Automated web application testing: where it has been and where it is going

Timothy Goodrich
North Carolina State University
tdgoodri@ncsu.edu

ABSTRACT

As programs grow larger and larger, testing them becomes more difficult. To some point a human can manually test the software project, but eventually the project becomes too large and complex to test by hand. Therefore *automated testing* has become a very active area of research in recent years.

Unfortunately, not all software lends itself to automated testing, such as *web applications*. A web application dynamically generates content in response to the interacting user, creating problems for testing such as (a) not having a uniform input/output format across web applications, (b) not having a clearly defined set of possible interactions a user could have at a given web page, and (c) not having a discrete/recognizable number of good outcomes (e.g. rendered pages). These problems make web applications very difficult to test.

This survey centers around problems and solutions introduced in N. Alshahwan and M. Harman's "Automated web application testing using search based software engineering," looking at the past results that these authors utilize and the future work that has extended these results.

Keywords

Web applications; Automated test generation; Search-based software engineering

1. INTRODUCTION

"Test case generation is among the most labour-intensive tasks in software testing and also one that has a strong impact on the effectiveness and efficiency of software testing" note Anand et al. [4] in the introduction to their survey on automated test generation. In this context, dynamic content applications (e.g. web applications) are the worst of the worst for testing: a vast array of user inputs, content formats, and content outputs presents quite a complex problem. Worse, web applications are open to the whole internet of users, a single bug in thousands of lines of code could lead to a DoS attack or a SQL injection. Therefore developing automated tests for web applications is an important problem.

This report is centered around a paper by N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," where the authors present a method for generating tests for PHP-based web applications, the difficulties they faced, and the open problems left for future research. To examine the full context of this work we will examine four papers from the past that Alshahwan et al. cite, and three papers in the future that cite the work of Alshahwan et al. From the start, we note that four common themes will appear:

Theme 1. Automated test generation is computationally difficult and requires heuristics designed specifically for complex problems (e.g. search-based techniques).

Theme 2. Test generation heuristics require well-defined decisions (features) and objectives. For functional values (e.g. coverage threshold) this is not a problem, but non-functional values (such as general efficiency) present quite the challenge and are very application-specific.

Theme 3. Web applications contain many poorly-defined inputs, outputs, and objectives.

Theme 4. Web application data is rare, fairly homogenous, and few (if any) real benchmarks exist.

As we examine the core paper and the related works, note how these themes develop and are addressed by specific results.

2. THE CORE PAPER

2.1 Summary

Alshahwan et al. [2] introduce a handful of new algorithms (implemented in a tool named SWAT) for automatically testing web pages. They test this tool on six PHP-based web pages, yielding a 54% increasing in branch coverage and 30% reduction in practical run time complexity.

2.2 Keywords

- Search-Based Software Engineering (SBSE): Applying metaheuristic search techniques (gradient-ascending, simulated annealing, etc.) to perform typical software

ACM ISBN NA.
DOI: NA

engineering tasks (generating software tests, optimizing parameters, etc.).

- Automated web application testing: Automatically testing web applications involve both static and dynamic analysis, with an emphasis on the latter. Dynamic testing is computationally difficult given the large number of input options for a web page, making it an active area of current research.
- Static analysis: Static analysis for software testing is where the code is tested without being run (verification). Typical tests include data flow charts and proofreading code.
- Dynamic analysis: Dynamic analysis for software testing is where the code is tested by being run (validation). Typical tests include checking how user input is handled, and gauging run times and memory usage.

2.3 Identified Problems with Solutions

In their approach section, the authors spend a good deal of space describing problems with this sort of testing and their proposed solutions. For example:

Issue: Dynamic Typing
Description: Web development languages such as PHP, Python and Ruby are dynamically typed. All variables are initially treated as strings. If used in an arithmetic expression, they are treated as numeric at that operation. However, the same input can be treated as numeric in one expression and as a string in a different expression within the same script. This makes it hard to decide the type of variables involved in a predicate, posing a problem when deciding which fitness function to use.
Solution: To solve this problem, types of variables are checked dynamically at run-time using built-in PHP functions and then directed to the appropriate fitness function.

In total, the problems identified and fixed are:

- Interface determination: Interfaces are not uniformly defined between applications and languages. The authors' solution is to perform static analysis to identify the variables that are GET, POST, and REQUESTed, then providing these inputs as parameters to the tool. A similar version of this technique was used in [9].
- Dynamic typing. Variables in web languages are dynamically typed, meaning they keep variable values as strings until an operation requires another data type (e.g. casting into ints for addition). The authors fix this problem by dynamically checking the variable types with built-in PHP functions and using the appropriate fitness functions.
- User simulation: Typically a user is presented with top-level pages that he or she then navigates, and content is generated client-side and rendered to the user; identifying these top-level pages is non-trivial. The authors' solution is to treat every file not included in another file as a top-level page.
- Dynamic includes: PHP supports including files dynamically, where the filename is determined at run time. The authors' solution is to "approximate" these include variables by including every file that could possibly be a value for these variables. This technique has been previously used by [19].

