Software Testing: A Research Travelogue (2000–2014)

Alessandro Orso
College of Computing
School of Computer Science
Georgia Institute of Technology
Atlanta, GA, USA
orso@cc.gatech.edu

Gregg Rothermel
Department of Computer Science
and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
grother@cse.unl.edu

ABSTRACT

Despite decades of work by researchers and practitioners on numerous software quality assurance techniques, testing remains one of the most widely practiced and studied approaches for assessing and improving software quality. Our goal, in this paper, is to provide an accounting of some of the most successful research performed in software testing since the year 2000, and to present what appear to be some of the most significant challenges and opportunities in this area. To be more inclusive in this effort, and to go beyond our own personal opinions and biases, we began by contacting over 50 of our colleagues who are active in the testing research area, and asked them what they believed were (1) the most significant contributions to software testing since 2000 and (2) the greatest open challenges and opportunities for future research in this area. While our colleagues' input (consisting of about 30 responses) helped guide our choice of topics to cover and ultimately the writing of this paper, we by no means claim that our paper represents all the relevant and noteworthy research performed in the area of software testing in the time period considered—a task that would require far more space and time than we have available. Nevertheless, we hope that the approach we followed helps this paper better reflect not only our views, but also those of the software testing community in general.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Algorithms, Experimentation, Verification

Keywords: Software testing

PROLOGUE

In 2000, the International Conference on Software Engineering held its first "Future of Software Engineering" track, and featured a number of papers described as "Roadmaps". These papers offered assessments, by authors expert in various areas of software engineering, of the directions we could expect research in those areas to take. Among the papers and presentations was one entitled, "Testing: A Roadmap", authored by Mary Jean Harrold.

In the spring of 2013, we (Alex and Gregg) were contacted by the organizers of the FOSE track for ICSE 2014, Matthew Dwyer and James Herbsleb. They told us that, in addition to the usual "Roadmaps", they wanted to include in the FOSE track some papers called "Travelogues", in which researchers would reflect on the research performed, since 2000, in various software engineering areas, as well as expose potential future directions. They asked us to prepare a Travelogue on software testing. Of course, we knew that the appropriate person to prepare such a Travelogue would be Mary Jean herself. However, we also knew that Mary Jean had declined their invitation, as she was facing a different and much more profound challenge.

As Mary Jean's Ph.D. students, postdoctoral advisees, close colleagues, and friends, we are humbled in taking on the task of writing a Travelogue that should, indeed, be hers to write. As we reread Mary Jean's own words, from her Travelogue of 2000, we hear her voice clearly in our heads; and we overhear again the many conversations we had with her, about work, about life, about life's work. We know we have been forever changed by her, and will forever feel the effects of those changes.

But of course, we are not the only persons who can say this. Mary Jean touched many lives in many ways, and helped so many people achieve a level of potential that they might not otherwise have achieved. Alex remembers something that Mary Jean said often during his early years at Georgia Tech, as they were pulling all-nighters for papers or research proposals: "If anyone can do it, you can, Alex." The way Mary Jean said this, with her genuine smile, completely captures her attitude toward people—students and collaborators in particular—and her ability to motivate and even nudge them a bit, while always doing so in a pleasant and encouraging way. Gregg recalls a day when, nearing graduation and beginning his job search, and tired of spending the bulk of his time in the lab, he told Mary Jean that he hoped to end up someplace where he could plant an orchard and grow trees. Mary Jean replied, "your students will be your trees". And so they have been.

We apologize to readers who may feel that this preface is too personal, or unnecessary in a paper that should be about research. We nevertheless believe that we owed this to Mary Jean, a wonderful person whose legacy is not just in the impact of her research (as far-reaching as that has been), but also in the lives of those she touched, and the lives of those who they will touch.

1. INTRODUCTION

As we mentioned in our preface, this is not the first paper to attempt to assess the state of the art and possible future directions for software testing. Where the Future of Software Engineering (FOSE) track is concerned, two such papers have appeared: Mary Jean Harrold's 2000 paper, "Testing: A Roadmap" [88] (already mentioned), and Antonia Bertolino's 2007 paper, "Software Testing Research: Achievements, Challenges, Dreams" [19]. We encourage our readers to also consider these earlier efforts to obtain a more comprehensive picture of the field.

FOSE'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2865-4/14/05
http://dx.doi.org/10.1145/2593882.2593885

Figure 1: Word cloud generated using the responses to our informal survey.

Mary Jean Harrold's roadmap, in particular, began with the following sentence: "A report by the Workshop on Strategic Directions in Software Quality posits that software quality will become the dominant success criterion in the software industry [143]." Few would argue that this prediction was inaccurate. Testing remains one of the most widely practiced approaches for assessing (and ultimately improving) the quality of software, and it remains one of the most extensively researched software engineering topics. In this paper, as requested by the FOSE chairs, we provide an accounting of the research in software testing over the past 14 years, focusing on the areas in which the greatest progress has been made and the highest impact has been achieved. We also comment on significant challenges and opportunities for future researchers in these areas.

While we, the authors of this paper, have a certain amount of experience in various areas of testing research, and follow much of the testing research to the best of our capabilities, we wished to go beyond our personal views, opinions, and knowledge of the area in preparing this Travelogue. We therefore began by reaching out to many of our colleagues, in an attempt to obtain some larger consensus as to the work that the testing research community views as having been the most important and promising. Specifically, we identified over 50 colleagues who are currently active in testing research, and sent them email asking two questions:

1. What do you think are the most significant contributions to testing since 2000, whether from you or from other researchers?
2. What do you think are the biggest open challenges and opportunities for future research in this area?

Primarily due to lack of forethought on our part, we gave our typically very busy colleagues a fairly short deadline for sending responses. We were heartened, therefore, when about 30 of them were able to get back to us with comments and, in many cases, extensive input. To provide a quick overview of the most common topics mentioned in the responses we received, Figure 1 contains a word cloud that we generated using all such responses and filtering out obvious keywords, such as "test" and "testing".

We used our colleagues' responses to compile lists of contributions, challenges, and opportunities, and we prioritized these based on the frequency with which they appeared in such responses. We classified most of the identified contributions as "research contributions", while classifying two as "practical contributions". These latter are contributions that, in our view, were driven more by industrial effort than by research activities, and yet have had a dramatic effect on both practice and research. As might be expected, in most of these areas of contributions there remain challenges and opportunities for future research, many of which were mentioned by our colleagues, and that we note in the remainder of this paper. We also selected several areas that were seen not as areas in which substantial contribution had yet been made, but rather, as areas that pose new (or continuing) challenges and opportunities for researchers.

In presenting our thoughts on contributions and opportunities, we attempted to cite relevant papers in the areas discussed. We make no pretense, however, of having cited all such papers, a task better left to survey papers, several of which we cite to guide readers further. Similarly, while our colleagues' input helped guide our choice of topics to cover, we do not claim that our paper represents *all* the relevant and noteworthy research performed in the area of software testing in the time period considered. Such a task would require considerably more space and time than we have available. Nevertheless, we truly hope that the approach we followed helped this paper better reflect the views of researchers in the software testing community and provide an unbiased view on this challenging and exciting research field.

We structure the rest of this paper as follows. Section 2 describes research contributions and additional opportunities in these areas. Section 3 describes practical contributions and additional opportunities in these areas. Section 4 describes additional areas in which opportunities and challenges exist. Finally, Section 5 concludes.

2. RESEARCH CONTRIBUTIONS

We classified the nine research contributions that we identified from our colleagues' responses into four categories: (1) automated test input generation, (2) testing strategies, (3) regression testing, and (4) support for empirical studies. The following subsections present contributions in each of these categories.

2.1 Automated Test Input Generation

Among the various contributions that testing researchers have made since 2000, the contributions that were most frequently mentioned by our colleagues involved automated test input generation. Automated test input generation techniques attempt to generate a set of input values for a program or program component, typically with the aim of achieving some coverage goal or reaching a particular state (*e.g.,* the failing of an assertion).

Test input generation is by no means a new research direction, and there is a considerable amount of work on the topic prior to 2000, but the last decade has seen a resurgence of research in this area and has produced several strong results and contributions. This resurgence may stem, in part, from improvements in computing platforms and the processing power of modern systems. However, we believe (and our colleagues' responses to our inquiry support this) that researchers themselves deserve the greatest credit for the resurgence, through advances in related areas and supporting technologies, such as symbolic execution, search-based testing, random and fuzz testing, and combinations thereof. In the rest of this section, we discuss each of these areas and supporting technologies.

2.1.1 Symbolic Execution

Advances in symbolic execution are one of the main reasons automated test input generation has become more relevant. Static symbolic execution is a program analysis technique that was first proposed by King in 1976 [107]. In its most general formulation, symbolic execution executes a program using symbolic instead of concrete inputs. At any point in the computation, the program state consists of a *symbolic state* expressed as a function of the inputs, and the conditions on the inputs that cause the execution to reach that point are typically expressed as a set of constraints in conjunctive form called the *path condition (PC)*. More formally, the symbolic state can be seen as a map $\mathcal{S} : \mathcal{M} \mapsto \mathcal{E}$, where $\mathcal{M}$ is the set of memory addresses for the program, and $\mathcal{E}$ is the set of possible symbolic values, that is, expressions in some theory $\mathcal{T}$ such that all free variables are input values.

