testers to specify the number of test cases in advance, whereas there is no such a constraint for ART.

To facilitate discussion, it is first necessary to define some terminology. By failure rate, we mean the ratio of the number (or size) of failure-causing inputs to the number (or size) of the set of all possible inputs (hereafter referred to as the input domain). By failure patterns, we mean the distributions and geometry of the failure-causing inputs. By efficiency, we refer to the computation time required, with lower computation time indicating higher efficiency. Strictly speaking, efficiency should also include memory, but memory will not be considered in this section due to space limitations and the lack of implementation details such as the data structures used. By effectiveness, we refer to the fault detection capability which can be measured by the effectiveness metrics including P-measure, E-measure, F-measure, etc. The F-measure is defined as the expected number of test cases required to detect the first failure; the P-measure is defined as the probability of detecting at least one failure; and the E-measure is defined as the expected number of failures detected. A set of test cases is assumed when using P-measure or E-measure.

RT is a popular testing method and ART has been originally proposed as an enhanced alternate to RT. This section will focus on the state-of-the-art of ART from the perspective of using RT as a baseline. We will only compare ART to RT, and not compare ART to other testing methods. We are interested in the problem that *when RT has been chosen as a viable testing method for a system, is it worthwhile to use ART instead*?

As a reminder to avoid any confusion and misunderstanding, cost-effectiveness in this section refers to the fault detection capability achieved for the resources spent. Some researchers (such as Arcuri and Briand, 2011) used the term “effective” where we use “cost-effective”, therefore, when comparing across papers, such a difference in the meanings should be noted. We decide to deal with effectiveness and efficiency separately in this section because such an approach will give us a better picture about which aspect or direction shall be improved.

6.2. Various ART algorithms

Various approaches have been identified to implement the concept of even spreading of test cases across the input domain. As a consequence, a number of different ART algorithms have been developed (Chan et al., 2006; Chen et al., 2004, 2009, 2004; Ciupa et al., 2008; Lin et al., 2009; Liu et al., 2011; Mayer, 2005; Shahbazi et al., 2013; Tappenden and Miller, 2009). The major approaches include:

1. Selection of the best candidate as the next test case from a set of candidates. This approach first generates a set of random inputs as candidates from which the best candidate, as defined against set criteria, is selected as the next test case.
2. Exclusion. In each round of test case generation, this approach first defines an exclusion region around each already executed test case. Random inputs are generated one by one until one input is outside all exclusion regions of the already executed test cases, and then this input is selected as the next test case.
3. Partitioning. This approach uses the information about the location of already executed test cases to divide the input domain into partitions, and then to identify a partition as a designated region from which the next test case will be generated.
4. Test profiles. Instead of using a uniform test profile as normally adopted by RT, this approach uses a specially designed test profile which is able to achieve an even spreading of test cases over the input domain. Dynamic adjustment of the test profile during testing is required in this approach.
5. Metric-driven. Distribution metrics, such as discrepancy and dispersion, are normally used to measure the degree of even distribution for a set of points. Instead of being used as a measurement metric, this approach uses the distribution metrics as selection criteria to select new test cases such that a more even distribution of the resultant test cases could be obtained.

The above is not an exhaustive list, but rather gives some of the most popular approaches. Furthermore, it should be noted that for each approach, different methods can be used to achieve an even spreading of test cases. Therefore, many ART algorithms have been developed. For example, the most popular algorithm taking the first approach is the Fixed-Sized-Candidate-Set ART (hereafter referred to as FSCS-ART) (Chen et al., 2004) in which a fixed-size candidate set of random inputs is first generated whenever a new test case is needed. For each candidate set, a selection criterion is applied to select the best candidate as the next test case. Adopted selection criteria include maxi-min, maxi-maxi, maxi-sum, etc. For maxi-min, the distance (or dissimilarity in the case of non-numeric inputs) between each candidate and its nearest already executed test case is first calculated. The candidate with the largest such distance is then selected as the next test case. For maxi-sum, the distances between each candidate and all the already executed test cases are first summed. The candidate with the highest such sum is then selected as the next test case. Intuitively speaking, inputs near the boundaries of the input domain will have a higher probability of being selected as test cases when maxi-min is used instead of maxi-sum. In other words, different ART algorithms have different effectiveness performance, efficiency performance and characteristics which in turn give rise to different favourable and unfavourable conditions for their applications.

As FSCS-ART is the first published ART algorithm and has been the most cited ART algorithm since the inception of ART, some previous studies have treated FSCS-ART and ART as equivalent or exchangeable. We would like to emphasize that FSCS-ART is only one of the many members of the family of ART algorithms, and FSCS-ART is not equivalent to *ART which refers to the family of testing methods in which test cases are random and evenly spread across the input domain*. Obviously, the strengths and weaknesses of a particular ART algorithm for a specific type of software are not necessarily valid nor expected to be similar for other ART algorithms.

For numeric input domains, the *distance* (or *dissimilarity*) metric used to measure “far apart” is easily and naturally defined. However, the choice of a distance metric for non-numeric input domains may not be straightforward. We have proposed a generic distance metric based on the concept of categories and choices (Kuo, 2006; Merkel, 2005). Ciupa et al. (2008) have proposed a specific distance metric for object-oriented software. Tappenden and Miller (in press) have proposed a specific distance metric for cookies collection testing of web applications. It is understood that there are currently investigations into the application of ART in input domains involving strings, trees, finite state machines, etc.

6.3. Effectiveness

In the studies of ART, the adopted effectiveness metrics include F-measure, P-measure and the time to detect the first failure. Obviously, different effectiveness metrics have different strengths and weaknesses, and there is no single best effectiveness metric. Also, it is common that a testing method is better than another testing method with respect to one effectiveness metric, but worse if measured against another metric. Therefore, a metric may be appropriate in one scenario but inappropriate in another. The selection of an appropriate metric is in itself a challenging problem.

The F-measure has been the most frequently used metric to compare the effectiveness of ART and RT. Chen et al. (2004) compared RT and ART using 12 open source numerical analysis programs written in C++, with seeded faults. For three out of these 12 programs, there was no significant difference between the F-measures of ART and RT; for one program, the F-measure of ART was about 90% of the F-measure of RT; and for the remaining eight programs, the F-measure of ART was between 50% and 75% of the F-measure of RT. Ciupa et al. (2008) compared RT and ART using real-life faulty versions of object-oriented programs selected from the EiffleBase Library. Their results showed that the average F-measure of ART was about 19% of the F-measure of RT. Lin et al. (2009) compared RT and their ART using six open source Java software artifacts with seeded faults. The average F-measures for ART and RT were 142 and 1246, respectively. In the study conducted by Zhou et al. (2013), they used four numerical programs written in C from GNU Scientific Library with seeded faults. They used two ART algorithms, and hence there were eight comparison scenarios between ART and RT. Their results showed that in one of the eight comparison scenarios, the F-measure of ART was about 107% of the F-measure for RT, and for the remaining seven out of the eight comparison scenarios, the F-measure of ART was between 25% and 75% of the F-measure for RT. Tappenden and Miller (2009) used simulations to compare ART and RT. They observed that "All of the testing methods (eAR, FSCS, RRT, and the Sobol sequence) s-significantly outperformed RT with respect to the block failure pattern.", "With respect to the strip pattern, [. . .] ART methods s-significantly outperformed RT for all failure rates.", and "Point pattern simulation yielded results similar to the strip pattern; ART methods performed slightly better, and not worse than RT, with s-significant effect sizes ranging from $r = 0.009$ to $r = 0.030$.". Arcuri and Briand have observed that for one mutant of a program, the F-measures for RT and ART were 72,237 and 56,382, respectively (see Table 4 in Arcuri and Briand, 2011). So, there has been a general consensus that ART is better than RT with respect to the F-measure. The superiority of ART over RT with respect to the F-measure is intuitively expected as the concept of even spreading of test cases originates from the objective of hitting the contiguous failure regions using fewer test cases. Furthermore, the F-measure improvement is quite significant and is in no way diminished by any potential challenge to previous experiments' validity.

A recent analytical study (Chen and Merkel, 2008) proves that even if we know the shapes, sizes and orientations of the failure regions, but not their locations, it is impossible to have a strategy that guarantees the detection of a failure with its F-measure being less than half of the F-measure for RT. In other words, 50% of RT's F-measure is an upper bound of the effectiveness improvement that we can possibly achieve when we know the sizes, shapes and orientations of the failure regions (in reality, we are not able to know the sizes, shapes and orientations of the failure regions prior to testing). Since ART never uses nor assumes such information, ART shall not have a lower F-measure than the optimal strategy which is designed according to such information. When interpreted with the simulation and experimental results of the F-measures of ART, this theoretical result implies a rather surprising but most welcome conclusion that ART is close to the optimal strategy, and that the upper bound is indeed a tight bound. As shown in the proofs and examples in Chen and Merkel (2008), technically speaking, the optimal strategy is to construct a grid of test cases according to the sizes, shapes and orientations of the failure regions. An even spreading of test cases is a lightweight approach to implement an approximation to such a grid, and hence ART can be viewed as a lightweight approach to implementing the optimal strategy. In other words, it seems unlikely that there are other testing techniques which can use significantly fewer test cases than ART to detect the first failure, unless there is access to the information about the locations of the failure regions, which is usually not possible. An immediate conclusion is that future research shall be focused on either how to use the information of the location of failure-causing input to develop new testing strategies that can outperform ART (for example, see Zhou et al., 2013), or how to improve the efficiency of ART by reducing the cost for test case generation (for example, see Chan et al., 2006; Chen et al., 2006).

As pointed out by Arcuri and Briand (2011), previous empirical studies or simulations only involved failure rates larger than 10^{-6}, and hence were perhaps not comprehensive enough. Therefore, it is worthwhile to conduct further experiments to verify whether or not ART does still have a lower F-measure than RT for extremely low failure rates. However, both the proofs of the theoretical analysis (Chen and Merkel, 2008) and the results of a simulation study (Chen et al., 2007) about the impact of the geometry of failure regions show that the fewer the distinct failure regions are, the better performance of F-measure ART has. This implies that ART will have a better F-measure performance than RT at later stages (relatively fewer distinct failure regions and lower failure rates) than at earlier stages (relatively more distinct failure regions and higher failure rates) of software development. As far as we know, all existing ART algorithms tend to achieve increasingly even spreading with more test cases. In other words, lower failure rates are actually favourable scenarios for ART with respect to F-measures. We are not aware of any work showing that at lower failure rates, RT has a lower F-measure than ART. Obviously, it is worthwhile to see more experimental data on this aspect. In summary, there is no challenge to the fact that ART has a significantly lower F-measure than that of RT.

Chen et al. (2006) have used simulations to compare the P-measure between ART and RT, and have found that ART outperforms RT. Recently, Shahbazi et al. (2013) have proposed an innovative approach to use the concept of Centroidal Voronoi Tessellations to evenly spread random test cases over the input domain and developed RT-RBCVT and RT-RBCVT-Fast methods, which also belong to the ART approach as they evenly spread random test cases over the input domain. A very important result is that their RT-RBCVT-Fast method is of the same order of computational complexity as RT. In the application of their methods, the size of a set of test cases is defined first. In both their simulations, which used various types of failure patterns, and their empirical analysis, which used mutants, the RT-RBCVT method consistently demonstrated higher P-measures than RT, as reported that "RBCVT is significantly superior to all approaches for the block pattern in the simulation framework at all failure rates as well as the studied mutants at all test set sizes. Although the magnitude of improvement in testing effectiveness results is higher for the block pattern compared to the point pattern, the results demonstrate statistically significant improvement in the point pattern".