2.4 Tools and Techniques

With these problems handled, the authors are able to establish a novel test generation algorithm and variants, implemented with [6, 7]. The full SWAT tool's architecture is visualized in Figure 2.4:

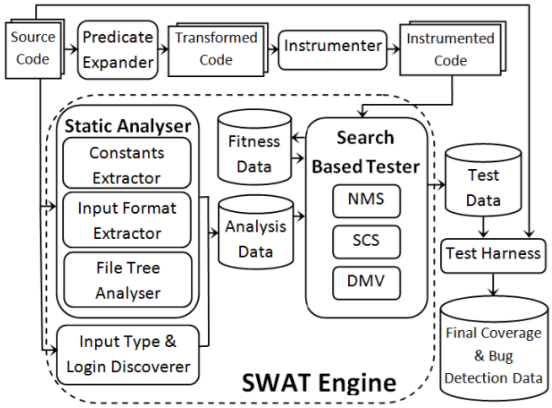


Figure 1: The SWAT architecture from [2].

Of note are the following details:

- Broad Test Generation Algorithm: Korel's hill climbing variant is used, which mutates a single decision at a time, and has a mechanism for preventing "over shooting" when close to optimal [13]. Furthermore, the code branches that are tracked are either (a) traversed immediately or (b) marked as "reached" and traversed later. This choice reduces computation by eliminating the need for "approach distances" [15], but is also not novel [17]. See Algorithm 1 in Appendix A for more details.
- Mutation Subroutine: If no improvement happened last time, then a new random input is generated. If the last decision that was modified decreased the overall fitness then a new decision is selected, otherwise the decision's "climbing speed" is accelerated. See Algorithm 2 in Appendix A for more details.
- Three novel seeding methods for the mutation subroutine are introduced and evaluated:
  1. Near Miss Seeding (NMS): If a decision was changed and it optimized the fitness of a branch other than the current one then it is considered a "near miss" and is recorded for later exploitation.
  2. Static Constant Seeding (SCS): SCS is where constants are used to initialize a new candidate instead of completely random values. This technique was introduced in [3] in the context of traditional applications and adapted to the web application format by the Alshahwan et al.
  3. Dynamically Mixed Value Seeding (DMV): When this hill climbing algorithm is run, a database

is maintained of constants associating certain input with certain output (e.g. selecting the "Help" button from the overhead menu will always lead to the help page, regardless of previous input). These associations are then used when mutating to provide optimal coverage: the algorithm knows which web pages have been covered and has a rough idea of what inputs led to this page, so new inputs are chosen to lead to new web pages. This DMV seeding is a particularly novel "kernel" to the SWAT tool and addresses both the problem of test generation coverage and the problem of broadly-defined web page inputs/outputs.

2.5 Data

This paper uses a dataset introduced earlier in Artzi et al [5], consisting of six open source PHP web applications:

THE WEB APPLICATIONS USED IN THE STUDY

App Name	Version	PHP Files	PHP ELoC	Description
FAQForge	1.3.2	19	834	FAQ management tool
Schoolmate	1.5.4	63	3,072	School admin system
Webchess	0.9.0	24	2,701	Online chess game
PHPSysInfo	2.5.3	73	9,533	System monitoring tool
Timeclock	1.0.3	62	14,980	Employee time tracker
PHPBB2	2.0.21	78	22,280	Customisable web forum

Figure 2: The PHP web application input corpus from [2].

Unfortunately, Artzi et al. provide little motivation for introducing this dataset other than noting that they are freely available on SourceForge [16]. Furthermore, the only real distinction in this dataset seems to be the split between small programs (FAQForge and Webchess with 19-24 PHP files) and medium-sized programs (Schoolmate, PHPSysInfo, Timeclock, and PHPBB2 with 62-78 files). So in some sense there is little motivation for using this dataset, other than because previous PHP-test generation algorithms have used this dataset.

2.6 Results

Visualized in Figure 2.6, the results fairly conclusively show that DMV seeding is by far the victor in terms of covered branches and in actual bugs (faults) found. A comparison to the previous work [5] is not given, potentially due to the latter not containing timing data (we expect an explicit-state model checker algorithm to perform better than a symbolic model checker because all states are explicitly checked, therefore solution quality alone is not a fair comparison).

2.7 Open Problems

While this work showed that dynamically tracking which inputs led to which outputs (the DMV seeding) was highly useful, some open problems still exist:

- Lack of tool automation. While the optimization process itself was automated, the tool itself was not and manually needed to be set up with the web application (a non-trivial process). Further work in unifying the input/output of web applications is needed for this automation to become a reality.
- Generalizing beyond PHP. While this tool worked well on the six PHP web application, PHP is a very specific language and these six web pages is a fairly small testing corpus. Generalizing both of these factors is an open problem.
- More intricate solutions to the preprocessing problems. The authors made several naive choices in handling the initial problems presented in preprocessing, such as importing all possible included files. Making more intricate choices here is desirable, both for run time purposes and automation purposes.