Both symbolic state and PC are built incrementally during symbolic execution, with PC initialized to `true`, each input expressed as a symbolic variable, and $\mathcal{S}$ initialized according to the semantics of the language. Each time a statement $stmt$ that modifies the value of a memory location m is executed, the new symbolic value e' of m is computed according to $stmt$'s semantics, and $\mathcal{S}$ is updated by replacing the old expression for m with e' ($\mathcal{S}' = \mathcal{S} \oplus [m \mapsto e']$, where $\oplus$ indicates an update). Conversely, when a predicate statement $pred$ that modifies the flow of control is executed, symbolic execution forks and follows both branches. Along each branch, PC is augmented with an additional conjunct that represents the input condition, expressed in terms of the symbolic state, that makes the predicate in $pred$ `true` or `false` (depending on the branch).

Symbolic execution, when successful, can be used to compute an input that causes a given path to be executed or a given statement to be reached. To do this, at program exit or at a point of interest in the code, the PC for that point is fed to an SMT solver, which attempts to find a solution for the PC. This solution, if found, consists of an assignment to the free variables in PC that satisfies PC, and thus constitutes the desired input.

Some important reasons behind the success of symbolic execution in the last decade are the dramatic increase in the computational power of modern computers, the development of powerful decision procedures that take advantage of such power (*e.g.,* [15, 52, 57, 132, 165]), and a considerable amount of engineering. As a result, we have witnessed an explosion both in the number of techniques that implement different variants of symbolic execution (*e.g.,* [29, 30, 37, 73, 102, 150, 172, 178, 191]) and in the number of (testing) techniques that rely on some form of symbolic execution. The latter are too many to cite, but the interested reader can begin simply by looking at the number of papers that build on KLEE, a symbolic execution technique and tool for C programs [28]. For additional information, we refer the reader to a recent survey on the use of symbolic execution in the context of software testing [31].

Among the different variants of symbolic execution defined since 2000, a particularly successful one is *Dynamic Symbolic Execution (DSE)* (*e.g.,* [73, 172, 191]). DSE addresses one of the main limitations of classic symbolic execution: the inability to handle PCs whose constraints go beyond the theories supported by the underlying constraint solver. To alleviate this problem, DSE performs symbolic execution and concrete execution at the same time. The idea of performing symbolic execution that follows a specific path is not entirely new, as it has been investigated previously (*e.g.,* [38, 81, 110, 139]). However, DSE is novel in its core intuition that symbolic analysis can leverage runtime information to address some of its limitations. In particular, when a PC cannot be solved due to limitations of the constraint solver, DSE can replace symbolic values with concrete values for those constraints that go beyond the capabilities of the solver. This can considerably improve the effectiveness and applicability of the approach [72].

Despite the excitement surrounding symbolic execution, and the amount of novel research it has generated, the extent to which symbolic execution (and DSE in particular) may have lasting practical impact is still unclear. This is due largely to inherent limitations of the approach in the presence of highly structured inputs, programs that interact with external libraries, and large complex programs in general. Another limiting factor is the need for an oracle that can assess whether the program under test behaves correctly for the inputs generated by symbolic execution (see Section 4.2). One recent application of symbolic execution that *has* had considerable practical impact is white-box fuzzing, which operates at the system (rather than unit) level, begins from a set of existing inputs and corresponding paths, and attempts to explore all paths reachable from this initial set in an iterative and systematic manner. One successful instance of white-box fuzzing is Sage [74], which operates at the binary level and has been extremely effective in discovering security vulnerabilities (which can be identified without the need for an oracle) in real-world software of considerable size and complexity.

2.1.2 Search-based Testing

While symbolic execution techniques received the largest number of mentions in our colleagues' responses, where test input generation techniques are concerned, the second largest number of mentions went to research on search-based test input generation techniques, or more generally, search-based software testing (SBST).

Harman and colleagues provide the most recent in a line of surveys on the use of SBST techniques, focusing on their use in software engineering in general [85, 86]. (Several other surveys are also available, including [2, 3, 6, 127, 128].) They point out that a majority of papers on the use of SBST (54%, considering papers through the end of 2008) address topics in test input generation, and that the number of papers on search-based techniques has been increasing regularly, and quite rapidly, ever since the year 2000. They also cite several instances in which industrial organizations such as Daimler, Microsoft, Nokia, Ericsson, Motorola, and IBM have considered the use of SBST techniques.

In general, search-based techniques target optimization problems, such as (in the area of testing) finding the smallest set of test cases that cover all the branches in a program. They do this by employing meta-heuristic search-based optimization techniques, which seek good solutions from among a space of candidate solutions, guided by fitness functions that can differentiate and rank such solutions.

The volume of papers on search-based test input generation is enormous, and the surveys mentioned above provide citations to those papers, so we do not attempt to duplicate that effort here. We do attempt, however, to convey some of the reasons why SBST research has been so prolific and successful to date.

One significant aspect of SBST is the range of testing-related problems to which it has been applied, which include structural testing, model-based testing, mutation testing, temporal testing, exception testing, configuration and interaction testing, stress testing, and integration testing, among others [85]. Most work on SBST, however, has addressed test input generation. A second significant aspect of SBST techniques is that they come in many forms; in the context of test input generation, researchers have utilized a number of approaches, including genetic algorithms (the most popular in the literature to date), simulated annealing, hill climbing techniques, scatter search, particle swarm optimization, and tabu search. Another important contribution in the SBST arena has involved testability transformation, in which program constructs that can be problematic for SBST are transformed into alternative constructs that SBST can handle (*e.g.,* [16, 84, 129]).

In Reference [6], Harman and colleagues describe what they consider to be the greatest open challenges and opportunities for SBST techniques; we summarize that discussion here. A first challenge involves the need for test oracles. This challenge, however, is faced by all test input generation techniques, and we discuss it further in Section 4.2. A second challenge involves combining SBST techniques with symbolic execution techniques, an effort that we describe further in Section 2.1.4. An opportunity for SBST involves co-evolutionary computation [1, 11], in which multiple populations evolve simultaneously, possibly under different fitness functions. Co-evolution models competitive predator-prey relationships; for example, and in relation to testing, it can model the case where test cases (predators) seek out faults (prey). A final opportunity involves "Hyper-heuristic software engineering", which seeks to unite different software engineering activities that utilize SBST, such as test input generation and test case prioritization.

A final area in which research on SBST techniques needs further work, arguably, involves empirical studies. As a case in point, Ali and colleagues [3] survey search-based test input generation techniques, focusing on empirical investigations of such techniques. They identify 64 papers from 1995 to 2007 that use SBST and that present some form of empirical study. While the authors find enough in this work to conclude that SBST, and in particular meta-heuristic techniques, do offer promise, they also identify shortcomings in the state of the art with respect to empirical studies. One shortcoming they cite is that a majority of studies in the time frame considered focus on unit testing and structural coverage, and there is thus limited evidence of the applicability of SBST approaches to other testing phases and types of coverage. The authors also find that the empirical studies they analyzed often did not adequately account for the naturally occurring random variation in technique results and not adequately compare the proposed SBST techniques to simpler, existing alternative techniques.

2.1.3 Random Testing

In addition to dynamic symbolic execution and search-based testing, another automated test input generation technique that has matured considerably in the last decade is random testing (RT). Specifically, we have witnessed an increasing interest among researchers in going beyond straightforward random input generation and investigating more sophisticated, and to some extent principled, approaches that can improve the effectiveness of this traditional technique. This increase in effectiveness is achieved by defining techniques that can either improve the random input generation process (*e.g.,* [35,46,144]) or manage the often overwhelmingly large number of test inputs generated (*e.g.,* [36]).

One example of these new random-testing approaches is adaptive random testing. Adaptive random testing (ART) [35] is a class of testing techniques designed to improve the failure-detection effectiveness of random testing by increasing the diversity of the test inputs executed across a program's input domain. In general, to generate an additional test input, ART techniques first randomly generate a number of candidate test inputs. The techniques then select as the next test input the candidate that is the most "distant" from previously executed test inputs, according to a certain criterion, while other candidates are discarded. Various studies (*e.g.,* [91, 117, 188]) have shown that ART techniques can require substantially fewer test inputs than traditional RT to reveal failures in programs. However, the approach has also been shown to have high overhead [10] and has not yet been extended to handle complex input formats. To address the former problem, researchers have proposed techniques based on mirroring [34], forgetting [32], and Voronoi tessellation [173].

Other representative and well-known examples of random-testing approaches include JCrasher, an automatic robustness tester for Java developed by Csallner and Smaragdakis [46], followup work that combines test input generation and static analysis [47,48], and Randoop, by Pacheco and colleagues [144–146]. Randoop, in particular, improves on traditional random testing by incorporating feedback into the process. More precisely, the technique generates test inputs in an incremental fashion by (1) reusing objects from previous test executions and (2) running and checking test inputs as soon as they are generated. This check determines whether the input should be discarded (*e.g.,* because it is redundant) or kept and used to generate more inputs. Randoop has become the de-facto random-testing tool for Java.

Additional random-testing approaches include work on swarm testing by Groce and colleagues (*e.g.,* [78]), which attempts to increase the diversity of randomly-generated test inputs by using a "swarm" of randomly-generated, incomplete feature sets, and the work on random testing of concurrent programs by Sen [170, 171], which leverages dynamic partial-order techniques and statically identified (potential) data races to drive a random scheduler towards executions that are likely to result in concurrency related failures. Finally, random testing has also been used in combination with dynamic symbolic execution (see Section 2.1.1) to generate inputs that can be used to seed symbolic analysis (*e.g.,* [73]).