In Arcuri and Briand's empirical analysis using mutants (Arcuri and Briand, 2011), it was reported that "Although the results in Fig. 3 suggest that ART can be better than random testing (the odds ratios are lower than 0.5 in most cases), the results in Fig. 4 show that ART is still very unlikely to detect faults. In most of the cases the P-measure is lower than 0.01, i.e., ART would have less than 1% chance of finding failure". By definition, the P-measure for RT or ART is a function of the size of the test set. Furthermore, the value of the P-measure for RT or ART will be increased if the size of the test set is increased. As an example for illustration, consider a program with failure rate of 0.0001. On average, RT needs to use 10,000 test cases to detect a failure. In other words, the P-measure for RT using 10,000 test cases will be very close to 1. Thus, if the size of the test set is chosen to be 100,000, then the P-measure for both RT and ART will be even closer to 1. On the other hand, if the size of the test set is chosen to be 10, then obviously the P-measures for RT and ART will be close to 0. Furthermore, for the scenarios of using 100,000 and 10 test cases, the differences between their P-measures for RT and ART, if any, are likely to be very small. Therefore, when comparing the P-measures of RT and ART, a full range of the sizes of the test set should be used in order to get a comprehensive and meaningful comparison. However, this problem may not occur when RT (or ART) is compared to other testing strategies which require a particular number of test cases for a specific program. Suppose that a program has k paths. For path coverage testing, a set of k test cases is required. In this case, when P-measure is used as the effectiveness metric to compare RT and path coverage testing, it is not only meaningful but also fair that a random test set of k elements for RT should be compared with a path coverage test set of k elements, irrespectively of the failure rate for the program under test. This specific value of k is not arbitrarily chosen and there is a justification. However, when P-measure is used to compare RT and ART, an immediate question is what should be the appropriate size of the test set used, simply because the size of the test set has significant impact on the returned values of the P-measure. Hence, the F-measure is more appropriate than the P-measure in the comparison of RT and ART.

Compared to the F-measure, the P-measure has been less often used in evaluating ART. Nevertheless, all studies have consistently shown that ART outperforms RT with respect to the P-measure. This universal observation is consistent with an analytical result that the P-measure of the Proportional Sampling Strategy (PSS) is not lower than that of RT (Chen et al., 2001). PSS is a test case selection strategy for subdomain testing, which allocates a number of test cases to a subdomain in proportion to the subdomain's size. PSS is in fact an ART algorithm, using a partitioning approach. Thus, it is intuitively appealing to expect the P-measure of other ART algorithms to be not lower than that of RT. Furthermore, it is important to note that PSS has been proved to be a necessary and sufficient condition for partition testing to outperform RT with respect to the P-measure. With regard to the E-measure, PSS and RT have been theoretically proved to have the same E-measure (Chen et al., 2001).

In addition to the F-measure and P-measure, the amount of time to taken to detect the first failure (or fault) has been used as a performance metric by Ciupa et al. (2008) in their investigation using real-life faulty programs, and Lin et al. (2009) in their investigation using open source Java programs with seeded faults. Strictly speaking, the measurement of the time to detect the first failure is better interpreted as a cost-effectiveness metric rather than an effectiveness metric. Ciupa et al. found that ART required an average of 1.6 times the amount of time required by RT to detect the first failure, but Lin et al. found that ART required an average of 0.13 times the amount. The apparently different observations are understandable, because this metric depends on the characteristics of the programs, which are different in these two studies. Ciupa et al. have proposed using a clustering technique to reduce the distance computation overheads, with the basic idea being to only compute the distances to the cluster centres, rather than to each of the already executed test cases. Their preliminary study shows "an average improvement of the time to first fault over ARTOO of 25% at no cost in terms of faults found". Since the time to detect first failure for ART is 1.6 times that for RT in their study, a 25% improvement is in fact a very encouraging result that justifies more research being conducted in this area.

In summary, both simulations as well as empirical analyses using real-life faulty programs and mutants have consistently shown that ART outperforms RT with respect to the P-measure and the F-measure, but ART may still use less time to detect the first failure than RT, despite the fact that ART requires more computation time for test generation because of the additional task of evenly spreading the test cases across the input domain.

6.4. Efficiency

Compared to RT, ART algorithms are expected to use more computation time and memory because of the additional task of evenly spreading the test cases (Chen et al., 2004). As explained above, we will only consider computation time in this section. Obviously, different ART algorithms have different orders of complexity for the generation of test case, ranging from the highest order of n^2, $n \log n$, to n, where n denotes the number of already executed test cases. Intuitively speaking, algorithms with higher orders of complexity for test case generation are expected to have better even spreading of test cases, and hence are expected to have a better fault detection capability. Such an expectation normally occurs but not always. Since different methods are used to achieve an even spread of test cases, we have different ART algorithms, each of which has its own strengths and weaknesses, as well as favourable and unfavourable conditions for its application. For example, a conventional implementation of FSCS-ART has n^2 complexity. Therefore, it would be inappropriate to apply FSCS-ART to programs with very small failure rates, unless the program execution time and test setup time are considerably larger than the time required by FSCS-ART to generate a test case.

There exist general techniques that are applicable to most of the ART algorithms to reduce their cost of test case generation. As discussed above, Ciupa et al. (2008) have proposed to use the technique of clustering to reduce the distance computation overheads, and have obtained positive results. Another technique is called mirroring (Chen et al., 2006). Its basic idea is to divide the input domain into k partitions, of which one partition is referred to as the source partition and the other partitions are referred to as the mirror partitions. ART is applied only on the source partition to generate test cases within it. With simple mappings, the test cases generated in the source partition are mapped into the mirror partitions to generate new test cases within themselves. For FSCS-ART, its test case generation overheads can be effectively reduced by a factor of $(1/k)^2$. Another technique is called forgetting (or aging) (Chan et al., 2006). Instead of using all already executed test cases to determine the next test case, we use only a portion or a constant number of already executed test cases to determine the next test case. If the option of a constant number of already executed test cases is used, the order of complexity for generating the next test case will be independent of n. Generally speaking, when a reduction method for distance computations is applied, the reduction may bring in new kinds of overheads and may be at the expense of the fault detection capability. However, such a deterioration of fault detection capability does not always occur. One instance is observed by Ciupa et al. (2008) in their investigation on the technique of clustering that "an average improvement of the time to first fault over ARTOO of 25% at no cost in terms of faults found". Apart from Ciupa et al.'s study which involved real-life faulty programs, other investigations into the impact of reduction techniques have used simulations and mutants. Therefore, it is important to have further experiments using real-life faulty programs to investigate the impact of these general reduction techniques on the efficiency of ART.

6.5. Frameworks for cost effective application of ART

After discussing the effectiveness and efficiency of ART, we are now ready to discuss how to apply ART in practice. As a reminder, this section only compares ART and RT. Therefore, our objective is to determine how to use ART as a cost-effective alternate to RT when RT has been chosen as a viable testing method to test a system.

There are two possible application scenarios, one with a fixed number of test cases (equivalently, a limited resource) and the other without such a constraint. For the first scenario, our recommendation is to use Tappenden and Miller's RT-RBCVT-Fast method, because it has the same order of computational complexity as RT but it has a higher P-measure than RT.

Now, let us consider the other scenario. Since different programs may have different execution times, different test setup times and different test case generation times, obviously an ART algorithm may be cost-effective for one program, but not cost-effective for another, as compared with RT. Therefore, for a given program, it is a challenging problem to select a cost-effective ART algorithm, let alone to select the most cost-effective ART algorithm. Let us explain the difficulty by first visiting the problem of selecting a cost-effective sorting algorithm for a given file. Similar to ART, there are many sorting algorithms which have different orders of computation complexity, and favourable and unfavourable conditions for their applications. However, when we are going to do sorting, normally we have some information about the file to be sorted. Such information will help us to choose an appropriate sorting algorithm. For example, if the file is known to be nearly ordered, we would use bubble-sort instead of quick-sort; if the file is known to be random or nearly random, then quick-sort rather than bubble-sort should be used. Similarly, if we know the execution time, test setup time and failure rate of the software under test, we would be able to use the information about test case generation complexity for an ART algorithm to determine whether it is more cost-effective than RT to test the software. But, in reality, though we may know some information about the execution time and test setup time of the software under test, we do not know its failure rate. In other words, we do not have sufficient information to determine whether an ART algorithm is more cost-effective than RT for this given software. Then, does it mean that ART is practically useless, as we are not able to determine whether an ART algorithm is more cost-effective than RT for the given program? The answer is no. Some potential frameworks for cost-effective applications of ART are presented as follows.

A simple framework is to successively apply RT-RBCVT-Fast with test sets of sizes n_1, n_2, n_3, . . ., n_k. An estimation of the failure rate should first be made, which is then used to determine the value of n_1. An over estimation of the failure rate is recommended. As an example for illustration, if some available information (such as, past testing history and program size) suggests that the failure rate for the software under test is not less than 0.001, then we may assume the estimated failure rate to be 0.01. With a failure rate of 0.01, RT needs to use on average 100 test cases in order to detect failure. Thus, we may set $n_1 = 100$. If RT-RBCVT-Fast cannot find failure with a random set of 100 test cases, then one of the many possible ways is to set $n_2, n_3, \cdots$ and n_k such that $n_2 = 2n_1$, $n_3 = 2n_2$, $\cdots$, $n_k = 2n_{k-1}$, and successively apply RT-RBCVT-Fast using test sets of sizes $n_2, n_3, \ldots$ and n_k.

Another framework can be built upon the technique of adaptive testing. By adaptive testing, it basically means that in the process of software testing, a testing method may be replaced by another testing method in response to some on-line collected feedbacks (Cai, 2002). Suppose we are required to test a program P. Let E denote its average execution time and $G(n)$ denote its generation time for the nth test case. Since the F-measure of ART shall not be less than half of the F-measure of RT (as proved analytically (Chen and Merkel, 2008)), obviously it is only worthwhile to continue ART if $G(n)$ is less than E. Suppose that we have ART-A, ART-B and ART-C whose test case generation complexities are of the orders of n^2, $n \log n$, and n, respectively. As a note, normally the higher the order of complexity is, the higher the fault detection effectiveness an ART algorithm has. Here, we assume ART-A performs better than ART-B which in turn performs better than ART-C, with respect to F-measure or P-measure. We shall start testing with ART-A first until we reach $G(n) \geq E$. Then, we use ART-B until we reach a new n' such that $G(n') \geq E$. Then, we use ART-C until we reach a new n'' such that that $G(n'') \geq E$. By then, we may apply RT-RBCV-Fast successively with different test sets as explained in the immediately preceding paragraph, or use the general reduction techniques to keep the cost of generating a new test case steady (such as, the technique of forgetting using a constant number of already executed test cases in distance computation).

The above sketches of the frameworks are very high level, but they are conceptually feasible. Obviously, a lot of technical details need to be defined in the actual application. Also, new types of overheads may be introduced. Therefore, the cost-effectiveness of these proposed frameworks needs to be validated by experimental analysis involving real-life programs.