3. RELATED WORKS: THE PAST

3.1 Dynamic test input generation for web applications (2008)

3.1.1 Summary

Wassermann et al [20, 19] address differences between typical application (C or Java) testing and web application (PHP or Javascript) testing. Specifically, they propose using concolic (concrete + symbolic) testing for web applications, and test their methods on three PHP web sites (Mantis, Mambo, and Utopia News) in the context of detecting SQL injections. Little result statistics are given, but their approach generally seems to work.

3.1.2 Keywords

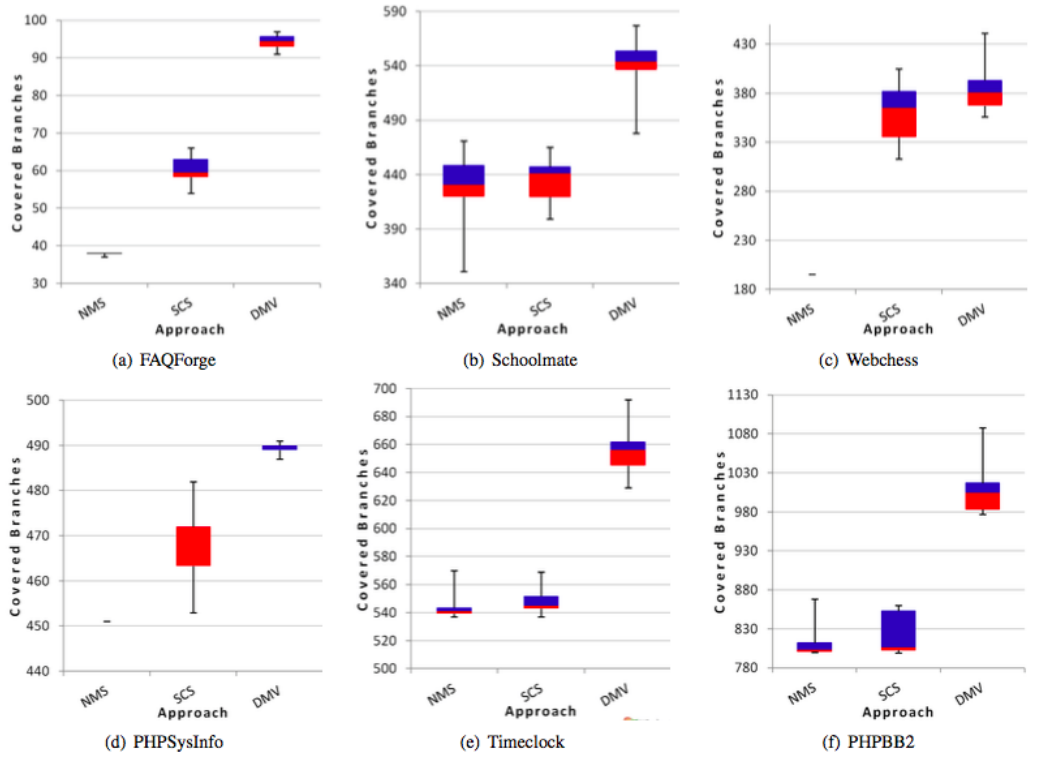
- Automated web application testing: Automatically testing web applications involve both static and dynamic analysis, with an emphasis on the latter. Dynamic testing is computationally difficult given the large number of input options for a web page, making it an active area of current research.
- Dynamic test generation: Dynamic testing gauges how well a piece of software responds to input, and generating sufficient tests is difficult (especially for web pages). Generating such tests is an active area of research.
- Test coverage: The coverage of a test measures how many branches (paths) a test covers in a piece of software. Different measures exist, such as lines covered, function calls covered, etc.
- Concolic testing framework: Concolic (concrete + symbolic) testing is a hybrid technique for performing both symbolic execution ("theoretical") and concrete execution ("practical") with real inputs.

3.1.3 Relevance to the Core Paper

Alshahwan et al. [2] cited this paper in the context of applying symbolic execution to source code (specifically web applications), but noted that the results were limited to function calls with SQL queries due to their interest in detecting SQL injection vulnerabilities.