2.1.4 Combined Techniques

In addition to specific testing techniques for input generation, in the last decade, researchers have also investigated ways to successfully combine techniques, as well as to combine testing with other types of verification techniques. In this context, a direction of particular interest is the combination of static verification and dynamic verification (*i.e.,* testing). A good representative of this line of work is the Yogi project at Microsoft Research [153], which seeks to combine testing, which under-approximates program behavior (*i.e.,* can produce false negatives) but is effective at discovering errors, and static verification, which is complete but over-approximates program behavior (*i.e.,* can produce false positives), in order to leverage their strengths while reducing their weaknesses. The Synergy [80] and Dash algorithms [18], in particular, combine tests and abstractions so that they benefit each other in an iterative and synergistic fashion: tests guide the refinement of abstractions, while abstractions guide the generation of new test inputs.

Another direction that has produced some initial success and is gaining increasing traction is the combination of symbolic execution (SE) and search-based software testing (SBST). A starting point for this work is the complementary nature of SE and SBST. SE is inherently limited in its ability to handle structured inputs, external libraries, and large and complex programs in general, but can be effectively guided towards a goal, such as the triggering of an assertion. SBST, conversely, can only be guided toward a given goal indirectly, through the use of a fitness function, which can be deceptive and can lead to local minima. SBST, however, is extremely robust with respect to complex and unknown program semantics (*e.g.,* black-box libraries, non-linear expressions, complex data structures, or reflection). Researchers have therefore tried to find and exploit possible synergies between these approaches in several ways, such as by using SE as an additional genetic operator [67, 122], alternating between SE and SBST [95], using fitness to select which path to explore in SE [200], and using symbolic execution to compute fitness values in SBST [14].

2.2 Testing Strategies

A second category of contributions mentioned frequently by our colleagues involved three of what we choose (for lack of a better term) to group under the header, "testing strategies". These include combinatorial testing, model-based testing, and mining and learning from field data. (Note that combinatorial and model-based testing also have, as goals, the generation of test inputs; however, we believe they are better classified as overall strategies.)

2.2.1 Combinatorial Testing

Modern software systems can often be run in an enormous number of different configurations, where configurations involve things such as different settings of parameters and user-configurable values, different environmental settings, or any other factors that may vary and affect system operation. These different configurations need to be considered during testing. Combinatorial testing is a technique that addresses this problem and that has been investigated for over two decades, but with greatly increased intensity since 2000. In fact, Nie and Leung [136], in a survey of the area, found that 77 of 93 papers published on the topic between 1985 and 2008 appeared after 2000.

In practice, testing all of a system's configurations is in most cases impossible due to the sheer size of the configuration space. Testers must therefore find ways to sample such spaces and perform effective testing while containing the cost involved. Combinatorial interaction testing (CIT) offers strategies for doing this. The basic CIT approach involves (1) modeling the system under test as a set of factors that can assume different values, (2) generating a sample of the possible combinations of factors and values, and (3) creating and executing test inputs corresponding to this sample.

The most common approach for performing the second of these three steps, as seen in the research literature, involves the use of covering arrays. A t-way covering array for a system model is a set of configurations in which each valid combination of factor values for every combination of t factors appears at least once [136]. For example, given a system with three parameters, A, B, and C, each of which can take on two values, 1 and 2, eight configurations exist, but a 2-way covering array need only contain four configurations (*e.g.,* A=1, B=1, C=1; A=1, B=2, C=2; A=2, B=1, C=2; A=2, B=2, C=1) to ensure that each factor of each parameter is used with each factor of each other parameter at least once.

Research in the area of combinatorial testing has considered several challenges relevant to its application. One such challenge involves the creation of a model of the system under test to which CIT can be applied, and various approaches for doing so have been suggested (*e.g.,* [77, 111, 119]). At a minimum, this requires the identification of parameters and environmental factors that may influence the system, and the selection of values for these. Also important, however, is the identification of (1) potential interactions that exist between parameters and (2) constraints between parameters and values. Both interactions and constraints can help in selecting combinations of parameters and values that are applicable, as well as combinations that have greater potential to be useful.

A second line of CIT research considered different types of approaches for creating covering arrays. Of course, the mathematics community has generated numerous approaches, which are surveyed in [43, 90]. The area on which software testing researchers have made the greatest impact, however, involves heuristic techniques for generating CIT samples. Initially, researchers considered various greedy algorithms (*e.g.,* [40, 44, 49]). More recently, more sophisticated approaches (*e.g.,* meta-heuristic approaches such as genetic and ant colony algorithms [175], simulated annealing [69], and tabu search [137]) have been utilized (see References [6, 136] for many additional examples.)

The issue of considering constraints and dependencies is another area in which researchers have made much progress (*e.g.,* [26, 41, 76, 168]). A further area of progress involves the development of software tools that implement various CIT approaches; these have been used in extensive empirical studies (*e.g.,* [40, 49, 174]).

Additional topics of research on CIT involve considering different classes of covering arrays. Variable-strength covering arrays, for instance, attempt to improve cost-effectiveness by mixing different selections of "t"; that is, using higher values of "t" for certain subsets of factors that are deemed worth spending more effort on [42]. Test-case-aware covering arrays attempt to address problems associated with systems on which both inputs and environmental factors are present, by considering the two classes of factors separately and then identifying constraints between them [204]. Finally, cost-aware covering arrays take into account the variance in cost between testing different configurations using various approaches, such as approaches that incorporate cost functions [53].

There still remain significant challenges to face in this area. A recent article by Yilmaz describes what several authors active in CIT see as the frontiers for CIT research [205]. In traditional CIT, developers face the difficult task of selecting CIT parameters, such as models and constraints. Yilmaz and colleagues describe the need for approaches that relieve developers of these tasks, such as making CIT incremental and adaptive by dynamically observing program behavior and testing results. Some of the strategies described by the authors for doing this include (1) the use of incremental covering arrays, where testing strength is allowed to increase as resources allow, (2) computing interaction trees using machine-learning techniques and using these to help determine the input space model used in testing, (3) prioritizing the CIT effort in order to cope with resource constraints, and (4) reducing masking effects using adaptive CIT or adaptive error-locating arrays. The authors also recommend that researchers consider the application of CIT to "non-traditional" testing domains, such as in testing GUIs, software product lines, and web applications.

2.2.2 Model-Based Testing

Model-based testing (MBT) involves deriving test suites from models of software systems (see the brief survey by Anand and colleagues [6] and the taxonomy by Utting and colleagues [193]).

MBT can be based on a wide range of models, including different flavors of scenario-based models (*e.g.,* message sequence charts or use case diagrams), various state-oriented notations (*e.g.,* finite state models, UML statecharts, or event flow graphs), and different types of process-oriented models whose notations map to labeled transition systems that describe operational semantics. Models can correspond to or be derived from system specifications, as is the case with UML models, or they can be derived from code or systems themselves, as is the case with models of GUIs such as event flow graphs. Depending on model type there are various ways to generate test inputs. Coverage-based approaches are the most obvious, requiring test cases to cover entities, or sequences of entities, in the models (*e.g.,* states, transitions, or sequences of transitions). Examples of other approaches are cause-effect or disjunctive normal form coverage of post-conditions, coverage of axioms. In any case, such test cases are typically "abstract" and must be translated into executable test cases by specifying appropriate inputs, environmental conditions, and expected outputs.

MBT techniques have had considerable success in industry, where they are used to enable large-scale test automation. Indeed, the abundance of tools that support MBT, both freely available and commercial, provides clear evidence of the success of, and interest in, these approaches. Anand and colleagues [6] summarize three tools that have been available for nearly 10 years: Conformiq Designer [45, 93]), Smartesting CertifyIt [113, 177]), and Microsoft Spec Explorer [75, 180]). In addition, Binder provides a listing of open-source tools for MBT [21]. Further evidence of the success of MBT can be found in the fact that there are companies that sell services built around existing tools (*e.g.,* Smartesting [177]).

One reason for the success of MBT in practice is that, from a practical standpoint, it has several perceived advantages over alternative test generation techniques. Traditional coverage measures, in particular, are not necessarily a reliable proxy for software quality, as they treat all code as equal, and are not an ideal way to drive automated test generation beyond the unit level. MBT, conversely, by relying on domain knowledge, human expertise, and abstraction, can better drive test generation. Once a model has been defined, MBT can allow generation not only of test inputs, but also of oracles. In addition, it can help maintain a picture of the traceability of test cases to requirements.

Some challenges faced by users of model-based testing include (1) difficulty generating models, which is a human intensive activity, (2) issues related to model explosion as the system size or number of system characteristics being modeled increase, and (3) relative lack of empirical understanding of the fault-detection capabilities that may be afforded by different test generation strategies. More generally, another opportunity for future research may lie in approaches for combining dynamic information obtained from system execution with information contained in the models. This sort of approach has been used, for example, in the area of GUI testing, where dynamic event-extraction based techniques have been used to improve either the static models or the test executions derived from those models [79, 124].

2.2.3 Mining and Learning from Field Data

In years past, testing related activities took place primarily in-house—testers ran their test suites, recorded test outcomes, and debugged observed problems in the lab. After deployment, the only information collected involved bug reports explicitly submitted by users. Nowadays, it is increasingly common to collect a broad spectrum of dynamic information from the software after it has been deployed, while it runs on user platforms, and use this field (or telemetry) data to support testing activities and improve their effectiveness.