6.6. Applications of ART and tools

Compared to RT, ART has been applied to fewer real-life programs. However, we expect a growth in the application of ART to real-life programs, because more and more efficient ART algorithms have been emerging. Chen et al. (2004) have applied ART to testing open source numerical analysis programs written in C++, using mutants. Ciupa et al. (2008) have compared RT and ART using real-life faulty versions of object-oriented programs selected from EiffelBase Library. Their experimental results showed that ART used significantly fewer test cases to detect the first failure than RT (0.19 times), but ART used more time to detect the first failure than RT (1.6 times). Also observed is that ART revealed faults that RT did not reveal in the same allocated time. Iqbal et al. (2011) have compared RT, ART and search-based testing using Genetic Algorithms and the (1 + 1) Evolutionary Algorithm. Their study included a real-life real-time embedded system which was a seismic system. They have observed that ART was the best performer, but "there is a 9% probability of wrongly claiming that ART is better than RT if that is actually not the case" (Iqbal et al., 2011). Hemmati et al. (2010, 2011) have used ART to test a safety monitoring component of a safety-critical control system written in C++and a core component of a video-conference system written in C. Tappenden and Miller (in press) have used Evolutionary ART in their cookie collection testing of six open-source web applications. Faults were detected in five out of the six web applications. Their results showed that Evolutionary ART was an effective testing method. Lin et al. (2009) have used six open source Java software artifacts with manually seeded faults in evaluating their ART method. Five of the six subjects were from Apache common library and the other was Siena.

With regard to the automated ART tools, AutoTest (Ciupa et al., 2008) supports ART for object-oriented programs. A very good design feature of AutoTest is to use ARTOO as a plug-in strategy for input generation. With such a feature, other ART algorithms could be easily supported by AutoTest. In the study by Shahbazi et al. (2013), programs were developed to support FSCS-ART, restricted random testing (ART by Exclusion), Evolutionary ART, RBCVT and RBCVT-Fast.[15] Iqbal et al. (2011) have developed an automated test framework which can support ART to test real-time embedded systems, and the framework has been found effective. Lin et al. (2009) have developed the tool ARTGen that supports the testing of Java programs using a divergence-oriented approach to ART. The majority of ART algorithms consist of two processes, namely, a process for random generation of inputs and a process to ensure an even spreading of test cases across the input domain. In fact, the majority of the processes of ensuring an even spreading of test cases are quite simple. Therefore, it is not difficult to build one's own ART tool on top of a random test case generator. In other words, it should be quite straightforward to plugin ART's even spreading component into an existing RT tool.

6.7. Future challenges and work

1. Majority of the previous ART investigations involved simulations and failure rates greater than 10^{-6}. Therefore, it is important to have more investigations which will involve lower failure rates using real-life faulty programs or mutants. Empirical analysis is required to validate the conjecture that lower failure rate is a favourable condition for ART with respect to F-measure, as discussed above.
2. As explained above, further research should be focused on reducing the cost of test case generation for ART algorithms in order to

[15] The software is available at URL: http://www.steam.ualberta.ca/main/Papers/RBCVT.

enhance their cost-effectiveness. So far, the investigated reduction techniques include clustering, mirroring and forgetting. With the exception of Ciupa et al.'s preliminary investigation on the technique of clustering, which involved real-life faulty programs, other investigations on the general reduction techniques only involved simulations and mutants. Though Ciupa et al.'s results are very positive for the technique of clustering, the impact of the other reduction techniques should be further analysed using more real-life programs.

3. The proposed framework for how to apply ART using the technique of adaptive testing has been briefly outlined above. The sketches of the framework are very high level but the framework is conceptually feasible. A lot of technical details need to be defined in actual application, such as, how to deal with the already executed test cases after switching from one ART algorithm to another ART algorithm. Obviously, its feasibility needs to be validated by experimental analysis involving real-life programs.
4. Failure patterns provide valuable information to help us to develop new and effective test case selection strategies. We coined this area as *failure-based testing*. The domain test strategy proposed by White and Cohen (1980) is not only a fault-based testing strategy as stated by them, but also a failure-based testing technique. It is indeed the first failure-based testing technique. Its target is the domain fault which gives rise to a specific failure pattern in the input domain. The concept of geometry is applied to the resultant failure pattern to design test cases that guarantee to detect the relevant fault. ART is a failure-based testing method using the most primitive information of the contiguity of failure-causing inputs. Since failure patterns also have other information, there is still great potential benefit to be gained from the use of this other information to develop new testing strategies. The search-based testing community has developed many searching techniques, some of which may become, or be adapted to become, new search techniques for failure regions.

6.8. Conclusion on adaptive random testing

All existing investigations have consistently shown that ART outperforms RT with respect to the F-measure and P-measure. These investigations include simulations and experimental analysis using both mutants and real-life faulty programs. The positive results of these simulation and experimental investigations are consistent with the interpretations and results of the theoretical analysis. Though the scope of existing investigations may not be considered sufficiently comprehensive (Arcuri and Briand, 2011), the superiority of ART over RT with respect to the F-measure and P-measure is unlikely to be challenged. Nevertheless, more comprehensive experiments on the F-measure and P-measure of RT and ART will still be worthwhile.

Compared to RT, ART has the additional task of evenly spreading the test cases. Therefore, ART will unavoidably consume more computation time and memory than RT. Hence, it is understandable that an ART algorithm is not necessarily more cost-effective than RT for a given program, despite the fact that it is superior to RT with respect to the F-measure and the P-measure. On the other hand, with respect to the metric of time required to find the first failure or fault, RT is not always superior to ART even though ART incurs more computation time than RT. This is also understandable because the characteristics of the programs under test will affect the value of this metric. Obviously, the characteristics of the program under test must be considered when determining whether an ART algorithm will be more cost-effective than RT.

Since its inception, the ART research has been focused on the development of new algorithms which would have a lower F-measure. As explained above, a recent analytical result shows that ART is in fact a lightweight approach to implementing the optimal strategy which is essentially equivalent to constructing a grid of test cases according to the sizes, shapes and orientations of the failure regions. An immediate conclusion is that ART has great potential to be a cost-effective alternate to RT. Attention should then be shifted from the effectiveness to the efficiency of ART, that is, to the reduction in time and space complexity, in order to make ART a cost-effective alternate to RT. Conceptually speaking, reduction in the computation and memory overheads are possible but may be at the expense of the degree of even spreading which in turn may affect the effectiveness. However, the recently published method of RT-RBCVT-Fast shows that ART can indeed serve as a cost-effective alternate to RT, because it has the same order of computational complexity as RT.

7. Test data generation in search-based software testing

By *Mark Harman*, *Phil McMinn*, *John Clark* and *Edmund K. Burke* [16]

> *Search based software testing (SBST) is a branch of search based software engineering (SBSE), in which search algorithms are used to automate the process of finding test data that maximises the achievement of test goals, while minimising testing costs. There has been much interest in SBST, leading to several recent surveys. This paper presents some emerging challenges and open problems for the development of this exciting research agenda. These include hybrids of SBST and DSE (Dynamic Symbolic Execution); optimizing to best handle demands of the oracle; co-evolving tests and software simultaneously; "hyper-heuristics" where SBST may be integrated into other aspects of SBSE, e.g. requirements prioritisation; and optimization of failures for ease of debugging.*

7.1. Introduction to search-based testing

As this paper shows, the problem of automatically generating test inputs is hard. For example, even the most basic activities, such as seeking to cover a branch in the code involve reachability questions that are known to be undecidable in general (Weyuker, 1979). The testing community has therefore focused on techniques that seek to identify test sets that cover near optimal sets of branches in reasonable time. Many of these techniques are covered in other sections of this paper.

This section is concerned with the area of Search-Based Software Testing (SBST). SBST is a branch of Search-Based Software Engineering (SBSE) (Harman and Jones, 2001), in which optimisation algorithms are used to automate the search for test data that maximises the achievement of test goals, while minimising testing costs. There has been much interest in SBST, leading to several recent surveys. This section presents some emerging challenges and open problems for the development of this exciting research agenda.

SBST is the process of generating test cases (or often the inputs of test cases) using search-based algorithms, guided by a fitness function that captures the current test objective. SBST has been applied to a wide variety of testing goals including structural (Harman and McMinn, 2010; McMinn et al., 2012; Michael et al., 2001; Tonella, 2004), functional (Wegener and Bühler, 2004), non-functional (Wegener and Grochtmann, 1998) and state-based properties (Derderian et al., 2006).

Search-based approaches have been developed to address a wide and diverse range of domains, including testing approaches

[16] *Acknowledgements:* The authors would like to thank Yue Jia for Fig. 4.

based on agents (Nguyen et al., 2009), aspects (Harman et al., 2009), interactions (Cohen et al., 2003), integration (Colanzi et al., 2011; Briand et al., 2002), mutation (Harman et al., 2011; Zhan and Clark, 2005), regression (Walcott et al., 2006; Yoo et al., 2009), stress (Grosso et al., 2005) and web applications (Alshahwan and Harman, 2011).

In all approaches to SBST, the primary concern is to define a fitness function (or set of fitness functions) that capture the test objectives. The fitness function is used to guide an algorithm, which searches the space of test inputs to find those that meet the test objectives. Because any test objective can, in principle, be re-cast as a fitness function, the approach is highly generic and therefore widely applicable (as the foregoing list of testing applications demonstrates). There are many different search-based algorithms to choose from, though much of the literature has tended to focus on evolutionary algorithms (Harman, 2011).

There are several surveys of these aspects of SBST (Afzal et al., 2009; Ali et al., 2010; Harman et al., 2009; McMinn, 2004, 2011). In these surveys the reader can find more detailed treatments of the work on SBST for non-functional properties (Afzal et al., 2009), Empirical evidence regarding SBST (Ali et al., 2010), as well as overviews of techniques (Harman et al., 2009; McMinn, 2004; McMinn et al., 2012). Therefore, in this section, we do not seek to provide yet another 'overview' of SBST. Rather, we focus on some of the exciting and challenging avenues that lie ahead for future work in this rapidly growing research and practitioner community.

In describing these future directions we seek to consider work which is already underway as well as more 'blue skies' directions for open challenges that could yield major breakthroughs. For example, we consider work underway on co-evolution and management of oracle cost as well as work on hybridising SBST with other test data generation techniques, such as Dynamic Symbolic Execution (DSE), a topic covered in more detail elsewhere in this paper. The oracle problem is important because the automation of testing requires automation of the checking of outputs as well as the generation of inputs. Co-evolution is interesting and important because it fits so well the way in which the testing process operates, as we shall see.

In all these emerging areas we can expect more work in the immediate future. We also consider open challenges such as the problem of migrating from generation of test cases to generation of testing strategies using search and optimising the insight that can be gained from SBST.

7.2. Hybrids of SBST and DSE

The Dynamic Symbolic Execution (DSE) approach (Godefroid et al., 2005) to handling dynamic data structures proved very effective, leading Lakhotia et al. (2008) to incorporate DSE's approach into SBST. Conversely, SBST handles floating point computation well, while DSE is limited by the power of the constraint solvers available (which typically cannot solve floating point constraints efficiently). Naturally, therefore, one might expect that the advantage of combining the two techniques, will be that the strengths of one can overcome the shortcomings of the other.

This led several authors to develop approaches to augment DSE with search-based approaches to solving floating point computations. Lakhotia et al. (2010) used a local search to augment the Pex DSE-based testing tool from Microsoft, while Souza and Borges (2011) augmented 'standard' constraint solving with a Particle Swarm optimiser to improve the performance of Symbolic PathFinder.

The first authors to propose a combination of SBST and DSE to produce a hybrid were Inkumsah and Xie (2007) who introduced the EVACON framework, which composes the two approaches, reporting the first results for a combined DSE/SBST approach. The AUSTIN search-based software testing tool also provides hybrid capabilities, for which results have been reported to compare SBST and DSE for 'out of the box' test data generation (Lakhotia et al., 2010).

Baars et al. (2011) developed a new approach to SBST, in which symbolic execution is integrated into the search by augmenting the fitness function used to guide SBST. In a way, this work 'does for SBST, what DSE does for constraint based testing'. However, the differences between SBST and DSE mean that the modifications to symbolic execution, required to make it scalable, are also different. That is, whereas DSE performs a complete symbolic execution using concrete values, Baars et al. use a purely symbolic execution with no concrete values, but apply it only to local regions of code to improve the fitness function.