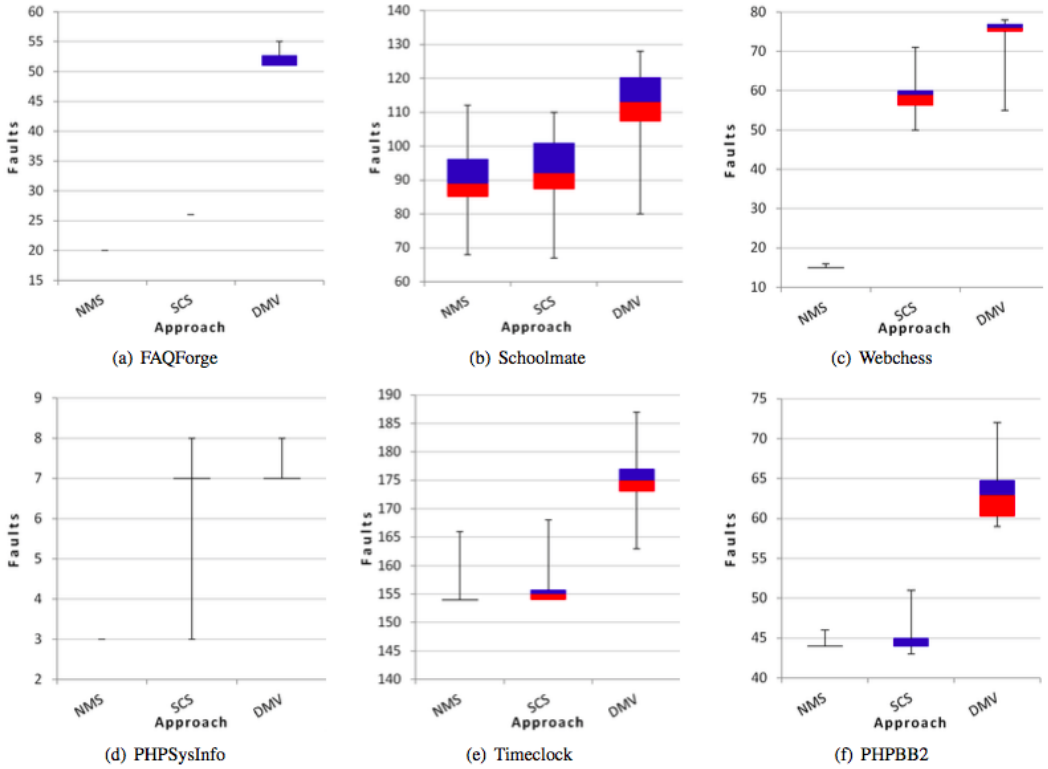
3.1.4 Broader Relevance

Ultimately, this paper did not contribute much to the web application testing community, seemingly because their results were very niche (SQL injection vulnerability detection)



(a) FAQForge (b) Schoolmate (c) Webchess

(d) PHPSysInfo (e) Timeclock (f) PHPBB2



(a) FAQForge (b) Schoolmate (c) Webchess

(d) PHPSysInfo (e) Timeclock (f) PHPBB2

Figure 3: Results (in terms of coverage and in faults found) of 30 runs of each of the 3 seeding methods for the 6 datasets from [2].

compared to the full testing that most researchers seem interested in. The three datasets used in this paper did not appear again in any subsequent papers I read or cited.

3.2 A systematic review of search-based testing for non-functional system properties (2009)

3.2.1 Summary

Afzal et al [1] provide a systematic survey paper of search-based testing techniques for non-functional properties. They identified Safety, Usability, Security, Quality of Service, and Execution Time as the non-functional properties examined in the literature, with the usual search-based techniques appearing (genetic algorithms, simulated annealing, hill climbing, etc.). The authors did not provide a recommendation on which technique to use for a given non-functional property, but rather provided the names of researchers who had applied x technique to y functional property, and noted this evolution over time.

3.2.2 Keywords

- Search-Based Software Engineering (SBSE): Applying metaheuristic search techniques (gradient-ascending, simulated annealing, etc.) to perform typical software engineering tasks (generating software tests, optimizing parameters, etc.).
- Test adequacy criterion: Test adequacy criteria is a set of requirements that evaluate whether a set of tests are sufficient or not. These criteria could include requirements about percent code coverage, which inputs are tested, etc.
- Non-functional properties/requirements: Non-functional properties/requirements indicate whether a piece of software is behaving correctly, but cannot specify particular behaviors. For example, properties such as speed, size, throughput, power consumption and bandwidth are non-functional, and the software requirements might specify ranges for each property; but the requirement cannot specify exact values.
- Fitness function: A fitness (objective) function summarizes a single attribute of a solution, particularly one used to gauge how "good" a solution is. For example, run time is a typical fitness function for algorithms.

3.2.3 Relevance to the Core Paper

Alshahwan et al. [2] cited this paper as an example of search-based techniques applying to both functional (natural) and non-functional (non-trivial) properties. However, none of the particular non-functional properties were used in web site testing, so the results were not directly applicable.

3.2.4 Broader Relevance

As a survey paper, this work is fantastic in overviewing which techniques were used by which groups of authors on specific properties. At the time this report was written, the paper had over 200 citations.

3.3 Precise interface identification to improve testing and analysis of web applications (2009)

3.3.1 Summary

Halfond et al [9] recognized that web applications have varied and ill-defined interfaces, making it a problem to create testing tools. They propose a form of symbolic execution used to abstract away these implementation details and provide a nice interface for a test generator. They test this symbolic executer on several applications, including test-input generation, penetration testing, and invocation verification.