This change is moving us towards a type of testing, and quality assurance in general, that is increasingly observational in nature. Moreover, this shift is happening not only in academia, but also in industry, where telemetry is collected on a massive scale. In both cases, researchers are studying ways to collect (*e.g.,* [25, 60] and to use field data for a variety of testing-related tasks, such as debugging (*e.g.,* [39, 71, 99, 116], failure reproduction (*e.g.,* [97, 210]), and failure clustering (*e.g.,* [54]).

The increased connectivity and increased computational power of today's computers is pushing this phenomenon even further, by allowing developers to collect an increasingly large amount of data, including execution profiles, program spectra, and failure data, and to use statistical, data mining, and machine learning techniques on these data. Although much of this work is still at an early stage, these techniques promise to better guide the testing process, link it to fault localization, permit more objective assessments of reliability and other software attributes, and permit the uncertainty in such assessments to be characterized. In this context, some important challenges are scalability (albeit a number of these techniques can scale well to large, high-dimensional datasets), the treatment of sensitive information, and the general inability to assess whether an execution in the field terminated correctly or resulted in a (non-crashing) failure.

2.3 Regression Testing

Given program P, modified version P', and test suite T, engineers use regression testing to test P'. Regression testing can be expensive, and the need for cost-effective techniques has helped it emerge as one of the most extensively researched areas in testing over the past two decades. In fact, Yoo and Harman's 2009 survey [206] on work in these areas identifies 159 papers, including 34 on test suite minimization (27 since the year 2000), 87 on regression test selection (45 since 2000), and 47 on test case prioritization (45 since 2000)—and these numbers exclude papers devoted solely to comparative empirical studies. A scan of the contents of journals and conference proceedings since 2009 shows that work in the area continues at a high pace.

To perform regression testing, engineers often begin by reusing T, but reusing all of T (the *retest-all approach*) can be inordinately expensive. *Regression test selection* (RTS) techniques (*e.g.,* [141, 158, 197]) attempt to address this problem by selecting, from test suite T, a subset T' that contains test cases that are important to re-run. When certain conditions are met, RTS techniques can be *safe*; that is, they will not omit test cases which, if executed on P', could reveal faults in P' due to code modifications [156]. *Test case prioritization* (TCP) techniques (*e.g.,* [159, 207]) reorder the test cases in T such that testing objectives (*e.g.,* coverage, fault detection) can be met more quickly. *Test suite reduction* (TSR) techniques (*e.g.,* [89]) attempt to reduce T to some subset, T_{min}, that achieves the same objectives as T (typically, such objectives involve code coverage). Unlike RTS techniques, however, TSR techniques permanently exclude test cases from further runs. The enormous number of papers on the foregoing topics preclude detailed discussion of specific techniques, and interested readers can refer to existing surveys for details (*e.g.,* [23, 206]).

While most of the initial research on RTS, TCP, and TSR focused on creating new techniques, more recently, there has been an increase in research aimed at applying the techniques to different software domains, such as to web applications and web services (*e.g.,* [68, 162]), graphical user interfaces (*e.g.,* [126]), and real-time embedded systems (*e.g.,* [209]). Because the need for regression testing applies across virtually all software domains, we believe that similar opportunities for cross-application will continue to emerge.

RTS, TCP, and TSR techniques all focus on existing test suites and test cases. Reusing existing test suites can be cost-effective, but in general it is not sufficient, because system changes tend to add new functionality, possibly affect existing functionality, and alter test coverage. Recent research has attempted to address this problem by creating *test suite augmentation* (TSA) techniques (*e.g.,* [164, 201]), which aim to identify where new test cases are needed (*e.g.,* code elements in the new program that are new or affected by changes) and possibly help generate them. Seminal work in this area (*e.g.,* [22, 155]) tended to address the identification step, leaving test input generation to engineers. More recent work has focused on automating also the input generation step by utilizing automatic test input generation techniques (*e.g.,* [187]), and in some cases leveraging existing test cases as seeds (*e.g.,* [106, 202]).

Despite the amount of research performed on regression testing, and the many advances made in the family of techniques described above, there is relatively little evidence in regard to the practical application of these techniques in industrial settings. A few papers have examined or reported on experiences with techniques in industrial contexts (*e.g.,* [181, 197]), but the number of such papers is dwarfed by the number of papers proposing new techniques or studying them in lab contexts. A recent survey of practitioners by Engstrom and Runeson [61] suggests, in fact, that practitioners have many concerns beyond those addressed by RTS, TCP, TSR, and TSA techniques. These concerns, which include problems with test automation, test design, test suite maintenance, and change impact assessment, pose potential challenges for future work.

In general, we believe that one reason why research on cost-effective regression testing techniques is still relevant, despite the increasing availability of computing infrastructure via server farms and the cloud, is that test suites tend to expand to use all available resources, as also reported by Google's test engineers [82]. This is especially true as test input generation becomes increasingly automated, as techniques such as random or fuzz testing are systematically applied, and as systems are tested under multitudes of configurations.

One context in which regression testing is likely to be particularly relevant involves continuous integration (see Section 3.2). Organizations interested in continuous integration have learned that it is essential that developers test their code prior to submission, to detect as many integration errors as possible before they can enter the codebase, break builds, and delay the fast feedback that makes continuous integration desirable. In this scenario, regression testing can be especially useful: RTS techniques may help developers decrease the cost of running regression test suites, TCP techniques can help find faults faster, and TSA techniques can help ensure that regression test suites are adequate for the code being tested and guide the generation of additional test inputs otherwise.

Continuous integration is just one example of the many new types of development processes that organizations are attempting to utilize. As these processes proliferate, researchers will need to find ways to adapt existing techniques to those new settings, or create more appropriate, new techniques.

2.4 Empirical Studies and Support for Them

Many of our colleagues cited empirical studies of testing as an area in which prominent advances have been made. Almost as many, however, also described this area as one that continues to offer challenges and opportunities.

Testing techniques are typically heuristics, and their performance can differ widely across different workloads and testing scenarios. To understand and assess these techniques, empirical methods are therefore essential. In the initial decades of work on testing, however, empirical methods were used relatively sparsely. A study of research papers on testing techniques conducted in 2005 [56], considering papers published in the two top software engineering journals and four active software engineering conferences over the years 1994–2003, found that among 224 papers on software testing topics, only 107 (47.7%) reported results of empirical studies. Of these papers, only 37 utilized controlled experiments, 60 utilized case studies (studies of single programs), and 10 presented results more aptly described as illustrations via examples. In addition, most of the studies reported in these papers utilized experiment objects (programs, test suites, faults, and so on) that were not available to other researchers, and most of the controlled experiments performed using publicly available artifacts focused on a handful of small (less than 100 lines of code) C programs known as the Siemens programs. (The Siemens programs were originally introduced to the research community in 1994, by Tom Ostrand and colleagues at Siemens Corporate Research [94].)

In the past ten years, this situation has changed dramatically. In papers submitted to conferences, and even more so in papers submitted to journals, we have witnessed a movement to a near *requirement* that empirical results (of some sort) be included in papers.

There are several reasons behind the change in expectations and research behaviors in the software testing research community. One reason involves the increasing availability of experiment objects. The establishment of objects adequate to support experimentation in testing is a non-trivial task, but the research community has been increasingly rising to meet this challenge. For example, in 2004, the second author of this paper, together with many colleagues, established (and continues to improve) the Software-artifact Infrastructure Repository (SIR) [176]. SIR contains a wide range of artifacts for use in experimentation, including programs written in C, Java, C++, and C#, many available in multiple versions with test suites and fault data. In addition, materials related to software product lines, multithreaded Java programs, container classes, and runtime monitoring of safety properties have recently been added. As of February 2014, over 2100 individuals from over 600 companies or institution have registered as SIR users, and over 580 published papers report utilizing artifacts from the repository.

In addition to SIR, several other repositories of infrastructure supporting experimentation have been established, such as (for example, and by no means exhaustively) iBugs [50], Bugbench [120], the SAMATE Reference Dataset [134], and Marmoset [179].

Established repositories are valuable, in part, because they have the potential to support replicable controlled experimentation. Beyond these repositories, however, open-source systems and systems for which test suites and data on faults are available also provide support for empirical studies. The availability of such repositories and data has also increased substantially over the past ten years, and researchers have increasingly utilized these artifacts for studies. (This effect has been magnified by the recent use of JUnit to create and package test cases with systems, as we also discuss in Section 3.1.) While such artifacts are often more appropriate for case studies than for experiments, given the relative lack of control on processes by which the artifacts are assembled, they can still be particularly useful in providing opportunities to study larger, real-world systems and addressing threats to external validity present in controlled studies performed on smaller artifacts.

A second reason behind the increased expectations for and research utilizing empirical studies in the testing community is related to the increased availability of instrumentation to support experimentation. For example, toolsets for performing program mutation have become increasingly available and robust (*e.g.,* [135, 138, 166]), allowing researchers to inject large numbers of potential faults in programs. While such faults may not be ideal in terms of external validity, there is some evidence that, for experimentation with testing techniques, they can be reasonable surrogates [8]. Moreover, the use of mutants allows researchers to obtain data sets that are sufficient to support statistical analysis, improving the ability to assert that observed effects are significant. Such studies can then be replicated by case studies on larger artifacts containing real faults to address generalizability threats.