Harman et al. (2011) also combined DSE and SBST to produce the first approach to test data generation for strong (and higher order) mutation testing. They use DSE to achieve weak mutation adequacy, following a variant of the approach of Liu et al. (2006) and Papadakis and Malevris (2010). This approach generates constraints, the satisfaction of which yields weak mutation adequacy. To extend this to strong mutation adequacy Harman et al. search the space of additional conjuncts for constraints to augment those that extend weak to strong. The fitness function seeks maximal control flow disruption in order to increase the likelihood of strong adequacy.

As this recent work demonstrates, there is much activity at the interface between SBST and DSE that is producing a form of 'crossover and mutation' of the two approaches. Because of their complementary nature we can expect to see more work on the combination of these two promising test data generation techniques. The proliferation of publicly available tools that support both approaches, and hybrids thereof, creates a rich infrastructure from which future research can draw.

7.3. Handling the oracle

Testing involves examining the behaviour of a system in order to discover potential faults. Determining the desired correct behaviour for a given input is called the Oracle Problem. Manual testing is expensive and time consuming, particularly because of the manual effort devoted to solving the oracle problem. This is the Human Oracle Cost. We need to develop SBST algorithms and methods that automatically generate test inputs that reduce Human Oracle Cost, thereby significantly reducing the overall cost of testing. We also need search-based techniques that can help to generate test oracles as well as test cases (Fraser and Zeller, 2010).

Of course, the cost of generating test inputs by hand is high. This has driven the growth of the search-based testing research area. Indeed, over 340 papers have been published in the area according to a recent survey (Harman et al., 2009). However, despite this considerable publication output, there is very little work on either reducing the Oracle Cost (Harman et al., 2010; McMinn et al., 2010) or using SBST to generate oracles (Fraser and Zeller, 2010).

Most previous work concentrates on the problem of searching for good test inputs, but it does *not* address the equally important problem of reducing the *cost* of checking the output produced in response to the inputs generated. The current state of the art in SBST thus addresses only the *benefit* half of the testing problem: that of generating inputs that meet the testing criterion. It fails to address the other half of the problem: the *cost* of checking the output produced.

This is simply not realistic for many testing applications; it assumes that all that matters to the tester is the achievement of the highest possible coverage, at any cost. However, a tester might, for example, prefer an approach that achieves 85% coverage with 30 test cases, over an alternative that achieves 90% coverage with

1000 test cases. Or, the tester may prefer a test suite that lowers the comprehension cost of individual test cases, by minimising test case verbosity (Fraser and Zeller, 2011) and maximising readability (McMinn et al., 2012).

Fortunately the SBST paradigm naturally generalises to multi-objective optimisation formulations, thereby allowing us to develop techniques that balance the multiple objectives of cost and benefit. If we can measure the oracle cost then we can make it a minimisation objective in our SBST test data generation approach. This will mean that all approaches to test data generation will be naturally multi-objective (Harman et al., 2007), because they will need to balance cost and benefit. This is a natural step forward for SBST, since testing is all about balancing cost and benefit. We know that exhaustive testing is impossible so we wish to achieve maximum benefit (ultimately fault finding, measured through surrogates such as coverage) for minimum cost (ultimately monetary, measured through surrogates such as effort and time).

7.4. Opportunities for co-evolution

With co-evolutionary computation, two or more populations evolve simultaneously, using possibly different fitness functions. In competitive co-evolution the idea is to capture a predator–prey model of evolution, in which both evolving populations are stimulated to evolve to better solutions. In co-operative co-evolution, the idea is to symbiotically co-evolve several populations, each relying on the other to work in concert as part of a larger system that contains them.

Adamopoulos et al. (2004) were the first to suggest the application of co-evolution for SBST, arguing that this could be used to evolve sets of mutants and sets of test cases, where the test cases act as predators and the mutants as their prey. For testing, the competitive model has hitherto proved best suited, since test cases make natural predators.

Various forms of testing and bug fixing have been attacked using competitive co-evolution. Arcuri et al. also used co-evolution to evolve programs and their test data from specifications (Arcuri, 2008; Arcuri and Yao, 2007) using co-evolution. Arcuri (2008) and Arcuri and Yao (2008) also developed a co-evolutionary model for bug fixing, in which one population essentially seeks out patches that are able to pass test cases, while test cases can be produced from an oracle in an attempt to find the shortcomings of a current population of proposed patches. In this way the patch is the prey, while the test cases, once again, act as predators.

We can expect to see work in which various software artefacts and their test cases are co-evolved. The test cases will be evolved to find counter examples that demonstrate that the artefacts being evolved are not yet optimal. The artefacts can then be re-evolved to produce new versions for which the counter examples no longer apply. Through iteration of this co-evolutionary cycle we seek to obtain, not only high quality artefacts, but also test cases that can be used to demonstrate their effectiveness.

7.5. Hyper-Heuristic Software Engineering

One key question for the SBSE/SBST research agenda is

> “Can we radically increase automation by integrating SBST with other forms of SBSE?”

Consider the two connected trees depicted in Fig. 4. The lower tree is a sub-tree of software engineering defined by the ACM classification system. Existing work on SBSE is currently applied at the leaves of this tree. More work can and will surely be done, across the community, within each of the leaves to better address instances of each software engineering problem. However, such approaches, on their own, can only offer improvement of a narrow set of very similar engineering activities. That is we will have improved test input generation and (separately) improved requirements prioritisation. What we will still need is to find ways to unite the two so that we can have optimised tests generated from requirements, with requirements prioritised in a manner that takes account of testability.

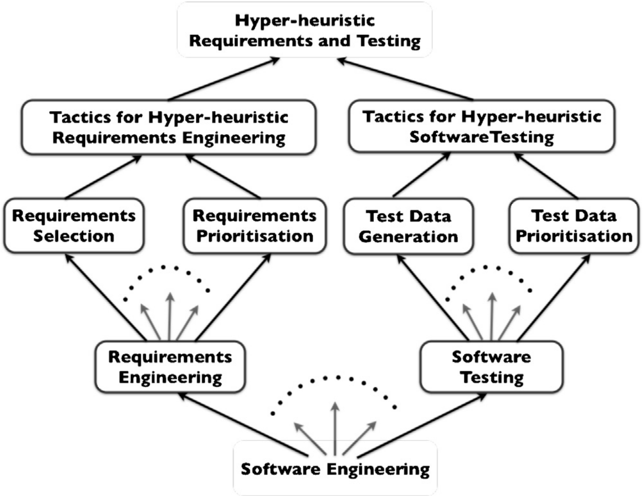

Hyper-heuristic Requirements and Testing
Tactics for Hyper-heuristic Requirements Engineering
Tactics for Hyper-heuristic SoftwareTesting
Requirements Selection
Requirements Prioritisation
Test Data Generation
Test Data Prioritisation
Requirements Engineering
Software Testing
Software Engineering

Fig. 4. Hyper-Heuristic SBSE: using Hyper Heuristics, we may be able to develop tactics an strategies that will unite different software engineering activities with SBST.

To solve this larger software engineering challenge, SBST needs to make a transition from solving instances to automatically finding tactics that solve instances. This will increase the abstraction level at which we apply SBSE, as indicated in the upper tree in Fig. 4, drawing together sets of related software engineering activities. For instance, we shall be able to combine different kinds of test data generation, searching for improved tactics that deploy each to maximise their effectiveness and minimise cost, automatically tailoring the methodology to suit the particular test problem in hand.

In this way, we would be making a leap from tactics that solve classes of problems to strategies that cross the existing software engineering boundaries, as illustrated by the top node of the upper tree in Fig. 4. Ultimately, the goal would be to unify previously poorly connected areas of software engineering activity within a single computational search process. This would be a kind of ‘Hyper-Heuristic Software Engineering’.

As an example of the possibilities for Hyper-Heuristic Software Engineering, suppose we succeed in combining requirements optimisation (Zhang et al., 2008) with SBST. We shall now be able to optimise the selection of requirements based, not only on traditional aspects of SBSE for requirements (customer satisfaction, cost etc.), but also on the implications for regression testing (coverage achievable, test execution time). We would reach the pinnacle (the root) of the upper tree in Fig. 4 with Hyper-Heuristic Requirements and Testing. However, we could go further still in our quest for a unified Hyper-Heuristic Software Engineering.

Suppose we now also manage to draw SBSE for project planning (Antoniol et al., 2011; Chicano and Alba, 2005) into our Hyper-Heuristic Software Engineering framework. We will then have a combined approach to optimised, requirements, project planning and testing. Instead of merely discussing requirements choices devoid of their technical and managerial consequences, we can use our combined automated approach to check the implications of requirement choices on the project plan used to staff the project and implement the requirements. We can also explore implications for regression testing and seek multi-objective solutions that balance the competing objectives of requirement choices, effective implementation plans and efficient and effective regression testing.

Armed with a Hyper-Heuristic Software Engineering tool, decision makers could then enter negotiations armed able to respond in real time to changing requirement choices with analysis and results on the implications for the cost and duration of the project and its testing. This would not be merely 'better requirements engineering' or 'better testing'; it would be a fundamentally different approach to software development in which optimisation would be at the heart; a *lingua franca* within which decisions could be made about requirements, design and testing with detailed investigation of their consequences. Gradually, as Hyper-Heuristic Software Engineering strategies draw in more of the process, the information available would be further enriched, bringing together aspects of marketing, negotiation, customer relations, project management, design and testing.

7.6. Optimising and understanding failures

Some failures are caused by exceptionally long and complex sequences of actions and events. This makes it theoretically hard (and sometimes practically impossible) to find the fault or faults that cause the failure. Therefore, a natural yet challenging and important question to ask is:

"Can we simplify the failure to make it easier to debug?"

In some ways, this problem is related to the oracle cost problem, described in Section 7.3. That is, if we can reduce the cost of understanding the output of a test, then we reduce the oracle cost to the human. On the other hand, there is also a human cost in understanding the input to a test. If a long sequence (and/or a complex sequence) of actions is required to replicate a failure at the developers' site, the engineers may find it too complicated to understand the causes of the failure, and therefore too difficult to find the faults that cause failure.

Suppose we can capture the failing behaviour with an assertion (or some such similar mechanism). Now we can have a fitness function that measures how close a test case comes to causing a failure to manifest itself. This would be one potential fitness function. If we can additionally measure the complexity of a test case then we can seek to minimize this, while maximising similarity to the failure of interest; another multi-objective formulation of test data generation.

A natural starting point for failures, would be 'wrong values in variables', since this would be easy to capture within existing SBST frameworks: fitness computation would be no different to that for branch coverage. One could use a simple testability transformation (Harman, 2008; McMinn et al., 2009) to insert a branch that captures the failing behaviour and seek to cover the branch. A starting point for test complexity would be simply the length of the input sequence required to reveal the fault. The challenge will be in finding supporting fitness functions and ways to smooth an otherwise rather 'spiky' landscape in order to provide guidance to shorter test inputs that manifest the desired failure.

7.7. Conclusion on search-based testing

Search based software testing (SBST) is a branch of Search Based Software Engineering (SBSE) which (re)formulates test objectives as fitness functions to guide automated search procedures. This provides a way to automate test generation for many different forms of testing. The approach is supremely general and astonishingly widely applicable because any test objective that can be measured is a candidate for this transformation into a fitness function. There surely remain many exciting, important and productive test objectives that have yet to be attacked using this SBSE reformulation, thereby providing many fruitful avenues for future work.