3.3.2 Keywords

- Program testing: Performing an investigation to gauge the quality of a piece of software, especially when used under specific conditions.
- Automated web application testing: Automatically testing web applications involve both static and dynamic analysis, with an emphasis on the latter. Dynamic testing is computationally difficult given the large number of input options for a web page, making it an active area of current research.
- Symbolic execution: Symbolic execution is a method of checking which (set of) inputs trigger parts of a program to execute, and can be represented graphically as a path. Execution time is typically exponential.
- Penetration testing: Penetration testing tests if a piece of software (usually a web application) can be exploited, particularly with malicious user input.

3.3.3 Relevance to the Core Paper

Alshahwan et al [2] used a similar approach in handling interface determination by performing static analysis over the GET, POST, and REQUEST statements. The authors also note that Halfond et al's specific approach was limited to Java applications, whereas Alshahwan et al. applied the same idea to PHP applications. Despite the specifics, however, it seems that Halfond et al's approach was necessary for Alshahwan et al's results.

3.3.4 Broader Relevance

More broadly, Halfond et al do a good job of establishing a rigorous way of handling ill-defined input with an abstract layer, and all future work on this question builds off of their work. Of all the papers in this survey, this paper seemed the most universally respected and useful.

3.4 Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking (2010)

3.4.1 Summary

Artzi et al [5] solve basically the same problem as Alshahwan et al [2], but with explicit-state model checking instead of symbolic model checking. The same six PHP web applications are used, and the results are an upper bound for the much faster symbolic model checker, as expected. In some sense, this paper answers the "Can we test PHP web applications?", whereas Alshahwan et al's paper answers "How can we quickly test PHP web applications?"

3.4.2 Keywords

- Program testing: Performing an investigation to gauge the quality of a piece of software, especially when used under specific conditions.
- Program verification: Program verification checks that the program was written correctly, and is typically associated with static testing.
- Dynamic test generation: Dynamic testing gauges how well a piece of software responds to input, and generating sufficient tests is difficult (especially for web pages). Generating such tests is an active area of research.
- Explicit state model checking: Model checking involves testing whether a piece of software meets its specification. Explicit state model checking walks through every possible state explicitly, and typically has exponential run time.

3.4.3 Relevance to the Core Paper

As mentioned before, this paper seems to be the groundwork for the Alshahwan et al paper. The latter authors note that their work differs in that they are optimizing for *branch coverage*, whereas the former optimized for *statement coverage*. However, the motivation remained the same, and this work was an important first stab at the problem.

3.4.4 Broader Relevance

In a broader context, this paper is eclipsed by the Alshahwan et al paper. Explicit state model checking is the expensive, "dumb" way of testing in the classical computational complexity sense ("how do we solve the problem? Try every possible combination!"). Contrastingly, Alshahwan et al's result provides an *intuition* for what seems to be working (the DMV seeds track what is important and so reduce run time without reducing too much solution quality). Artzi et al's result is a stepping stone, then, and not a standalone result that has proven reusable (like Halfond et al's result).

4. RELATED WORKS: THE FUTURE

4.1 Software Engineering Meets Evolutionary Computation (2011)

4.1.1 Summary

This paper [11] is Harman's high-level survey of search-based techniques (specifically with evolutionary components) applied to software engineering problems. Harman emphasizes the exponential increase in the number of annual SBSE publications, and highlights the different software engineering problems that reduce to an optimization problem to be solved with evolutionary computing.

4.1.2 Keywords

- Evolutionary computation: Evolutionary computation is a class of algorithms that adopt Darwinian principles (e.g. evolution) to generate waves of candidate solutions. Examples include genetic programming, ant-colony optimization, etc.
- Realistic algorithm: An informal definition, a realistic algorithm can be expected to run in a reasonable amount of time on a reasonable amount of input; exact orders of magnitude will be domain-specific. For example, in theoretical computer science a polynomial-time algorithm is fast, whereas in engineering applications an algorithm needs linear (or log-linear) run time to be useful.
- Search-Based Software Engineering (SBSE): Applying metaheuristic search techniques (gradient-ascending, simulated annealing, etc.) to perform typical software engineering tasks (generating software tests, optimizing parameters, etc.).
- Software design: Software design is the process of formalizing a set of specifications, requirements, and constraints that a proposed piece of software will satisfy. These details can be quantitative or qualitative, and typically evolve as the software matures.

4.1.3 Relevance to the Core Paper

In some sense this paper is the least interesting because it is so high-level. But taking this step back emphasizes one fact: web application testing can and is done with evolutionary computing. The Alshahwan et al paper has "made it," so to say.

4.1.4 Broader Relevance

This survey paper does a good job at offering an updated picture of the field portrayed by Afzal et al. Several repeated citations also suggests that there's a strong correlation between "interesting software engineering problem" and "non-functional properties," an interesting side-note.