We have also witnessed a marked improvement in the manner in which empirical studies are conducted and reported in papers. It seems that the extent to which researchers are aware of and utilize empirical methodologies clearly has increased over the past ten years. In part, this may relate simply to the increased emphasis placed by reviewers on strong studies. We suspect, however, that the ever increasing number of examples of strong studies in the literature, the influence of papers and books whose primary topic concerns the proper conduct of experiments and case studies (*e.g.,* [109, 161, 198]), and the increased feasibility of conducting studies have also played a role.

While the foregoing discussion indicates the range of improvements witnessed in empirical work over the past ten years, there is much more to be done. There is still, in our opinion, an overabundance of studies that focus on small, arguably non-representative, experiment objects. For example, the Siemens programs served a prominent role in moving empirical work forward, but it is time for researchers to move on to more realistic objects of study. Furthermore, as noted above, controlled experiments on somewhat less significant artifacts need to be complemented by larger empirical studies of more substantial artifacts, and ideally studies should ultimately be conducted on industrial systems. Finally, studies in which different engineering processes are utilized are needed, to evaluate the effects of process differences on the cost-effectiveness, and even viability, of techniques.

A second area in which improvements are needed involves user studies. Certainly, much of the research conducted in software testing has to do with algorithms and heuristics that can be automated, and studying the cost-effectiveness of applying such techniques to various workloads is an important first step in assessing their worth. Most of the techniques being created in the research community, however, are destined to be employed by software engineers. The fact that a particular technique displays a particular level of effectiveness relative to some metric does not imply that it will be practical or cost-effective in the hands of engineers. As a case in point, research on fault localization techniques focused for years on the ability of techniques to reduce the number of program statements that might be labeled as suspicious, given a set of (passed and failing) program runs. However, only recently has any attempt been made to assess whether the data produced by these techniques can actually guide engineers in their localization of faults. This attempt actually casts doubt on the efficacy of these approaches, and hence, on the appropriateness of the metric being used and the assumptions being made in earlier studies [147].

The costs of performing more extensive studies, studies in industry contexts, and user studies are high, and some may argue that the research community does not currently provide sufficient rewards for these. However, we believe that without the results of such studies, it will be much more difficult for us, as a research community, to make the potential impact of our work apparent, and to see the results of that work transferred into practice.

3. PRACTICAL CONTRIBUTIONS

In addition to research contributions, the software testing area has also witnessed important improvements in the state of the practice. Two major contributions identified by our colleagues, in this context, are the definition of new frameworks for test execution and the widespread adoption of innovative practices that promote shorter cycles in the testing process, such as continuous integration.

3.1 Frameworks for Test Execution

Although we would not classify them as research contributions, frameworks for test execution have dramatically improved the state of the art in software testing. In addition, they have also indirectly affected research, as we discuss in the rest of this section.

Arguably, the most famous and widespread framework for automating test execution is JUnit [100]. JUnit is a relatively straightforward framework that allows developers to write repeatable tests in what has became a de-facto standard format. In fact, today JUnit is supported in many IDEs (*e.g.,* Eclipse [59]), has spawned a number of similar frameworks for other languages referred to collectively as xUnit (*e.g.,* NUnit for C#, PHPUnit for PHP), and has become almost a synonym for unit testing.

In a nutshell, xUnit frameworks provide a standard way to encode the four fundamental parts of a (unit) test case: setting the initial state, invoking the functionality under test, checking the results of the test, and performing any necessary cleanup. For example, a JUnit test case for the `characterRead` method of a `File` class may (1) create an instance of `File`, associate it with an existing file f, and open it (setting the initial state), (2) invoke `characterRead` to read one character from f (invoking the functionality under test), (3) assert that the character read is the first character of f (checking the results of the test), and (4) close the file (performing any necessary cleanup).

Frameworks for automated test execution have become particularly popular in the context of agile development processes. One of the cornerstones of agile processes is, in fact, the notion of testing early and testing often; test cases represent immediate feedback that can tell developers whether their changes introduced any regression errors or whether the code they just wrote satisfies a given requirement (previously encoded in the form of tests). For example, it is common practice to use JUnit in the context of eXtreme Programming (XP) [17] or Scrum [167] processes.

As noted at the beginning of this section, frameworks for test execution also affected, and in a sense aided, software testing research. The availability of open source software that is released together with JUnit test cases for that software, for example, has enabled researchers not only to perform empirical evaluations on more realistic systems (as noted in Section 2.4), but also to define their tools in a de-facto standard way. In fact, a tool that can operate on JUnit test cases can ideally be run on the many programs for which that type of test cases is available. It is also not uncommon, for researchers in the area of test input generation, to encode their test cases using some standard framework, which can facilitate adoption of their technique and foster sharing of research results.

3.2 Continuous Integration

Another practical contribution to software testing is the practice of continuous integration (CI) [58, 64]. The basic idea behind CI is to commit, one or more times a day, all of the working copies of the software on which different developers or groups of developers are working. CI is related to the concept of automated test execution frameworks, in that a regression test suite should be automatically run against the code (ideally, prior to commit it) to help ensure that the codebase remains stable (*i.e.,* no regression errors have been introduced) and continuing engineering efforts can be performed more reliably. If some of the tests fail, the developers responsible for the changes must then correct the problems revealed by the tests. This approach is advantageous because it can reduce the amount of code rework that is needed in later phases of development, and speed up overall development time.

CI was first proposed in 1997 by Kent Beck and Ron Jeffries (who, together with Ward Cunningham, were also the main creator of extreme programming (XP) [17]), and represents a clear departure from more traditional development and testing practices. Traditionally, integration was performed at the end of a possibly long cycle, and numerous incompatibility problems among the modules to be integrated typically arose—this is informally known as "integration hell". CI is now common practice in many organizations that create software, such as Google (where it is performed at an incredibly large scale), Mozilla (with its Tinderbox software [133]), and many others. These companies are increasingly relying on CI to improve their product development. Consequently, tools for supporting CI are becoming more and more widespread (*e.g.,* [13, 96, 190]). In some contexts, entire computer farms are devoted to running build servers that perform intensive testing of the production code in the main repository and update a dashboard where developers can check the status of the code in the system.

Even when large server farms or the cloud are employed in testing, however, CI poses challenging (regression) testing problems (*e.g.,* [82]). Developers must conform to the expectation that they will commit changes frequently, typically at a minimum of once per day, but in many cases even more often. Version control systems must support atomic commits, in which sets of related changes are treated as a single commit operation, to prevent builds from being attempted on partial commits. Testing must be automated and must involve automated oracles. Finally, test infrastructure must be robust enough to continue to function in the presence of significant levels of code churn.

Another challenge, in the context of CI, is how to shorten the save-compile-check cycle and move the checking even closer to the IDE, so that developers can (ideally) obtain feedback as they code. This would be similar to what already happens today, in common IDEs, for syntactic checks, but it would be at an increasingly semantic level. In more general terms, automated testing, CI, and other related practices provide clear evidence of how the testing process has improved in practice. Software testing has gone from being an ad-hoc, often neglected activity assigned to novice developers to a systematic and often sophisticated approach that is (1) considered essential for the quality of the software, (2) tightly integrated into the development process, and (3) performed by skilled professional [61, 160].

4. CHALLENGES AND OPPORTUNITIES

As noted in Section 1, our colleagues also provided input on areas that, while not yet seen as sources of substantial research contributions, offer new challenges and opportunities for researchers. Here, we discuss the areas that were most salient in their responses.

It is worth noting that, in this section, we focus on *research* challenges. That is, we decided not to discuss challenges that, albeit mentioned in some of the responses we received, were primarily related to human factors (*e.g.,* developer habits and mentality) and technology transfer (*e.g.,* transition of research to practice or better communication with industry).

4.1 Testing Modern, Real-World Systems

Many testing techniques, and especially the academic techniques that have not yet found their way into industrial practice, target traditional software—software that is written in a single language, homogeneous, non-distributed, and in some cases even small-sized. Unfortunately, many of today's software systems are very different from these traditional systems, and have characteristics that render them unsuitable for existing testing techniques. Specifically, many modern software systems consist of components of diverse nature and provenance, with different degrees of coupling with one another, and they are often distributed and highly dynamic. Numerous classes of applications that are increasingly popular, such as mobile applications, web applications, software product lines, service-oriented architectures, and cloud-based applications, have these characteristics.

To illustrate, consider web applications. A web application typically includes a set of server-side components that may consist of a mix of Java servlets, PHP scripts, and XML configuration files, hosted on an application server whose behavior and performance depend on a myriad of configuration parameters. In addition, these components often interact with a database server that may be hosted on a different machine and have, itself, a complex configuration space. A web application also typically has a set of client-side components that consist of a mix of dynamically generated HTML code and scripts that must run on various browsers and platforms.

In general, the characteristics of modern systems can render them extremely problematic for existing testing approaches. Heterogeneity, rich environments, and high configurability make it difficult to model a system in its entirety, as the techniques used to define and build these models are usually defined for single-language, self-contained systems. These characteristics also make it difficult to identify differences between versions of a system, a key element in understanding system evolution and performing regression testing; techniques for testing evolving software tend to assume that a change consists of a modification of the code and even make specific assumptions on the invariability of the environment (*e.g.,* [141, 157]). The characteristics of modern systems can even invalidate the traditional concept of coverage—another notion that is fundamental to existing testing techniques. Whereas structural coverage is well defined for traditional code, for instance, it is not clear how it should be defined for a database, a remote service, or a set of configuration files. Finally, heterogeneity and environment dependence affect the ability to perform impact analysis, yet another key technique that is heavily used during maintenance.