8. Conclusion and acknowledgement

This paper presents a survey of some of the most prominent techniques of automated test data generation, including symbolic execution, model-based, combinatorial, adaptive random and search-based testing.

The survey has followed the novel approach of orchestrated surveys. We believe that, by coordinating renowned specialists of carefully selected topics, we are able to balance breadth with depth to produce one overall article of reasonable size.

Editing this paper has been a new experience for the editors. The editors would like to thank the authors of the sections for their participation and excellent work carried out in the project. The editors would also like to express their appreciation to the reviewers of the sections. Their constructive and critical comments have been invaluable to the success of the project. The editors are most grateful to Prof. Hans van Vliet, the Editor-in-Chief of the Journal of Systems and Software, for his support to this project and for the valuable advice and direction he has given to the editors of the paper, as well as for his patience during the long process of the project.

References

Aarts, F., Vaandrager, F.W., 2010. Learning I/O automata. In: Gastin, P., Laroussinie, F. (Eds.), CONCUR. Springer, pp. 71–85.

Abrial, J.R., 1996. The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA.

Adamopoulos, K., Harman, M., Hierons, R.M.,2004. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In: Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO'04). Springer, pp. 1338–1349, LNCS 3103/2004.

Afzal, W., Torkar, R., Feldt, R., 2009. A systematic review of search-based testing for non-functional system properties. Information and Software Technology 51 (6), 957–976.

Aho, A., Dahbura, A., Lee, D., Uyar, M., 1988. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. In: Aggarwal, S., Sabnani, K. (Eds.), Protocol Specification, Testing, and Verification VIII, North-Holland. , pp. 75–86.

de Alfaro, L., Henzinger, T.A., 2001. Interface automata. ESEC/SIGSOFT FSE, 109–120.

Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K., 2010. A systematic review of the application and empirical investigation of search-based test-case generation. IEEE Transactions on Software Engineering 36 (6), 742–762.

Alshahwan, N., Harman, M., 2011. Automated web application testing using search based software engineering. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11), pp. 3–12.

Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y., 1998. Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (Eds.), CONCUR. Springer, pp. 163–178.

Anand, S., Godefroid, P., Tillmann, N.,2008. Demand-driven compositional symbolic execution. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 367–381.

Anand, S., Harrold, M.J., 2011. Heap cloning: enabling dynamic symbolic execution of java programs. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11), pp. 33–42.

Anand, S., Orso, A., Harrold, M.J., 2007. Type-dependence analysis and program transformation for symbolic execution. In: Proceedings of the the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 117–133.

Anand, S., Pasareanu, C.S., Visser, W., 2009. Symbolic execution with abstraction. International Journal on Software Tools for Technology Transfer 11, 53–67.

Antoniol, G., Di Penta, M., Harman, M., 2011. The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study. Software – Practice and Experience 41 (5), 495–519.

Arcuri, A.,2008. On the automation of fixing software bugs. In: Proceedings of the Doctoral Symposium of the IEEE International Conference on Software Engineering (ICSE'08). ACM, pp. 1003–1006.

Arcuri, A., Briand, L., 2011. Adaptive random testing: an illusion of effectiveness? In: Proceedings of the 20th International Symposium on Software Testing and Analysis, pp. 265–275.

Arcuri, A., Yao, X.,2007. Coevolving programs and unit tests from their specification. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07). ACM, pp. 397–400.

Arcuri, A., Yao, X.,2008. A novel co-evolutionary approach to automatic software bug fixing. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC'08). IEEE Computer Society, pp. 162–168.

Baars, A., Harman, M., Hassoun, M., Lakhotia, Y.K., McMinn, P., Tonella, P., Vos, T., 2011. Symbolic search-based testing. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11), pp. 53–62.

Bertolino, A., 2007. Software testing research: achievements, challenges, dreams. In: Proceedings of the 1st Workshop on Future of Software Engineering (FOSE'07) at ICSE 2007, pp. 85–103.

Binder, B., 2012. Open Source Tools for Model-based Testing. http://www.robertvbinder.com/robertvbinder.com/open-source-tools-for-model-based-testing

Bjørner, N., Tillmann, N., Voronkov, A., 2009. Path feasibility analysis for string-manipulating programs. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 307–321.

Boonstoppel, P., Cadar, C., Engler, D.R., 2008. RWset: attacking path explosion in constraint-based test generation. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 351–366.

Borges, M., d'Amorim, M., Anand, S., Bushnell, D., Pasareanu, C., 2012. Symbolic execution with interval constraint solving and meta-heuristic search. In: Proceedings of the International Conference on Software Testing, Verification and Validation, pp. 111–120.

Bougé, L., Choquet, N., Fribourg, L., Gaudel, M.C., 1986. Test sets generation from algebraic specifications using logic programming. Journal of Systems and Software 6, 343–360.

Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., Utting, M., 2007. A subset of precise UML for model-based testing. In: A-MOST. ACM, pp. 95–104.

Briand, L.C., Feng, J., Labiche, Y., 2002. Using genetic algorithms and coupling measures to devise optimal integration test orders. In: Proceedings of International Conference on Software Engineering and Knowledge Engineering (SEKE'02), pp. 43–50.

Brinksma, E., Grieskamp, W., Tretmans, J., 2005. 04371 summary – perspectives of model-based testing. In: Brinksma, E., Grieskamp, W., Tretmans, J., (Eds.), Perspectives of Model-based Testing, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany. URL http://drops.dagstuhl.de/opus/volltexte/2005/364

Brinksma, E., Tretmans, J., 2000. Testing transition systems: an annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Ryan, M.D. (Eds.), MOVEP. Springer, pp. 187–195.

Brownlie, R., Prowse, J., Phadke, M.S., 1992. Robust testing of AT&T PMX/StarMAIL using OATS. AT & T Technical Journal 71 (3), 41–47.

Brucker, A., Wolff, B., 2012. On theorem prover-based testing. Formal Aspects of Computing, 1-3910.1007/s00165-012-0222-y.

Brumley, D., Poosankam, P., Song, D.X., 0002, J.Z., 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 143–157.

Bryce, R., Colbourn, C., 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. Journal of Information and Software Technology 48 (10), 960–970.

Bryce, R.C., Colbourn, C.J., 2007. One-test-at-a-time heuristic search for interaction test suites. In: Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO'07), Search Based Software Engineering Track, pp. 258–269.

Burroughs, K., Jain, A., Erickson, R.L., 1994. Improved quality of protocol testing through techniques of experimental design. In: Supercomm/IC: Proceedings of IEEE International Conference on Communications, pp. 745–752.

Cadar, C., Dunbar, D., Engler, D.R., 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the Symposium on Operating Systems Design and Implementation, pp. 209–224.

Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W., 2011. Symbolic execution for software testing in practice: preliminary assessment. In: Proceedings of the International Conference on Software Engineering (ICSE'11), pp. 1066–1071.

Cai, K.-Y., 2002. Optimal software testing and adaptive software testing in the context of software cybernetics. Information and Software Technology 44 (14), 841–855.

Calvagna, A., Gargantini, A., 2009. Combining satisfiability solving and heuristics to constrained combinatorial interaction testing. In: Proceedings of the 3rd International Conference on Tests and Proofs (TAP'09), pp. 27–42.

Calvagna, A., Gargantini, A., 2010. A formal logic approach to constrained combinatorial testing. Journal of Automated Reasoning 45, 331–358.

Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M., 2005. Testing concurrent object-oriented systems with spec explorer. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (Eds.), FM. Springer, pp. 542–547.

Castro, M., Costa, M., Martin, J.P., 2008. Better bug reporting with better privacy. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08), pp. 319–328.

Chan, K.P., Chen, T.Y., Towey, D., 2006. Forgetting test cases. In: Proceedings of the 30th Annual International Computer Software and Application Conference (COMPSAC'06), pp. 485–492.

Chan, K.P., Chen, T.Y., Towey, D., 2006. Restricted random testing: adaptive random testing by exclusion. International Journal of Software Engineering and Knowledge Engineering 16 (4), 553–584.

Chen, T.Y., Eddy, G.R., Merkel, G., Wong, P.K., 2004. Adaptive random testing through dynamic partitioning. In: Proceedings of the 4th International Conference on Quality Software (QSIC'04), pp. 79–86.

Chen, T.Y., Kuo, F.-C., Liu, H., 2009. Adaptive random testing based on distribution metrics. Journal of Systems and Software 82 (9), 1419–1433.

Chen, T.Y., Kuo, F.-C., Merkel, R.G., 2006. On the statistical properties of testing effectiveness measures. Journal of Systems and Software 79 (5), 591–601.

Chen, T.Y., Kuo, F.-C., Merkel, R.G., Ng, S.P., 2004. Mirror adaptive random testing. Information and Software Technology 46 (15), 1001–1010.

Chen, T.Y., Kuo, F.-C., Merkel, R.G., Tse, T.H., 2010. Adaptive random testing: the ART of test case diversity. Journal of Systems and Software 83 (1), 60–66.

Chen, T.Y., Kuo, F.-C., Zhou, Z.Q., 2007. On favourable conditions for adaptive random testing. International Journal of Software Engineering and Knowledge Engineering 17 (6), 805–825.

Chen, T.Y., Leung, H., Mak, I.K., 2004. Adaptive random testing. Proceedings of the 9th Asian Computing Science Conference. LNCS 3321, 320–329.

Chen, T.Y., Merkel, R., 2008. An upper bound on software testing effectiveness. ACM Transactions on Software Engineering and Methodology 17 (3), 16:1–16:27.

Chen, T.Y., Tse, T.H., Yu, Y.T., 2001. Proportional sampling strategy: a compendium and some insights. Journal of Systems and Software 58 (1), 65–81.

Chicano, F., Alba, E., 2005. Management of software projects with GAs. In: Proceedings of the 6th Metaheuristics International Conference (MIC'05).

Chipounov, V., Kuznetsov, V., Candea, G., 2011. S2e: a platform for in-vivo multi-path analysis of software systems. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11), pp. 265–278.

Chow, T., 1978. Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering 4, 178–187.

Ciupa, I., Leitner, A., Oriol, M., Meyer, B., 2008. ARTOO: adaptive random testing for object-oriented software. In: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), pp. 71–80.

Claessen, K., Hughes, J., 2000. QuickCheck: a lightweight tool for random testing of haskell programs. In: Odersky, M., Wadler, P., (Eds.), Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00), ACM. pp. 268–279.

Clements, P., Northrop, L.M., 2001. Software Product Lines: Practices and Patterns. Addison Wesley, Boston, MA.

Cochran, W.G., Cox, G.M., 1957. Experimental Designs, 2nd ed. J. Wiley & Sons, Inc., New York.

Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C., 1996. The combinatorial design approach to automatic test generation. IEEE Software 13 (5), 83–88.

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

Cohen, M.B., Colbourn, C.J., Gibbons, P.B., Mugridge, W.B., 2003. Constructing test suites for interaction testing. In: Proceedings of the 25th International Conference on Software Engineering (ICSE'03), pp. 38–48.

Cohen, M.B., Colbourn, C.J., Collofello, J., Gibbons, P.B., Mugridge, W.B., 2003. Variable strength interaction testing of components. In: Proceedings of the 27th IEEE International Computer Software and Applications Conference (COMPSAC'03), pp. 413–418.

Cohen, M.B., Colbourn, C.J., Ling, A.C.H., 2003. Augmenting simulated annealing to build interaction test suites. In: Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'03), pp. 394–405.

Cohen, M.B., Dwyer, M.B., Shi, J., 2006. Coverage and adequacy in software product line testing. In: Proceedings of the ISSTA 2006 Workshop on the Role of Architecture for Testing and Analysis (ROSATEA'06), pp. 53–63.