4.2 The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs (2012)

4.2.1 Summary

The GISMOE challenge is a framework proposed by Harman et al [12] for offering the software developer a choice of coding trade-offs before any code is written. More formally, the software engineering project's decisions and objectives are encoded into an optimization problem, and a Pareto frontier (surface) is generated. By picking any candidate on this surface, the software developer can choose between several tangible trade-offs. This framework's benefits are two-fold: (a) the optimization algorithm does not need to solve the trade-offs itself, so generating a large frontier is almost preferred, and (b) the software developer is not simply presented with a single result, but is offered an entire surface of options. Ultimately, Harman et al hope to fundamentally change how SBSE interacts with humans, resulting in more efficient algorithms and more useful results.

4.2.2 Keywords

- Non-functional properties/requirements: Non-functional properties/requirements indicate whether a piece of software is behaving correctly, but cannot specify particular behaviors. For example, properties such as

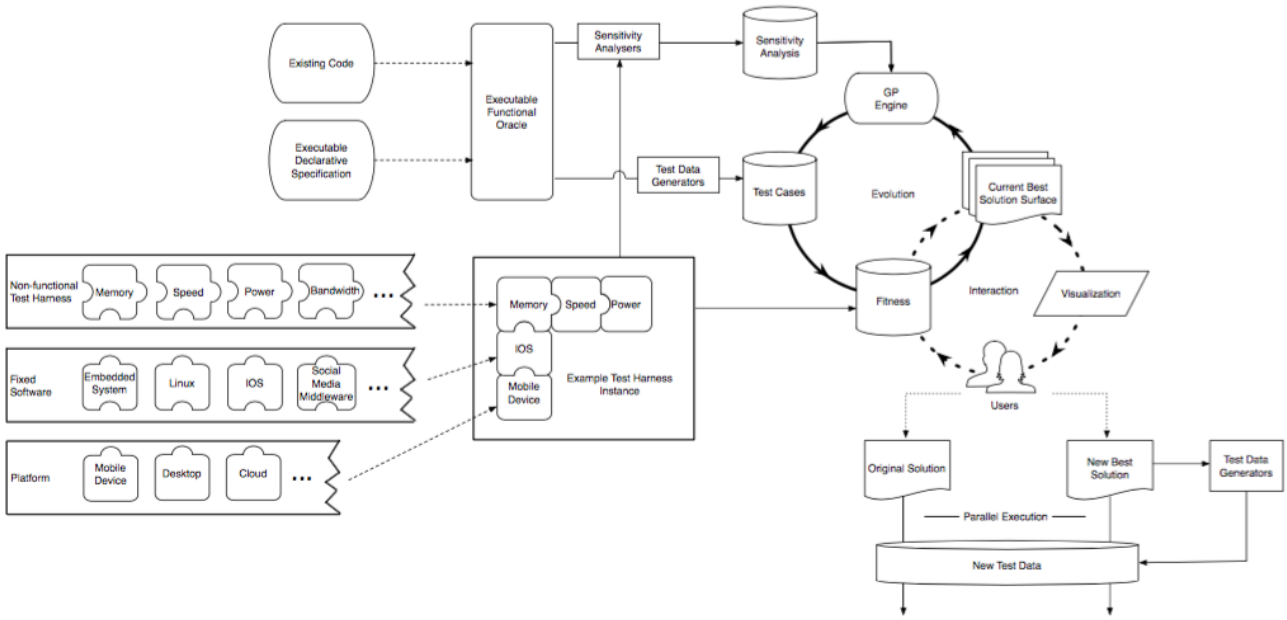


Figure 4: The proposed GISMOE framework from [12].

speed, size, throughput, power consumption and bandwidth are non-functional, and the software requirements might specify ranges for each property; but the requirement cannot specify exact values.

- Functional properties/requirements: Functional properties/requirements specify exactly what output the software should have.
- Search-Based Software Engineering (SBSE): Applying metaheuristic search techniques (gradient-ascending, simulated annealing, etc.) to perform typical software engineering tasks (generating software tests, optimizing parameters, etc.).
- Pareto surface/frontier: In multi-objective optimization, the Pareto surface/frontier is the set of all candidate solutions which are not dominated.

4.2.3 Relevance to the Core Paper

The GISMOE paper does not mention web application testing explicitly, but rather groups the Alshahwan et al paper in with "recent SBSE advances" in general. Again, we see that Harman in particular sees the Alshahwan et al result as strong enough proof that SBSE provides an advantage when generating tests for web applications.

4.2.4 Broader Relevance

GISMOE diverges from the typical description of SBSE in the sense that it does not propose to be an algorithm that takes in a software engineering problem and outputs a solution. Instead, GISMOE could/will be a tool for understanding a problem's fundamental trade-offs and choosing (manually) which objectives it makes sense to optimize/prioritize. This claim can be taken one of two ways. First, GISMOE is too high-level to be useful, i.e. a pseudo-science with no real fundamental results. On the other hand, it could take the existing SBSE field, with all the results cited in this survey, and change around the picture to involve human interaction in a new manner. The GISMOE project might be an interesting work to keep track of over the next several years.