In summary, many existing testing techniques fall short when applied to commonly-used modern software systems, leading to ineffective testing and, ultimately, to poor software quality. It is true that there is increasingly greater attention being paid, by the research community, to the problems involved in testing these applications (*e.g.,* [4, 12, 123]). In practice, however, testing of these systems is often performed in ad-hoc, inadequate ways, which can have dramatic consequences (*e.g.,* Amazon's outage in 2008, which cost the company dearly [192]). One important challenge for researchers in this area is therefore to define techniques that can go beyond the state of the art and be applied in the real world and on modern software, regardless of its complexity and size.

4.2 Oracles

Long ago recognized as a significant problem [196], the "oracle problem"—the problem of determining the correctness of a program's behavior under test—is still relevant today. In fact, active work specific to test oracles is a relatively recent phenomenon [185], and several authors have recently discussed the need to focus on test oracles when evaluating the quality of the testing process [6, 185]. (A recent survey by Harman and colleagues provides a comprehensive discussion of the state of the art in test oracle research [87].)

Overall, although there have been some initial research efforts in this direction, the problem of constructing oracles in an automated or semi-automated fashion is still by and large open.

Some researchers have worked on generating oracles for regression testing. Xie and Memon, in particular, explore methods for constructing test oracles for GUI systems, yielding several recommendations [130, 199]. In addition, several tools exist for automatically generating invariant-based test oracles for use in regression testing, including Eclat [144] and DiffGen [186]. These approaches assume that the program is currently correct and identify differences in behavior between the current version and the next.

In other work, several authors have proposed methods for inferring invariants from programs, with the potential for using them as oracles in testing [62,195]. Fraser and colleagues propose $\mu TEST$, which generates JUnit test cases for object oriented programs [66]. Both of these approaches assume that the tester will later manually correct the generated test oracles. Work evaluating this paradigm with users is mixed, but overall, somewhat discouraging [65, 183].

A third class of techniques attempts to support, rather than completely automate, the creation of test oracles. Staats and colleagues propose a mutation-based approach for selecting oracle data based on how often a variable reveals a fault in a mutant [182]. This work's limitations are scalability and the need to estimate the number of required mutants to select effective oracle data. Loyola and colleagues [182] propose an approach meant to assist engineers in constructing expected value oracles—oracles that specify, for a single test input, the concrete expected value for one or more program values. Finally, Pastore and colleagues [148] propose CrowdOracles, an approach to use crowdsourcing for checking assertions. These approaches all aim to leverage added semantic knowledge that can be provided by engineers or system users—if appropriately qualified engineers or users can be found. However, they will face problems of scale in cases where automatic test input generation tools are being used to generate enormous numbers of test inputs.

Overall, as test case generation approaches improve, and the ability to automatically generate large volumes of test cases increases, the need for approaches that can determine whether test cases elicit proper program behavior will become even more significant. As a consequence, the oracle problem will continue to be a significant challenge.

4.3 Probabilistic Program Analysis

Since 2000, we have witnessed the growth of a research area that has generated considerable interest and that we believe may play an important role in the area of testing in the future: probabilistic program analysis, performed in various flavors (*e.g.,* [92, 131, 163]). The basic idea behind these probabilistic analysis approaches is to go beyond traditional program analysis, which typically tries to determine whether a given behavior can occur or not in a program. Probabilistic program analysis, conversely, aims to quantify the probability of an event of interest, such as the failure of an assertion, to occur. In more recent years, researchers have applied this idea to symbolic execution [70]. Whereas traditional symbolic execution analyzes paths within a program and uses a constraint solver to decide whether a given path is feasible, probabilistic symbolic execution assigns an estimated probability to each path.

In general, probabilistic program analysis has a number of potential applications in the context of testing. For instance, the (estimated) probabilities associated with a program's paths could be used to decide where to focus the testing effort. As another example, the same probabilities could be use to compute the expected reliability of a program (*e.g.,* [63]). Although promising, however, this area of investigation is extremely challenging, due to the inherent cost and limitations of the techniques on which these approaches rely (*e.g.,* model counting [51], quantification of the solution space for a set of constraints in general [24]). Making these techniques scale and work on realistic programs will therefore require considerable effort, but we believe that the benefits that could be obtained may be well worth it.

4.4 Testing Non-Functional Properties

Much of the research on testing to date has focused on testing for functional correctness, known in some quarters as "debug testing". There are, however, other properties of software systems that engineers may wish to assess by means of testing, many of which have received relatively little attention in the research literature to date.

In particular, one property cited several times by our colleagues was "performance". Performance issues can arise with virtually all software systems, but they are of paramount importance for some classes of systems. These classes include web applications and web services, for which response time can be a key element in user adoption, and for which a wide range of monographs (*e.g.,* [149]) and both commercial tools (*e.g.,* [98, 189]) and open-source tools (for a list, see [140]) are available. They also include hard real-time systems, which have been an issue of concern for some time, and whose importance is growing in par with the increasing relevance and widespread use of mobile platforms. Several researchers have proposed approaches for estimating worst-case interrupt latencies (WCILs) based on analyses of systems (*e.g.,* [104, 169]). However, these approaches are conservative and can over-approximate WCILs, which affects the precision and applicability of their results. Testing-based approaches, in contrast, may underestimate WCILs, but there is some evidence that these approaches, when coupled with certain static analysis techniques, can provide accurate assessments [108, 151, 208].

Another example of a non-functional property that has begun to garner attention is energy consumption. Researchers have proposed various testing and analysis techniques for verifying this property (*e.g.,* [20, 83, 114, 118]). Researchers have also considered issues involving the efficient use of power while running regression tests [101] and the use of power by test suites that run in-situ on devices to provide self-check capabilities [115]. We believe that this area will be increasingly important in the years to come, especially in the context of mobile computing platforms, such as mobile phones and tablets, unmanned aerial vehicles, and sensor devices.

4.5 Domain-Based Testing

When new programming paradigms or technologies emerge and become successful, inevitably, researchers turn their attentions to them. Thus, in recent years we have witnessed researchers propose testing techniques that address relatively new application domains, such as component-based systems, web applications, mobile applications, and so on. Clearly, the need to address emerging languages and application classes will continue, and we believe that this will remain one of the larger opportunities for testing researchers. Researchers are considering techniques for testing dynamic multi-tier web applications (*e.g.,* [4, 12, 123]), for example, but this is still a relatively young field, as one might argue is research on techniques for testing software product lines (*e.g.,* [103, 142, 194]) and testing of Android applications (*e.g.,* [5, 7, 121]).

One area of domain-based testing research that we believe will offer opportunities to researchers relates to end-user programming systems. End-user programmers are individuals who have not been trained in software engineering, but who nonetheless create software that may eventually be used beyond its expected lifetime and scope. Spreadsheets were among the first end-user programming paradigms, and have been a subject of research where testing techniques are concerned (*e.g.,* [154]). Nowadays, languages supporting end-user programmers are proliferating. In particular, we are now seeing end-user programmers writing software in highly popular and commercially successful environments, such as Matlab [125] and Labview [112], and in open-source programming environments, such as those provided by App Inventor [9], R [152], and Yahoo! Pipes [203]. The increasing availability of programming environments that are accessible to and usable by non-professionals will surely contribute to the democratization of computing, but it will also contribute to the democratization of bugs. Addressing this problem will require research on methods for helping and supporting end-user programmers in their efforts without requiring them to actually become software engineers.

4.6 Leveraging the Cloud and the Crowd

The cloud has becoming pervasive in the last decade. We have moved from a situation in which most of our computational power and data resided on local machines to one in which our data are increasingly remote, transparently stored in data centers, and a great deal of computation also occurs on remote servers. It is only natural to begin using the cloud for testing, and in fact we are already witnessing research in this direction (*e.g.,* [27, 105, 184]). A noteworthy example of such efforts involves *Cloud IDEs*; that is, integrated development environments that allow developers to code, build, and test in the cloud, through a web interface. Clearly, cloud IDEs could untap the potential of testing techniques that rely on analyses that are often too expensive to be performed on a single machine (*e.g.,* test input generation or heavy-weight static analyses).

Another opportunity that is loosely related to cloud computing, but pushes the boundary even further, is the use of crowd-sourcing to improve software testing. Despite the great deal of effort devoted to automating most testing tasks, from test input generation to oracle creation, testing is still a human intensive activity in most cases. It therefore makes sense to try to leverage the crowd in the context of testing. Thus far, researchers have just scratched the surface of what might be achieved by crowd-sourcing verification and validation tasks (*e.g.,* [33,55,148]), but we expect that this area will attract increasing attention in the upcoming years. One challenge, in this context, will be to find ways to encode testing problems in ways that humans can handle. This may involve, for instance, encoding problems as games, but then, these must be games that real humans will be willing to play. Moreover, and clearly, problems must be encoded in such a way that "solving" the game is not as difficult as solving the original problem.

5. CONCLUSION

Our goal, when we set out to write this paper, was to present and discuss (some of) the most successful software testing research performed since the year 2000. It is undoubtedly difficult, if not impossible, to summarize in any complete way almost 15 years of research and cite all relevant papers in a relatively short report such as this one. We therefore did not attempt to cover all relevant themes and efforts, but rather focused on those that our colleagues and we thought were particularly relevant, had already had a considerable impact, or seemed likely to have impact in the (near) future.

We hope that interested readers will find our reflections useful and thought-provoking. We welcome comments and feedback, and look forward to seeing, in the next FOSE paper on software testing, what the "state of the art in software testing" will be then.