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

Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J.,2003. Constructing test suites for interaction testing. In: Proceedings of the 25th International Conference on Software Engineering (ICSE'03). IEEE Computer Society, pp. 38–48.

Colanzi, T.E., Assunçáo, W.K.G., Vergilio, S.R., Pozo, A.T.R.,2011. Integration test of classes and aspects with a multi-evolutionary and coupling-based approach. In: Proceedings of the 3rd International Symposium on Search Based Software Engineering (SSBSE'11). Springer.

Colbourn, C.J., 2004. Combinatorial aspects of covering arrays. Le Matematiche (Catania) 58, 121–167.

Colbourn, C.J., 2012. Covering Array Tables. Available at http://www.public.asu.edu/ccolbou/src/tabby/catable.html

Colbourn, C.J., Cohen, M.B., Turban, R.C., 2004. A deterministic density algorithm for pairwise interaction coverage. In: Proceedings of the IASTED International Conference on Software Engineering, pp. 345–352.

Colbourn, C., McClary, D., 2008. Locating and detecting arrays for interaction faults. Journal of Combinatorial Optimization 15 (1), 17–48.

Czerwonka, J., 2006. Pairwise testing in real world. In: Proceedings of the Pacific Northwest Software Quality Conference, pp. 419–430.

Dalal, S.R., Jain, A., Patton, G., Rathi, M., Seymour, P., 1998. AETGTDEQN^{SM}web: a Web based service for automatic efficient test generation from functional requirements. In: Proceedings of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT'98), pp. 84–85.

Dan, H., Hierons, R.M.,2011. Conformance testing from message sequence charts. In: Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST'11). IEEE Computer Society, pp. 279–288.

Derderian, K., Hierons, R., Harman, M., Guo, Q., 2006. Automated Unique Input–Output sequence generation for conformance testing of FSMs. The computer Journal 49 (3), 331–344.

Dick, J., Faivre, A., 1993. Automating the generation and sequencing of test cases from model-based specifications. In: Woodcock, J., Larsen, P.G. (Eds.), FME'93: Industrial-Strength Formal Methods, LNCS 670. Springer, pp. 268–284.

Dumlu, E., Yilmaz, C., Cohen, M.B., Porter, A., 2011. Feedback driven adaptive combinatorial testing. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11), pp. 243–253.

Dunietz, I.S., Ehrlich, W.K., Szablak, B.D., Mallows, C.L., Iannino, A., 1997. Applying design of experiments to software testing. In: Proceedings of the 19th International Conference on Software Engineering, (ICSE'97), pp. 205–215.

Dutertre, B., de Moura, L., 2006. A fast linear-arithmetic solver for DPLL(T). In: Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06), pp. 81–94.

Ehrig, H., Mahr, B., 1985. Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. volume 6 of Monographs in Theoretical Computer Science. An EATCS Series. Springer.

Ernits, J.P., Kull, A., Raiend, K., Vain, J., 2006. Generating tests from efsm models using guided model checking and iterated search refinement. In: (Havelund et al., 2006). pp. 85–99.

ETSI, 2011. Requirements for Modelling Notations. Technical Report ES 202 951. ETSI.

Fisher, R.A., 1971. The Design of Experiments, 8th ed. Hafner Publishing Company, New York.

Fouché, S., Cohen, M.B., Porter, A., 2009. Incremental covering array failure characterization in large configuration spaces. In: Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09), pp. 177–187.

Feijs, L.M.G., Goga, N., Mauw, S., Tretmans, J., 2002. Test selection, trace distance and heuristics. In: Schieferdecker, I., König, H., Wolisz, A., (Eds.), Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems (TestCom 2002), pp. 267–282. Kluwer.

Frantzen, L., Tretmans, J., Willemse, T.A.C., 2004. Test generation based on symbolic specifications. In: Grabowski and Nielsen (2005). pp. 1–15.

Fraser, G., Zeller, A.,2010. Mutation-driven generation of unit tests and oracles. In: Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10). ACM, pp. 147–158.

Fraser, G., Zeller, A.,2011. Exploiting common object usage in test case generation. In: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'11). IEEE, pp. 80–89.

Free Software Foundation, 2012. GNU GCC Compiler Collection. Available at: http://gcc.gnu.org/

Friedman, G., Hartman, A., Nagin, K., Shiran, T., 2002. Projected state machine coverage for software testing. In: Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA'02), pp. 134–143.

Ganesh, V., Dill, D.L., 2007. A decision procedure for bit-vectors and arrays. In: Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), pp. 519–531.

Ganov, S.R., Killmar, C., Khurshid, S., Perry, D.E., 2008. Test generation for graphical user interfaces based on symbolic execution. In: Proceedings of the 3rd IEEE/ACM International Workshop on Automation of Software Test (AST'08), pp. 33–40.

Garvin, B.J., Cohen, M.B., Dwyer, M.B., 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empirical Software Engineering (EMSE) 16 (February (1)), 61–102.

Gaudel, M.C., 1995. Testing can be formal, too. In: Mosses, P.D., Nielsen, M., Schwartzbach, M.I. (Eds.), TAPSOFT'95: Proceedings of the 6th International Joint Conference on Theory and Practice of Software Development, LNCS 915. Springer, pp. 82–96.

Glover, F., Kochenberger, G. (Eds.), 2003. Handbook of Metaheuristics. Springer.

Godefroid, P., 2007. Compositional dynamic test generation. In: Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07), pp. 47–54.

Godefroid, P., Kiezun, A., Levin, M.Y., 2008. Grammar-based whitebox fuzzing. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08), pp. 206–215.

Godefroid, P., Klarlund, N. and Sen, K., 2005. DART: directed automated random testing. In: Sarkar, V., Hall, M.W. (Eds.), Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05), pp. 213–223. ACM.

Godefroid, P., Levin, M.Y., Molnar, D.A., 2008. Automated whitebox fuzz testing. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08).

Godefroid, P., Luchaup, D., 2011. Automatic partial loop summarization in dynamic test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11), pp. 23–33.

Godefroid, P., Taly, A., 2012. Automated synthesis of symbolic instruction encodings from I/O samples. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12), pp. 441–452.

Grechanik, M., Csallner, C., Fu, C., Xie, Q., 2010. Is data privacy always good for software testing? In: Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE'10), pp. 368–377.

Grieskamp, W., 2006. Multi-paradigmatic model-based testing. In: (Havelund et al. 2006), pp. 1–19.

Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M., 2002. Generating finite state machines from abstract state machines. In: Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA'02), pp. 112–122.

Grieskamp, W., Hierons, R.M., Pretschner, A., 2011. 10421 Summary – model-based testing in practice. In: Grieskamp, W., Hierons, R.M., Pretschner, A. (Eds.), Model-based Testing in Practice, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. URL http://drops.dagstuhl.de/opus/volltexte/2011/2925

Grieskamp, W., Kicillof, N., 2006. A schema language for coordinating construction and composition of partial behavior descriptions. In: Whittle, J., Geiger, L., Meisinger, M. (Eds.), Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'06). ACM, pp. 59–66.

Grieskamp, W., Kicillof, N., Stobie, K., Braberman, V.A., 2011b. Model-based quality assurance of protocol documentation: tools and methodology. Software Testing. Verification and Reliability (STVR) 21, 55–71.

Grieskamp, W., Kicillof, N., Tillmann, N., 2006a. Action machines: a framework for encoding and composing partial behaviors. International Journal of Software Engineering and Knowledge Engineering 16, 705–726.

Grieskamp, W., Nachmanson, L., Tillmann, N., Veanes, M., 2003. Test case generation from AsmL specifications. In: Börger, E., Gargantini, A., Riccobene, E. (Eds.), Abstract State Machines. Springer, p. 413.

Grieskamp, W., Qu, X., Wei, X., Kicillof, N., Cohen, M.B., 2009. Interaction coverage meets path coverage by smt constraint solving. In: Núñez, M., Baker, P., Merayo, M.G., (Eds.), Proceedings of the Joint Conference of the 21st IFIP International Conference on Testing of Communicating Systems and 9th International Workshop on Formal Approaches to Testing of Software (TESTCOM/FATES'09), LNCS 5826, pp. 97–112. Springer.

Grieskamp, W., Tillmann, N., Schulte, W., 2006b. XRT – exploring runtime for.net – architecture and applications. Electronic Notes on Theoretical Computer Science 144, 3–26.

Grieskamp, W., Tillmann, N., Veanes, M., 2004. Instrumenting scenarios in a model-driven development environment. Information & Software Technology 46, 1027–1036.

Grosso, C.D., Antoniol, G., Penta, M.D., Galinier, P., Merlo, E.,2005. Improving network applications security: a new heuristic to generate stress testing data. In: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, vol. 1. ACM, pp. 1037–1043.

Groz, R., Charles, O., Renévot, J., 1996. Relating conformance test coverage to formal specifications. In: Gotzhein, R., Bredereke, J., (Eds.), Proceedings of the IFIP TC6/6.1 International Conference on Formal Description Techniques IX/Protocol Specification, Testing and Verification XVI (FORTE'96), pp. 195–210. Chapman & Hall.

Harman, M., 2008. Open problems in testability transformation. In: Proceedings of the 1st International Workshop on Search Based Testing (SBT'08), Keynote paper.

Harman, M., 2011. Software engineering meets evolutionary computation. IEEE Computer 44 (10), 31–39.

Harman, M., Islam, F., Xie, T., Wappler, S., 2009. Automated test data generation for aspect-oriented programs. In: Proceedings of the 8th International Conference on Aspect-oriented Software Development (AOSD'09), pp. 185–196.

Harman, M., Jia, Y., Langdon, B.,2011. Strong higher order mutation-based test data generation. In: Proceedings of the 8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11). ACM, pp. 212–222.

Harman, M., Jones, B.F., 2001. Search based software engineering? Information and Software Technology 43 (14), 833–839.

Harman, M., Kim, S.G., Lakhotia, K., McMinn, P., Yoo, S., 2010. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In: Proceedings of the 3rd International Workshop on Search-Based Software Testing (SBST'10), pp. 182–191.

Harman, M., Lakhotia, K., McMinn, P.,2007. A multi-objective approach to search-based test data generation. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation(GECCO'07). ACM Press, pp. 1098–1105.

Harman, M., Mansouri, A., Zhang, Y., 2009. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, Department of Computer Science, King's College London, April.

Harman, M., McMinn, P., 2010. A theoretical and empirical study of search based testing: local, global and hybrid search. IEEE Transactions on Software Engineering 36 (2), 226–247.

Hartman, A., 2002. Model-based Test Generation Tools. Technical Report. AGEDIS Consortium. http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf

Hartman, A., Raskin, L., 2004. Problems and algorithms for covering arrays. Discrete Mathematics 284, 149–156.

Hartmann, J., Vieira, M., Foster, H., Ruder, A., 2005. A UML-based approach to system testing. Innovations in Systems and Software Engineering (ISSE) 1 (1), 12–24.

Helke, S., Neustupny, T., Santen, T., 1997. Automating test case generation from Z specifications with Isabelle. In: Bowen, J.P., Hinchey, M.G., Till, D., (Eds.), Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation (ZUM'97), pp. 52–71. Springer.

Hemmati, H., Arcuri, A., Briand, L., 2010. Reducing the cost of model-based testing through test case diversity. In: Proceedings of the 22nd IFIP International Conference on Testing Software and System (ICTSS'10), pp. 63–78.

Hemmati, H., Arcuri, A., Briand, L., 2011. Empirical investigation of the effects of test suite properties on similarity-based test case selection. In: Proceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST'11), pp. 327–336.