4.3 An orchestrated survey of methodologies for automated software test case generation (2013)

4.3.1 Summary

Anand et al [4] present a survey paper on automated software testing, using the following techniques: (a) structural testing using symbolic execution, (b) model-based testing, (c) combinatorial testing, (d) random testing and its variant of adaptive random testing, and (e) search-based testing. The search-based testing section is written by Mark Harman, Phil McMinn, John Clark and Edmund Burke, and cites dozens of applications (including web application testing).

4.3.2 Keywords

- Dynamic test generation: Dynamic testing gauges how well a piece of software responds to input, and generating sufficient tests is difficult (especially for web pages). Generating such tests is an active area of research.
- Search-Based Software Engineering (SBSE): Applying metaheuristic search techniques (gradient-ascending, simulated annealing, etc.) to perform typical software engineering tasks (generating software tests, optimizing parameters, etc.).

- Non-functional properties/requirements: Non-functional properties/requirements indicate whether a piece of software is behaving correctly, but cannot specify particular behaviors. For example, properties such as speed, size, throughput, power consumption and bandwidth are non-functional, and the software requirements might specify ranges for each property; but the requirement cannot specify exact values.

- Functional properties/requirements: Functional properties/requirements specify exactly what output the software should have.

4.3.3 Relevance to the Core Paper

As seems to be the theme with the future papers, no new results are made in automated web testing; instead, the current results are cited and "web application testing" is cited as an area where SBSE has suceeded.

4.3.4 Broader Relevance

Whereas the Afzal et al survey covered search-based techniques alone, this survey includes search-based techniques as only one of five different options. This survey also emphasizes results and provides recommendations based on effectiveness, whereas the Afzal et al paper concentrated on the trends in metaheuristic usage over time and the clusters of researchers working with these metaheuristics. We see a shift, then: before, the emphasis was on the techniques used and how well they tested different properties. Now we see the emphasis on the question "What can you *do* with these metaheuristics?", pushing the limit on what the field as a whole can do. Of note: web application testing appears as an application with a solid solution.

5. COMMENTARY

In this report, we have done the following:

- Zoomed in on a particular result of search-based techniques applied to generated automated tests for web applications.

- Noted that while some previous results came close to solving the same problem, no future results have (so far) looked at the same problems.

- Also noted that more recent works have happily called "web application testing" a tried-and-true area of results.

In some sense it seems like this area is stagnant. The existing result is fairly substantial, and the more recent works have no emphasizes any open problems in the area. However, if we want to find some open problems, we need not look too far. Instead, we merely need to add more theory or more applications:

Theory 1. The DMV seeding found correlations between conditional branches and input values. Is there a smarter way to keep track of these values? For example, differential evolution [18] "accidentally" keeps track of best values by not throwing their candidates out. At a high level we can imagine that differential evolution makes a natural fit: we can crossover the common features to candidates so that more branches are explored. Perhaps there are some results to come from looking at DEs.

Theory 2. Tabu search [8] is a general technique for keeping track of previously seen values and ignoring them in a future search, and yet this technique was not mentioned in the Alshahwan et al paper. Why not? This seems like a natural mechanism for seeding the mutations.

Application 1. The PHP datasets used in the papers here were small and not very regular. A six PHP web application dataset was used in two papers, and a three PHP web application dataset was used in another paper with an emphasis on SQL injections. Why stick with PHP if the datasets are so rare and small? One alternative might be to look at Android application repositories [14, 10]. Android applications offer the same problems as web applications (dynamically generated content, a wide array of user inputs), and dozens of good open source Android datasets exist. Perhaps by extending the current results to these new datasets, new features (good and bad) will be revealed.

6. CONCLUSION

In conclusion, we have seen how a result begins from a collection of techniques applied to a vague problem (web applications), become concrete (search-based techniques applied to PHP applications), and then become vague in the techniques and specific in the problem (evolutionary computing as a whole applied to specific web applications). In some sense we have seen the lifespan of this result: it was born as a wide array of techniques were applied to a specific problem, and now is written off as an area where SBSE has found results. Curiously, this might be why we have not seen new results in the area of web application testing – the problem is written off as solved and uninteresting now. Perhaps new datasets will liven up the work, perhaps not. Perhaps new algorithms will lead to more intriguing problems, perhaps not. Either way, we see how research iterates between techniques and problems as the driving force, and how many attempts must be made before a good solution is found.

7. REFERENCES

[1] Afzal, W., Torkar, R., and Feldt, R. A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol. 51*, 6 (June 2009), 957–976.

[2] Alshahwan, N., and Harman, M. Automated web application testing using search based software engineering. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on* (Nov 2011), pp. 3–12.

[3] Alshraideh, M., and Bottaci, L. Search-based software test data generation for string data using program-specific search operators: Research articles. *Softw. Test. Verif. Reliab. 16*, 3 (Sept. 2006), 175–203.