Acknowledgements

This work was partially supported by NSF awards CCF-1320783, CCF-1161821, and CCF-0964647, and by funding from Google, IBM Research and Microsoft Research to Georgia Tech, and by the AFOSR through award FA9550-09-1-0129 to the University of Nebraska - Lincoln. We are extremely grateful to all of our colleagues who found time to respond (in several cases quite extensively) to our request for input on contributions and challenges in the area of software testing. They are too many to mention by name, but our most heartfelt thanks go to them all.

6. REFERENCES

[1] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *GECCO (2), Volume 3103 of Lecture Notes In Computer Scienc*, pages 1338–1349. Springer, 2004.

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

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

[4] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *Proceedings of Automated Software Engineering*, pages 3–12, 2011.

[5] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A GUI crawling-based technique for Android mobile application testing. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 252–261, 2011.

[6] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, Aug. 2013.

[7] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 59:1–59:11, 2012.

[8] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering*, pages 402–411, May 2005.

[9] Mit app inventor. http://appinventor.mit.edu/explore/.

[10] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 265–275, July 2011.

[11] A. Arcuri and X. Yao. Coevolving programs and unit tests from their specification. In *In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 397–400, 2007.

[12] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of Javascript web applications. In *Proceedings of the International Conference on Software Engineering*, pages 571–580, 2011.

[13] Atlassian. Atlassian software systems: Bamboo. https://www.atlassian.com/software/bamboo.

[14] A. I. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. E. J. Vos. Symbolic search-based testing. In *Proceedings of the International Conference on Automated Softwaare Engineering*, pages 53–62, 2011.

[15] J. Bailey and P. J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Proceedings of the International Conference on Practical Aspects of Declarative Languages*, pages 174–186, 2005.

[16] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability

transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 108–118, 2004.

[17] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[18] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 3–14, 2008.

[19] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103, 2007.

[20] A. Bertolino, G. DeAngelis, and A. Sabetta. VCR: Virtual capture and replay for performance testing. In *Proceedings of Automated Software Engineering*, pages 399–402, Nov. 2008.

[21] R. V. Binder. Open source tools for model-based testing. http://robertvbinder.com/open-source-tools-for-model-based-testing.

[22] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8), Aug. 1997.

[23] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. Regression test selection techniques: A survey. *Informatica*, 35:289–321, 2011.

[24] M. Borges, A. Filieri, M. d'Amorim, C. Pasareanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[25] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 2–8, Nov. 2002.

[26] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960–970, Oct. 2006.

[27] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, pages 183–198, 2011.

[28] C. Cadar. KLEE-related publications and systems. `http://klee.github.io/klee/Publications.html`, 2014.

[29] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 209–224, 2008.

[30] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the SPIN Symposium on Model Checking of Software*, pages 2–23, 2005.

[31] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the International Conference on Software Engineering*, pages 1066–1071, 2011.

[32] K.-P. Chan, T. Y. Chen, and D. Towey. Forgetting test cases. In *Proceedings of the International Computer Software and Applications Conference*, volume 1, pages 485–494, 2006.

[33] N. Chen and S. Kim. Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles. In *Proceedings of the International Conference on Automated Software Engineering*, pages 140–149, 2012.

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

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

[36] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

[37] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems (HotDep)*, 2009.

[38] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, 1985.

[39] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proceedings of the International Conference on Software Engineering*, pages 261–270, 2007.

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

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

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

[43] C. Colbourn. Combinatorial aspects of covering arrays. *Le Matematich (Catania)*, 58:121–157, 2004.

[44] C. J. Colbourn and M. B. Cohen. A deterministic density algorithm for pairwise interaction coverage. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 242–252, 2004.

[45] Conformiq. http://www.conformiq.com.

[46] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software Practice and Experience*, 34(11):1025–1050, Sept. 2004.

[47] C. Csallner and Y. Smaragdakis. Check'n'crash: combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering*, pages 422–431. ACM, 2005.

[48] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):8:1–8:37, 2008.

[49] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proceedings of the Pacific Northwest Sofware Quality Conference*, 2006.

[50] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of Automated Software Engineering*, pages 433–436, Nov. 2007.

[51] J. A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation*, 38(4):1273–1302, 2004.

[52] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[53] G. Demiroz and C. Yilmaz. Cost-aware combinatorial interaction testing. In *Proceedings of the Internatinoal Conference on Advances in System Testing and Validation Lifecycles*, pages 9–16, Nov. 2012.

[54] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the International Conference on Software Engineering*, pages 339–348, 2001.

[55] W. Dietl, S. Dietzel, M. D. Ernst, N. Mote, B. Walker, S. Cooper, T. Pavlik, and Z. Popović. Verification games: Making verification fun. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs*, pages 42–49, 2012.

[56] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[57] B. Dutertre and L. de Moura. The YICES SMT Solver.

[58] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, 2007.

[59] Eclipse Foundation. Eclipse. `http://www.eclipse.org/`, 2014.

[60] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 65–75, 2004.

[61] E. Engström and P. Runeson. A qualitative survey of regression testing practices. In *Proceedings of the International Conference on Product-Focused Software Process Improvement*, pages 3–16, 2010.

[62] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[63] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 622–631, Piscataway, NJ, USA, 2013. IEEE Press.

[64] M. Fowler. Continuous integration. martinflowler.com/articles/continuousIntegration.html.

[65] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 188–198, July 2013.

[66] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[67] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *Proceedings of the International Symposium on Softaware Reliability Engineering*, pages 360–369, 2013.

[68] D. Garg and A. Datta. Test case prioritization due to database changes in web applications. In *Proceedings of the International Conference on Software Testing*, pages 726–730, Apr. 2012.

[69] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, Feb. 2011.

[70] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 166–176, 2012.

[71] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, pages 103–116, 2009.

[72] P. Godefroid. Higher-order test generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–269, 2011.

[73] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[74] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[75] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: Tools and methodology. *Journal of Software Testing, Verification, and Reliability*, 21:55–71, 2010.

[76] W. Grieskamp, X. Qu, X. Wei, N. Kicillof, and M. B. Cohen. Interaction coverage meets path coverage by smt constraint solving. In *Joint Conference of the IFIP International Conference on Testing of Communicating Systems and International Workshop on Formal Approaches to Testing of Software*, 2009.

[77] M. Grindal and J. Offutt. Input parameter modeling for combination strategies. In *Proceedings of the Conference on IASTED International Multi-Conference: Software Engineering*, pages 255–260, 2007.

[78] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 78–88, 2012.

[79] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: High coverage, no false alarms. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 67–77, 2012.

[80] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–127, 2006.

[81] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 219–227, September 2000.

[82] P. Gupta, M. Ivey, and J. Penix. Testing at the speed and scale of google. `http://googletesting.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html`, June 2011.

[83] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE 2013)*, pages 92–101, 2013.

[84] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan. 2004.

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

[86] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61, Dec. 2012.

[87] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical report, Research Memoranda CS-13-01, Department of Computer Science, University of Sheffield, 2013.

[88] M. J. Harrold. Testing: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 61–72, 2000.

[89] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[90] A. Hartman and R. L. Problems and algorithms for covering arrays. *Discrete Mathematics*, 248:149–156, 2004.

[91] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology*, 22(1):6:1–6:42, 2012.

[92] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444, 2006.

[93] A. Huima. Implementing Conformiq Qtronic. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12. 2007.

[94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, May 1994.

[95] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the International Conference on Automated Softwaare Engineering*, pages 297–306, 2008.

[96] Jenkins. Jenkins: An extendable open source continous integration server. jenkins-ci.org.

[97] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *Proceedings of the International Conference on Software Engineering*, pages 474–484, 2012.

[98] Apache JMeter. https://jmeter.apache.org.

[99] J. A. Jones, A. Orso, and M. J. Harrold. Gammatella: Visualizing program-execution data for deployed software. *Information Visualization*, 3(3):173–188, 2004.

[100] JUnit Testing Framework. `http://www.junit.org`, 2014.

[101] E. Y. Kan. Energy efficiency in testing and regression testing: a comparison of DVFS techniques. In *Proceedings of the International Conference on Quality Software*, pages 280–283, 2013.

[102] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, 2003.

[103] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 57–68, 2011.

[104] J. Kim, H. Oh, H. Ha, S.-H. Kang, J. Choi, and S. Ha. An ILP-based worst-case performance analysis technique for distributed real-time embedded systems. In *Proceedings of the Real-time Systems Symposium*, pages 363–372, 2012.

[105] M. Kim, Y. Kim, and G. Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. In *Proceedings of the International Conference on Software Testing*, pages 340–349, Apr. 2012.

[106] Y. Kim, Z. Xu, M. Kim, M. B. Cohen, and G. Rothermel. Hybrid directed test suite augmentation: An interleaving framework. In *Proceedings of the International Conference on Software Testing*, Apr. 2014.

[107] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[108] R. Kirner, P. Puschner, and I. Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proceedings of the IEEE Workshop on Software Technology for Future Embedded and Ubiquitous Systems*, pages 7–10, 2004.

[109] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, Aug. 2002.

[110] B. Korel. A Dynamic Approach of Test Data Generation. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 311–317, November 1990.

[111] R. Krishnan, S. M. Krishna, and P. S. Nandhan. Combinatorial testing: Learnings from our experience. *SIGSOFT Software Engineering Notes*, 32(3):1–8, May 2007.

[112] LabView System Design Software. http://www.ni.com/labview/.