Hierons, R.M., 2010. Reaching and distinguishing states of distributed systems. SIAM Journal on Computing 39, 3480–3500.

Hierons, R.M., Merayo, M.G., Núñez, M., 2008. Implementation relations for the distributed test architecture. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T., (Eds.), Testing of Software and Communicating Systems: Proceedings of the Joint Conference of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems and the 8th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES'08). LNCS 5047, pp. 200–215. Springer.

Hierons, R.M., Ural, H., 2008. The effect of the distributed test architecture on the power of testing. Computer Journal 51, 497–510.

Hnich, B., Prestwich, S., Selensky, E., Smith, B., 2006. Constraint models for the covering test problem. Constraints 11, 199–219.

Huima, A., 2007. Implementing conformiq qtronic. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W., (Eds.), Testing of Software and Communicating Systems: Proceedings of the Joint Conference of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES'07), LNCS 4581, pp. 1–12. Springer.

Huo, J., Petrenko, A., 2005. Covering transitions of concurrent systems through queues. In: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, (ISSRE'05), pp. 335–345.

Inkumsah, K., Xie, T., 2007. Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In: Stirewalt, R.E.K., Egyed, A., Fischer, B., editors, Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), pp. 425–428. ACM.

Iqbal, M.Z., Arcuri, A., Briand, L. 2011. Automated system testing of real-time embedded systems based on environment models. Technical Report 2011–2019, Simula Research Laboratory.

Jard, C., Jéron, T., 2005. TGV: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer (STTT) 7, 297–315.

Jayaraman, K., Harvison, D., Ganeshan, V., Kiezun, A., 2009. A concolic whitebox fuzzer for Java. In: Proceedings of the 1st NASA Formal Methods Symposium, pp. 121–125.

Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E., 2005. Symbolic test selection based on approximate analysis. In: Halbwachs, N., Zuck, L.D., (Eds.), Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), pp. 349–364. Springer.

Kaplan, M., Klinger, T., Paradkar, A.M., Sinha, A., Williams, C., Yilmaz, C.,2008. Less is more: A minimalistic approach to UML model-based conformance test generation. In: Proceedings of The 1st IEEE International Conference on Software Testing Verification and Validation (ICST'08). IEEE Computer Society, pp. 82–91.

Katara, M., Kervinen, A., 2006. Making model-based testing more agile: a use case driven approach. In: Bin, E., Ziv, A., Ur, S. (Eds.), Haifa Verification Conference, pp. 219–234. Springer.

Kähkönen, K., Launiainen, T., Saarikivi, O., Kauttio, J., Heljanko, K., Niemelä, I., 2011. LCT: an open source concolic testing tool for Java programs. In: Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bycode'11), pp. 75–80.

Khurshid, S., Pasareanu, C., Visser, W., 2003. Generalized symbolic execution for model checking and testing. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), pp. 553–568.

Khurshid, S., Suen, Y.L., 2005. Generalizing symbolic execution to library classes. In: Proceedings of the 6th Workshop on Program Analysis for Software Tools and Engineering (PASTE'05), pp. 103–110.

King, J.C., 1975. A new approach to program testing. In: Programming Methodology. LNCS, Vol. 23, pp. 278-290.

Kuhn, D., Higdon, J., Lawrence, J., Kacker, R., Lei, Y., 2012. Combinatorial methods for event sequence testing. In: Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST'12), pp. 601–609.

Kuhn, D., Okun, V., 2006. Pseudo-exhaustive testing for software. In: Proceedings of the 30th NASA/IEEE Software Engineering Workshop (SEW'06), pp. 153–158.

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

Kuliamin, V.V., Petrenko, A.K., Kossatchev, A., Burdonov, I.B., 2003. The UniTesK approach to designing test suites. Programming and Computer Software 29, 310–322.

Kuo, F.-C., 2006. On Adaptive Random Testing. Ph.D. Thesis. Faculty of Information and Communication Technologies, Swinburne University of Technology.

Lakhotia, K., Harman, M., McMinn, P.,2008. Handling dynamic data structures in search based testing. In: Proceedings of the 10th annual conference on Genetic and evolutionary computation (GECCO'08). ACM Press, pp. 1759–1766.

Lakhotia, K., McMinn, P., Harman, M., 2010. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. Journal of Systems and Software 83 (12), 2379–2391.

Lakhotia, K., Tillmann, N., Harman, M., de Halleux, J., 2010. Flopsy – Search-based floating point constraint solving for symbolic execution. In: Proceedings of the 23rd IFIP International Conference on Testing Software and Systems (ICTSS'10), pp. 142–157.

Larsen, K.G., Mikucionis, M., Nielsen, B., 2004. Online testing of real-time systems using UPPAAL. In: Grabowski and Nielsen (2005). pp. 79–94.

Lee, D., Yannakakis, M.,1996. Principles and methods of testing finite state machines – a survey. In: Proceedings of the IEEE. IEEE Computer Society Press, pp. 1090–1123, 84(8).

Legeard, B., Peureux, F., Utting, M., 2002. Automated boundary testing from Z and B. In: Eriksson, L.H., Lindsay, P.A. (Eds.), FME 2002: Formal Methods-Getting IT Right, LNCS 2391. Springer, pp. 21–40.

Legeard, B., Utting, M., 2010. Model-based testing –next generation functional software testing. Journal of Software Technology 12. http://journal.thedacs.com/issue/52/145

Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J., 2008. Ipog-ipog-d: efficient test generation for multi-way combinatorial testing. Software Testing Verification and Reliability 18, 125–148.

Lin, Y., Tang, X., Chen, Y., Zhao, J., 2009. A divergence-oriented approach to adaptive random testing of Java programs. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE'09), pp. 221–232.

Liu, M.-H., Gao, Y.-F., Shan, J.-H., Liu, J.-H., Zhang, L., Sun, J.-S., 2006. An approach to test data generation for killing multiple mutants. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06), pp. 113–122.

Liu, H., Xie, X., Yang, J., Lu, Y., Chen, T.Y., 2011. Adaptive random testing through test profiles. Software: Practice and Experience 41 (10), 1131–1154.

Ma, K.K., Khoo, Y.P., Foster, J.S., Hicks, M., 2011. Directed symbolic execution. In: Proceedings of the 18th International Conference on Static Analysis (SAS'11), pp. 95–111.

Malaiya, Y.K., 1995. Antirandom testing: Getting the most out of black-box testing. In: Proceedings of the 6th International Symposium on Software Reliability Engineering (ISSRE'95, pp. 86–95.

Mandl, R., 1985. Orthogonal Latin squares: an application of experiment design to compiler testing. Communications of the ACM 28 (10), 1054–1058.

Majumdar, R., Saha, I., 2009. Symbolic robustness analysis. In: Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS'09), pp. 355–363.

Majumdar, R., Sen, K., 2007. Hybrid concolic testing. In: Proceedings of the 29th International Conference on Software Engineering (ICSE'07), pp. 416–426.

Majumdar, R., Xu, R.G., 2007. Directed test generation using symbolic grammars. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), pp. 134–143.

Majumdar, R., Xu, R.G., 2009. Reducing test inputs using information partitions. In: Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09), pp. 555–569.

Mayer, J., 2005. Lattice-based adaptive random testing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), pp. 333–336.

McGregor, J.D., 2001. Testing a software product line. Tech. Rep. No. cmu/sei-2001-tr-022, Carnegie Mellon Software Engineering Institute.

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

McMinn, P.,2011. Search-based software testing: past, present and future. In: Proceedings of the 4th International Workshop on Search-Based Software Testing (SBST 2011). IEEE, pp. 153–163.

McMinn, P., Binkley, D., Harman, M., 2009. Empirical evaluation of a nesting testability transformation for evolutionary testing. ACM Transactions on Software Engineering Methodology 18 (3), 11:1–11:27.

McMinn, P., Harman, M., Hassoun, Y., Lakhotia, K., Wegener, J., 2012. Input domain reduction through irrelevant variable removal and its effect on local, global and hybrid search-based structural test data generation. IEEE Transactions on Software Engineering 38 (2), 453–477.

McMinn, P., Shahbaz, M., Stevenson, M., 2012. Search-based test input generation for string data types using the results of web queries. In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST'12).

McMinn, P., Stevenson, M., Harman, M., 2010. Reducing qualitative human oracle costs associated with automatically generated test data. In: Proceedings of the 1st International Workshop on Software Test Output Validation (STOV'10), pp. 1–4.

Merkel, R. G., 2005. Analysis and Enhancements of Adaptive Random Testing. Ph.D. Thesis. School of Information Technology, Swinburne University of Technology.

Michael, C., McGraw, G., Schatz, M., 2001. Generating software test data by evolution. IEEE Transactions on Software Engineering 27 (12), 1085–1110.

de Moura, L.M., Bjørner, N., 2008. Z3: an efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), pp. 337–340.

Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W., 2004. Optimal strategies for testing nondeterministic systems. In: Avrunin, G.S., Rothermel, G. (Eds.), Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04), pp. 55–64.

Naito, S., Tsunoyama, M., 1981. Fault detection for sequential machines by transition-tours. In: Proceedings of the 11th annual International Symposium on Fault-Tolerant Computing (FTCS'81), pp. 238–243.

Ngo, M.N., Tan, H.B.K., 2007. Detecting large number of infeasible paths through recognizing their patterns. In: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 215–224.

Nguyen, C., Perini, A., Tonella, P., Miles, S., Harman, M., Luck, M., 2009. Evolutionary testing of autonomous software agents. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09), pp. 521–528.

Nie, C., Leung, H., 2011. A survey of combinatorial testing. ACM Computing Surveys 43 (2), 1–29.

Nielsen, B., Skou, A., 2003. Automated test generation from timed automata. International Journal on Software Tools for Technology Transfer (STTT) 5, 59–77.

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

Offutt, A.J., Abdurazik, A., 1999. Generating tests from UML specifications. In: France, R.B., Rumpe, B., (Eds.), Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard (UML'99), pp. 416–429. Springer.

Ostrand, T.J., Balcer, M.J., 1988. The category-partition method for specifying and generating functional tests. Communications of the ACM 31, 678–686.

Papadakis, M., Malevris, N., 2010. Automatic mutation test case generation via dynamic symbolic execution. In: Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10).

Pasareanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M.R., Person, S., Pape, M., 2008. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: International Symposium on Software Testing and Analysis (ISSTA'08), pp. 15–26.

Pasareanu, C.S., Rungta, N., 2010. Symbolic PathFinder: Symbolic execution of Java bytecode. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), pp. 179–180.

Pasareanu, C.S., Rungta, N., Visser, W., 2011. Symbolic execution with mixed concrete-symbolic solving. In: Proceedings of 2011 International Symposium on Software Testing and Analysis (ISSTA'11), pp. 34–44.

Pasareanu, C.S., Visser, W., 2009. A survey of new trends in symbolic execution for software testing and analysis. International Journal on Software Tools for Technology Transfer 11, 339–353.

Paulson, L.C., 1994. Isabelle – a generic theorem prover (with a contribution by T. Nipkow). LNCS 828, Springer.

Perrouin, G., Sen, S., Klein, J., Baudry, B., Traon, Y.L., 2010. Automated and scalable t-wise test case generation strategies for software product lines. In: Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST'10), pp. 459–468.

Pezzè, M., Young, M., 2007. Software Testing and Analysis – Process, Principles and Techniques. Wiley.

Plat, N., Larsen, P.G., 1992. An overview of the iso/vdm-sl standard. SIGPLAN Notice 27, 76–82.

Qi, D., Roychoudhury, A., Liang, Z., Vaswani, K., 2009. Darwin: an approach for debugging evolving programs. In: Proceedings of the 2009 Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 33–42.