[4] Anand, S., Burke, E., Chen, T., Clark, J., Cohen, M., Grieskamp, W., Harman, M., Harrold, M., and Mcminn, P. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw. 86*, 8 (Aug. 2013), 1978–2001.

[5] Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., and Ernst, M. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *Software Engineering, IEEE Transactions on 36*, 4 (July 2010), 474–494.

[6] Bouwers, E., and Bravenboer, M. Php-front: Static analysis for php.

[7] Bravenboer, M., Kalleberg, K., Vermaas, R., and Visser, E. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program. 72*, 1-2 (June 2008), 52–70.

[8] Glover, F., and Laguna, M. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[9] Halfond, W., Anand, S., and Orso, A. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (New York, NY, USA, 2009), ISSTA '09, ACM, pp. 285–296.

[10] Hamasaki, K., Kula, R., Yoshida, N., Cruz, A., Fujiwara, K., and Iida, H. Who does what during a code review? datasets of oss peer review repositories. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (Piscataway, NJ, USA, 2013), MSR '13, IEEE Press, pp. 49–52.

[11] Harman, M. Software engineering meets evolutionary computation. *Computer 44*, 10 (Oct. 2011), 31–39.

[12] Harman, M., Langdon, W., Jia, Y., White, D., Arcuri, A., and Clark, J. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2012), ASE 2012, ACM, pp. 1–14.

[13] Korel, B. Automated software test data generation. *IEEE Trans. Softw. Eng. 16*, 8 (Aug. 1990), 870–879.

[14] Krutz, D., Mirakhorli, M., Malachowsky, S., Ruiz, A., Peterson, J., Filipski, A., and Smith, J. A dataset of open-source android applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (Piscataway, NJ, USA, 2015), MSR '15, IEEE Press, pp. 522–525.

[15] McMinn, P. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab. 14*, 2 (June 2004), 105–156.

[16] Media, S. Sourceforge.

[17] Michael, C., McGraw, G., and Schatz, M. Generating software test data by evolution. *IEEE Trans. Softw. Eng. 27*, 12 (Dec. 2001), 1085–1110.

[18] Storn, R., and Price, K. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization 11*, 4 (Dec. 1997), 341–359.

[19] Wassermann, G., and Su, Z. Sound and precise analysis of web applications for injection vulnerabilities. *SIGPLAN Not. 42*, 6 (June 2007), 32–41.

[20] Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., and Su, Z. Dynamic test input generation for web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2008), ISSTA '08, ACM, pp. 249–260.

APPENDIX

A. PSEUDOCODE

Require: Application Name *AppName*
Require: Static Analysis results *AnalysisDB*
$\mathcal{U}$: queue of top level file units to be processed. Retrieved from the File Tree Analyser results.
$\mathcal{B}$: queue of branches reached but not covered.
$\mathcal{C}$: Coverage table of all branches with the best achieved distance.
$\mathcal{T}$: Test cases that achieved best distance for each reached or covered branch.
$\mathcal{F}$: set of branch and fitness values achieved for the executed test case.
Input : setOf(inputname, value)
IV: Input Vector consisting of setOf(*Input*)
Distance: holds fitness value for a certain branch.

```
1: U := getTestUnits(AppName, AnalysisDB)
2: T := φ
3: for all U in U do
4:    IV:= φ
5:    F :=executeTestcase(U, IV)
6:    T :=updateTestdata(T,U,IV,F)
7:    C :=updateCoveragedata(C, F)
8:    while first run or coverage improved do
9:       B := getReachedBranches(C)
10:      for all B in B do
11:         initState()
12:         IV := setInputVector(B, AnalysisDB,T)
13:         Input := NULL
14:         CurrentDistance := getBranchDist(B, C)
15:         while CurrentDistance > 0 and not no improve-
            ments for 200 tries do
16:            initilaizeDB()
17:            Input:= mutateInputs(IV,Input,
               CurrentDistance, PreviousDistance)
18:            IV:= replaceInputValue(IV,Input)
19:            F :=executeTestcase(U, IV)
20:            T :=updateTestdata(T,IV,F)
21:            C :=updateCoveragedata(C, F)
22:            PreviousDistance = CurrentDistance
23:            CurrentDistance = getBranchDist(B, C)
24:         end while
25:      end for
26:   end while
27: end for
28: return T
```

Figure 5: Algorithm 1 from [2].

Require: *IV*, *Input*, *CurrentDistance*, *PreviousDistance*, *AnalysisDB*

```
1: if Input is NULL or CurrentDistance = PreviousDistance
   then
2:    Input := selectNewInput(Input,IV )
3: else
4:    if CurrentDistance > PreviousDistance then
5:       changeMutationOperator()
6:    else
7:       if CurrentDistance < PreviousDistance then
8:          accelerateOperation()
9:       end if
10:   end if
11: end if
12: Input:= mutate(Input,AnalysisDB)
13: return Input
```

Figure 6: Algorithm 2 from [2].