[113] B. Legeard and M. Utting. Model-based testing - next generation functional testing. *SoftwareTech News*, 12(4):9–18, Jan. 2010.

[114] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89, 2013.

[115] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. J. Halfond. Integrated energy-directed test suite optimization. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2014.

[116] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[117] Y. Lin, X. Tang, Y. Chen, and J. Zhao. A divergence-oriented approach to adaptive random testing of Java programs. In *Proceedings of the International Conference on Automated Software Engineering*, pages 221–232, 2009.

[118] Y. Liu, C. Xu, and S. C. Cheung. Where has my battery gone? Finding sensor related energy black holes in smartphone applications. In *Proceedings of the International Conference on Pervasive Computing and Communications*, pages 2–10, 2013.

[119] C. Lott, A. Jain, and S. Dalal. Modeling requirements for combinatorial software testing. In *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, pages 1–7, 2005.

[120] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[121] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 224–234, 2013.

[122] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *Proceedings of the International Conference on Automated Softwaare Engineering*, pages 436–439, 2011.

[123] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *Proceedings of the International Conference on Software Testing*, pages 121–130, 2008.

[124] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. AutoBlackTest: A tool for automatic black-box testing. In *Proceedings of the International Conference on Software Engineering*, pages 1013–1015, 2011.

[125] Matlab. http://www.mathworks.com/products/matlab/.

[126] S. McMaster and A. Memon. Call-stack covereage for GUI test suite reduction. *IEEE Transactions on Software Engineering*, 34(1):99–115, 2008.

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

[128] P. McMinn. Search-based software testing: past, present, and future. In *Proceedings of the International Workshop on Search-Based Software Testing*, pages 153–163, Mar. 2011.

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

[130] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *Proceedings of Automated Software Engineering*, pages 164–173, Nov. 2003.

[131] D. Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–101, 2001.

[132] A. Morgado, F. Heras, and J. Marques-Silva. Improvements to Core-guided Binary Search for MaxSAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 284–297, 2012.

[133] Mozilla Developer Network. Tinderbox. `https://developer.mozilla.org/en-US/docs/Tinderbox`, 2014.

[134] National Institute of Standards and Technology. Nist software assurance reference dataset project. samate.nist.gov/SRD/.

[135] Nester. nester.sourceforge.net.

[136] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11:1–11:29, Feb. 2011.

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

[138] J. Offut, Y.-S. Ma, and Y.-R. Kown. MuJava: An automated class mutation system. *Journal of Software Testing, Verification, and Reliability*, 15(2):97–133, June 2005.

[139] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Software Practice and Experience*, 29(2):167–193, 1997.

[140] Open Source Software Testing Tools. www.opensourcetesting.org/performance.php.

[141] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 241–252, Nov. 2004.

[142] S. Oster, M. Zink, M. Lochau, and M. Grechanik. Pairwise feature-interaction testing for SPLs: potentials and limitations. In *Proceedings of the International Conference on Software Product Lines*, pages 6:1–6:8, 2011.

[143] L. Osterweil. Strategic directions in software quality. *ACM Computing Surveys*, 4:738–750, Dec. 1996.

[144] C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. *Proceedings of the European Conference on Object-Oriented Programming*, pages 504–527, 2005.

[145] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random Testing for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 815–816, 2007.

[146] C. Pacheco, S. K. Lahiri, and T. Ball. Finding Errors in .NET with Feedback-Directed Random Testing. In *Proceedings of the 2008*

International Symposium on Software Testing and Analysis, pages 87–96, 2008.

[147] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 199–209, July 2011.

[148] F. Pastore, L. Mariani, and G. Fraser. CrowdOracles: Can the crowd solve the oracle problem? In *Proceedings of the International Conference on Software Testing*, Apr. 2013.

[149] Performance Testing Guidance for Web Applications. perftestingguide.codeplex.com/releases/view/.

[150] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 34–44, 2011.

[151] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Proceedings of the real-time systems symposium*, pages 134–143, Dec. 1998.

[152] The r project for statistical computing. http://www.r-project.org/.

[153] M. Research. The yogi project, 2014. http://research.microsoft.com/en-us/projects/Yogi.

[154] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10(1):110–147, Jan. 2001.

[155] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 169–184, Aug. 1994.

[156] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug. 1996.

[157] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, Apr. 1997.

[158] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ programs. *Journal of Software Testing, Verification, and Reliability*, 10(2):77–109, June 2000.

[159] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.

[160] P. Runeson. A survey of unit testing practices. *IEEE Softw.*, 23(4):22–29, July 2006.

[161] P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering*. John Wiley and Sons, Hoboken, NJ, 2012.

[162] S. Sampath, R. C. Bryce, G. Viswanath, and V. Kandimalla. Prioritizing user-session-based test cases for web application testing. In *Proceedings of the International Conference on Software Testing*, pages 141–150, Apr. 2008.

[163] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. *SIGPLAN Notices*, 48(6):447–458, June 2013.

[164] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of Automated Software Engineering*, Sept. 2008.

[165] SAT4J. `http://www.sat4j.org`, 2012.

[166] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 297–298, 2009.

[167] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[168] I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 254–264, 2011.

[169] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static WCET analysis of real-time task-oriented code in vehicle control systems. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 212–219, 2006.

[170] K. Sen. Effective random testing of concurrent programs. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 323–332, 2007.

[171] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[172] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272, 2005.

[173] A. Shahbazi, A. F. Tappenden, and J. Miller. Centroidal Voronoi tessellations — A new approach to random testing. *IEEE Transactions on Software Engineering*, 39(2):163–183, 2013.

[174] G. B. Sherwood, S. S. Martirosyan, and C. J. Colbourn. Covering arrays of higher strength from permutation vectors. *Journal of Combinatorial Design*, 3(14):202–213, 2005.

[175] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the Annual International Computer Software and Applications Conference*, pages 72–77, 2004.

[176] Software-artifact Infrastructure Repository. `http://sir.unl.edu/`, Apr. 2012.

[177] Smartesting. http://www.smartesting.com.

[178] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information Systems Security*, pages 1–25, 2008.

[179] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software repository mining with marmoset: An automated programming project snapshot and testing system. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[180] Spec explorer. http://www.specexplorer.net.

[181] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2002.

[182] M. Staats, G. Gay, and M. Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the International Conference on Software Engineering*, pages 870–880, May 2012.

[183] M. Staats, S. Hong, M. Kim, and G. Rothermel. Understanding user understanding: Determining correctness of generated program invariants. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 188–198, July 2012.

[184] M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 183–194, July 2010.

[185] M. Staats, M. W. Whalen, and M. Heimdahl. Programs, tests, and oracles: The foundations of testing revisited. In *Proceedings of the International Conference on Software Engineering*, pages 391–400, 2011.

[186] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of Automated Software Engineering*, pages 407–410, Nov. 2008.

[187] K. Taneja, T. Xie, N. Tillmann, J. Halleux, and W. Schulte. eXpress: Guided path exploration for regression test generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2011.

[188] A. F. Tappenden and J. Miller. A novel evolutionary approach for adaptive random testing. *IEEE Transactions on Reliability*, 58(4):619–633, 2009.

[189] Test Studio. www.telerik.com/teststudio/performance-testing.

[190] ThoughtWorks. Go: Continous delivery. www.thoughtworks.com/products/go-continuous-delivery.

[191] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *Proceedings of the International Conference on Tests and Proofs*, pages 134–153, 2008.

[192] TransWorldNews. Amazon.com Down for Hours After Unknown Outage. `http://www.transworldnews.com/NewsStory.aspx?storyid=49790`, 2010.

[193] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Journal of Software Testing, Verification, and Reliability*, 22(5):297–312, 2012.

[194] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. Testing software product lines using incremental test generation. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 249–258, 2008.

[195] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, May 2011.

[196] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 15(4):465–470, 1982.

[197] L. White and B. Robinson. Industrial real-time regression testing and analysis using firewalls. In *Proceedings of the International Conference on Software Maintenance*, Sept. 2004.

[198] C. Wohlin, P. Runeson, M. Host, , M. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering*. Kluwer Academic Publishers, Norwell, MA, 2000.

[199] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16(1):4, 2007.

[200] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided Path Exploration in Dynamic Symbolic Execution. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 359–368, 2009.

[201] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 2010.

[202] Z. Xu, Y. Kim, K. M, and G. Rothermel. A hybrid directed test suite augmentation technique. In *Proceedings of the International Symposium on Software Reliability Engineering*, 2011.

[203] Yahoo! pipes. http://pipes.yahoo.com/pipes/.

[204] C. Yilmaz. Test case-aware combinatorial interaction testing. *IEEE Transactions on Software Engineering*, 39(5):684–706, May 2013.

[205] C. Yilmaz, S. Fouche, M. B. Cohen, A. Porter, G. Demiroz, and U. Koc. Moving forward with combinatorial interaction testing. *IEEE Computer*, 47(2):37–45, Feb. 2014.

[206] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification and Reliability*, 22(2), 2012.

[207] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[208] T. Yu, W. Srisa-an, and G. Rothermel. SimLatte: A framework to support testing for worst-case interrupt latencies in embedded software. In *Proceedings of the International Conference on Software Testing*, Apr. 2014.

[209] T. Yu, W. Srisa-an, and G. Rothermel. SimRT: An automated framework to support regression testing for data races. In *Proceedings of the International Conference on Software Engineering*, June 2014.

[210] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th European Conference on Computer Systems*, pages 321–334, 2010.