Qu, X., Cohen, M.B., Rothermel, G., 2008. Configuration-aware regression testing: an empirical study of sampling and prioritization. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08), pp. 75–85.

Qu, X., Cohen, M.B., Woolf, K.M., 2007. Combinatorial interaction regression testing: a study of test case generation and prioritization. In: Proceedings of the 23rd International Conference on Software Maintenance (ICSM'07), pp. 255–264.

Santelices, R.A., Chittimalli, P.K., Apiwattanapong, T., Orso, A., Harrold, M.J., 2008. Test-suite augmentation for evolving software. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08), pp. 218–227.

Santelices, R.A., Harrold, M.J., 2010. Exploiting program dependencies for scalable multiple-path symbolic execution. In: Proceedings of the 2010 International Symposium on Software Testing and Analysis (ISSTA'10), pp. 195–206.

Saxena, P., Poosankam, P., McCamant, S., Song, D., 2009. Loop-extended symbolic execution on binary programs. In: Proceedings of the 2010 International Symposium on Software Testing and Analysis (ISSTA'10), pp. 225–236.

Segall, I., Tzoref-Brill, R., Farchi, E., 2011. Using binary decision diagrams for combinatorial test design. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11), pp. 254–264.

Sen, K., Agha, G., 2006. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06), pp. 419–423.

Sen, K., Marinov, D., Agha, G., 2005. CUTE: A concolic unit testing engine for C. In: Proceedings of the 2005 Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 263–272.

Shahbazi, A., Tappenden, A., Miller, J., 2013. Centroidal voronoi tessellations – a new approach to random testing. IEEE Transactions on Software Engineering 39 (2), 163–183.

Sherwood, G.B., 1994. Effective testing of factor combinations. In: Proceedings of the 3rd International Conference on Software Testing, Analysis & Review (STAR'94), pp. 151–166.

Souza, M., Borges, M., d'Amorim, M., Pasareanu, C.S., 2011. CORAL: solving complex constraints for symbolic pathfinder. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (Eds.), NASA Formal Methods – Third International Symposium (NFM'11), LNCS 6617, pp. 359–374. Springer.

Stardom, J., 2001. Metaheuristics and the search for covering and packing arrays, Master's Thesis. Simon Fraser University.

Sterling, L., Shapiro, E.Y., 1994. The Art of Prolog – Advanced Programming Techniques, 2nd ed. MIT Press.

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

Tappenden, A., Miller, J., 2009. A novel evolutionary approach for adaptive random testing. IEEE Transactions on Reliability 58 (4), 619–633.

Tappenden, A., Miller, J., 2013. Automated cookie collection testing. ACM Transactions on Software Engineering and Methodology, in press.

Tillmann, N., de Halleux, J., 2008. Pex-White box test generation for. NET. In: Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08), pp. 134–153.

Tomb, A., Brat, G.P., Visser, W., 2007. Variably interprocedural program analysis for runtime error detection. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07), pp. 97–107.

Tonella, P.,2004. Evolutionary testing of classes. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04). ACM, pp. 119–128.

Tretmans, J., 1996. Test Generation with Inputs, Outputs, and Repetitive Quiescence. Software–Concepts and Tools 17, 103–120.

Tretmans, J., 2008. Model Based Testing with Labelled Transition Systems. In: Hierons, R., Bowen, J., Harman, M. (Eds.), Formal Methods and Testing. Springer-Verlag, pp. 1–38.

Tretmans, J., Brinksma, E., 2003. TorX: Automated model based testing. In: Proceedings of the 1st European Conference on Model-Driven Software: Foundations and Applications (ECMDA-FA'05), pp. 31–43.

Tung, Y., Aldiwan, W.S., 2000. Automating test case generation for the new generation mission software system. In: Proceedings of the IEEE Aerospace Conference, pp. 431–437.

Utting, M., Legeard, B., 2007. Practical Model-based Testing – A Tools Approach. Morgan Kaufmann.

Vasilevskii, M., 1973. Failure diagnosis of automata. Kibernetika, 98–108.

Veanes, M., Bjørner, N., 2010. Alternating simulation and IOCO. In: Petrenko, A., da Silva Simão, A., Maldonado, J.C. (Eds.), Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS'10), pp. 47–62. Springer.

Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L., 2008. Model-based testing of object-oriented reactive systems with Spec Explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (Eds.), Formal Methods and Testing. Springer, pp. 39–76.

Veanes, M., Campbell, C., Schulte, W., 2007. Composition of model programs. In: Derrick, J., Vain, J., (Eds.), Proceedings of the 27th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems (FORTE'07), pp. 128–142. Springer.

Veanes, M., de Halleux, P., Tillmann, N., 2010. Rex: Symbolic regular expression explorer. In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10), pp. 498–507.

Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S.,2006. Time aware test suite prioritization. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'06). ACM Press, pp. 1–12.

Wegener, J., Bühler, O.,2004. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In: Genetic and Evolutionary Computation Conference (GECCO 2004), LNCS 3103. Springer, pp. 1400–1412.

Wegener, J., Grochtmann, M., 1998. Verifying timing constraints of real-time systems by means of evolutionary testing. Real-Time Systems 15 (3), 275–298.

Weyuker, E.J., 1979. Translatability and decidability questions for restricted classes of program schemas. SIAM Journal on Computing 8 (4), 587–598.

White, L.J., Cohen, E.I., 1980. A domain strategy for computer program testing. IEEE Transactions on Software Engineering 6 (3), 247–257.

Wieczorek, S., Stefanescu, A., 2010. Improving testing of enterprise systems by model-based testing on graphical user interfaces. In: Sterritt, R., Eames, B., Sprinkle, J. (Eds.), Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'10), pp. 352–357.

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

Yoo, S., Harman, M., Tonella, P., Susi, A., 2009. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: Proceedings of the 2009 ACM International Conference on Software Testing and Analysis (ISSTA 09), pp. 201–212.

Yuan, X., Cohen, M., Memon, A., 2011. GUI interaction testing: Incorporating event context. IEEE Transactions on Software Engineering 37 (4), 559–574.

Zhan, Y., Clark, J.A., 2005. Search-based mutation testing for simulink models. In: Beyer, H.-G., O'Reilly, U.-M. (Eds.), Proceedings of the 2005 Genetic and Evolutionary Computation Conference (GECCO'05), pp. 1061–1068. ACM.

Zhang, Y., Finkelstein, A., Harman, M.,2008. Search based requirements optimisation: existing work and challenges. In: Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'08), LNCS 5025. Springer, pp. 88–94.

Zhang, P., Elbaum, S.G., Dwyer, M.B., 2011. Automatic generation of load tests. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11), pp. 43–52.

Zhou, B., Okamura, H., Dohi, T., 2013. Enhancing performance of random testing through Markov chain Monte Carlo methods. IEEE Transactions on Computers 62 (1), 186–192.

Zhu, H., Hall, P.A.V., May, J.H.R., 1997. Software unit test coverage and adequacy. ACM Computing Surveys 29 (4), 366–427.

Saswat Anand is a Research Associate in the department of Computer Science at Stanford University. He is interested in developing static and dynamic analysis techniques and building systems that can be applied to real-world software. He received the PhD in computer science from Georgia Institute of Technology.

Professor Edmund Burke is Deputy Principal for Research at the University of Stirling. His research interests lie at the interface of Operational Research and Computer Science. He is also a Fellow of the Operational Research Society, the British Computer Society and the Institute of Mathematics and its Applications. Prof. Burke is Editor-in-chief of the Journal of Scheduling, Area Editor (for Combinatorial Optimisation) of the Journal of Heuristics, Associate Editor of the INFORMS Journal on Computing, Associate Editor of the IEEE Transactions on Evolutionary Computation, a member of the Advisory Board of the EURO Journal on Computational Optimization and he is a member of the Editorial Board of Memetic Computing. Since 1995, Prof. Burke has led the organisation of the international series of conferences on the Practice and Theory of Automated Timetabling (PATAT). In addition, he has played a leading role in the organisation of several other major international conferences in his research field. He has edited/authored 14 books and has published over 250 refereed papers. He has been awarded 56 externally funded grants worth over £16M from a variety of sources including EPSRC, ESRC, BBSRC, EU, Research Council of Norway, East Midlands Development Agency, HEFCE, Teaching Company Directorate, Joint Information Systems Committee of the HEFCs and commercial organisations.

Tsong Yueh Chen got his BSc and MPhil from The University of Hong Kong, MSc and DIC from the Imperial College, The University of London, and PhD from The University of Melbourne. He is currently a Professor of Software Engineering at Swinburne University of Technology, Australia. Prior to joining Swinburne, he has taught at The University of Hong Kong and The University of Melbourne. His main research interests include software testing and debugging.

John Clark is Professor of Critical Systems at the University of York. His primary research interests centre on the use of AI techniques for the development, analysis and testing of high integrity software and systems. Secure systems have formed an important long-running and wider topic of research. He works within the Non-standard Computation research group: his research interests extend to the use of AI for NMR pulse sequence generation and the automated synthesis of quantum circuits and algorithms (quantum software engineering). His work is currently sponsored by the EPSRC (EP/G043604/1 - The Birth Life and Death of Semantic Mutants; EP/J017515/1 - DAASE: Dynamic Adaptive Automated Software Engineering) and Defence Science Technology Ltd (DSTL).

Myra Cohen is an Associate Professor in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln where she is a member of the Laboratory for Empirically based Software Quality Research and Development (ESQuaReD). She received the PhD degree in computer science from the University of Auckland, New Zealand, the MS degree in computer science from the University of Vermont, and the BS degree from the School of Agriculture and Life Sciences, Cornell University. She is a recipient of a National Science Foundation Faculty Early CAREER Development Award and an Air Force Office of Scientific Research Young Investigator Program Award. Her research interests include testing of configurable software systems and software product lines, combinatorial interaction testing, and search based software engineering.

Dr. Wolfgang Grieskamp is a Staff Engineer at Google, were he works on tools and frameworks for building reliable services in the cloud. Before joining Google, Wolfgang was a Principal Architect at Microsoft and previously a Senior Researcher at Microsoft Research. Wolfgang obtained his PhD from the Technical University of Berlin, has written a number of software systems, and published numerous papers in the areas of model-based testing, formal methods, verification, and compiler construction.

Mark Harman is professor of Software Engineering in the Department of Computer Science at University College London where he directs the CREST centre. He is widely known for work on source code analysis and testing and was instrumental in the founding of the field of Search Based Software Engineering (SBSE). Since its inception in 2001, SBSE has rapidly grown to include over 800 authors, from 270 institutions spread over 40 countries.

Mary Jean Harrold is a Professor in the School of Computer Science at Georgia Institute of Technology. Her research includes analysis and testing of large, evolving software, fault localization and failure identification using statistical analysis and visualization, and monitoring deployed software to improve quality. Harrold received an NSF National Young Investigator Award, and was named an ACM Fellow and an IEEE Fellow. She serves on the Board of Directors for the Computing Research Association (CRA) and the editorial board of JSTVR, and served on the editorial boards of ACM TOSEM and TOPLAS and IEEE TSE. She served as NSF ADVANCE Professor of Computing, on the leadership of team of the National Center for Women in Information Technology (NCWIT), and on the CRA Committee on the Status of Women in Computing Research (CRA-W). She received the PhD in computer science from the University of Pittsburgh.

Phil McMinn is a Senior Lecturer in Computer Science at the University of Sheffield, UK. He has published several papers in the field of search based testing. His research interests cover software testing in general, program transformation and agent-based systems and modelling. He is currently funded by the UK Engineering and Physical Science Research Council (EPSRC) to work on reducing oracles costs of testing.