An Industrial Case Study of the Effectiveness of Test Generators

Pietro Braione, Giovanni Denaro
University of Milano-Bicocca
Milano, Italy
{*braione*|*denaro*}*@disco.unimib.it*

Andrea Mattavelli, Mattia Vivanti
University of Lugano
Lugano, Switzerland
{*andrea.mattavelli*|*mattia.vivanti*}*@usi.ch*

Ali Muhammad
VTT Technical Research Center of Finland
Tampere, Finland
ali.muhammad@vtt.fi

***Abstract*—Automatic test generators pursue some type of systematic coverage of the program code or heuristic sampling of the program inputs. Test generators are effective after the assumption, often (enthusiastically) embraced by researchers, that the generated test cases produce informative data for domain experts, e.g., pinpoint important bugs. This paper investigates the validity of such assumption through a case study of using test generators on industrial software with nontrivial domain-specific peculiarities. Our results properly enhance the available body of knowledge on the strengths and weaknesses of test generators.**

I. Introduction

Many modern techniques and tools for software testing embody capabilities to automatically generate test cases. This paper uses the terminology *test generators* to refer to these techniques and tools. Test generators promise significant impact on the testing practice, since they promote extensively tested software within reasonable effort and cost bounds.

Test generators pursue some form of systematic exploration of the input space of the program under test. We classify test generators as random, path-crawling or coverage-driven. *Random test generators* model the input space by a set of random variables and generate test cases by extracting random samples accordingly [1], [2], [3], [4]. *Path-crawling test generators* formulate the test generation problem as the one of satisfying the executability conditions of the possible program paths, typically derived through some type of symbolic analysis of the program [5], [6], [7], [8], [9], [10]. *Coverage-driven test generators* explicitly steer the test generation process towards yet uncovered elements, according to adequacy criteria such as statement or branch coverage [11], [12], [13], [14], [15], [16], [17], [18]. We survey these technologies in more detail in Section II.

This paper studies the effectiveness of test generators on industrial software. It reports on a case study that confronts a (combination of a) set of state-of-the-art test generators with the testing of a family of programs with nontrivial domain-specific peculiarities. The subject of study is a software component of a real-time and safety critical control system that remotely operates maintenance robots within a nuclear fusion power plant. The component is programmed in C, embeds several floating point non-linear arithmetics and integrates within a time dependent control task specified in LabView, a graphical language to design embedded systems. The study considers four incremental versions of this component. Our study addresses the overall research question on whether test cases yielded by test generators are effective, meaning that they can produce informative data for domain experts, such as, pinpointing unknown bugs or undesirable program behaviors, at complement of manual testing.

Whilst many papers argue that test generators can achieve effective results, we see weaknesses of the previous studies that call for empirical data as contributed by this paper.

The amount of papers that report on industrial benchmarks is limited. Many papers consider codes from student work or programming libraries [1], [4], [16]. Even when that is not the case, the subject programs mostly handle integer-typed variables, thus simplifying the job of the test generators. Our study challenges test generators to deal with industrial programs that exploit floating point arithmetics to large extent. It enlightens that the scarce support of floating point arithmetics in current (non random) test generators is a major hinder for the transfer of this promising technology to industry, and should be thus addressed as a top priority by the researchers in the field.

Many popular test generators are presented as tools to unveal program crashes or violations of specification contracts implemented as assertions in the code [1], [2], [4], [7], [9], [5], [10]. While these are interesting types of failures, it is not clear whether they should be the prevalent (or the only) target for test generators. In facts, the failures found in our study never crashed the subject programs, including low level divisions by zero that were silently propagated as NaN values.

Several experiments only assess the relative effectiveness of test generators, based on code coverage indicators [3], [9], [14], [19], [16], [17], [20]. These data give a weak feedback on the effectiveness of the approach. We use code coverage as the means to steer the test generation process, but assess effectiveness as the ability of producing informative data for domain experts. We contribute novel empirical evidence that test generators can expose both unknown (and subtle) bugs in the considered industrial programs. Our results confirm the high potential of test generators to promote significant boost of effectiveness of software testing in practice.

II. Approaches to Test Generation

This section surveys the main approaches to automatic test generation.

Random testing: Random testing consists of randomly sampling the input space of the program, and running the program against the obtained random inputs [1], [2], [3]. Random testing has the distinctive advantage of possibly being the least expensive strategy as it quickly finds large amounts of test cases, without any analysis of the program structures, nor the source code. On the negative side, it is substantially unable to elicit program behaviors associated with small or singular subsets of the inputs. Adaptive random testing attempts to improve the effectiveness of random testing by controlling how the test cases are distributed over the input domain [4].

Path-crawling testing: Path-crawling testing looks for inputs that exercise the program paths [7], [8], [9]. Path-crawling testing is structural, meaning that it requires access to the program code. It will elicit behaviors that entail distinct flows of instructions, which cannot be easily spot by randomly sampling the input space.

Path-crawling test generators usually embody some variant of symbolic execution, a program analysis technique that simulates the execution of programs over symbolic data. Symbolic execution computes the *path conditions*, that is, the executability conditions of the simulated program paths. The test generators then rely on external decision procedures (solvers) to solve the path conditions and obtain test cases. The final precision of the techniques, that is, the paths for which exercising inputs can be indeed found, depends on the theories that can be successfully handled by the the underlying solver.

An efficient variant of the technique is *concolic testing* [5], [6], [19], [21], [22]. Concolic testing merges random and path-crawling testing based on symbolic execution, aimed at improving over both. Starting from an initial test suite, possibly obtained by random testing or designed by test analysts, concolic testing performs symbolic execution along the control flow paths exercised by the available test cases. Throughout the resulting path conditions, it then negates clauses at selected points to obtain the path conditions of not-yet-covered paths, and solves them to generate new test cases. These are fed back to the next iterations of the procedure. A distinctive characteristic of concolic testing is to heuristically exploit data from the concrete executions to approximate path conditions within the theory handled by the underlying solver.

Coverage-driven testing: In this paper, we use the terminology *coverage-driven testing* to indicate those approaches that augment the test generation process with the explicit knowledge of which coverage targets have not been executed yet, according to an adequacy criterion of interest, e.g., statement or branch coverage. These approaches monitor the program code for the coverage targets covered by the generated test cases, and steer the subsequent test generation activity towards yet uncovered targets.

Some approaches leverage optimization algorithms, using goal and fitness functions derived from symbolic path conditions and information on the uncovered code elements [11], [12], [13]. Other approaches extend path-crawling testing with coverage-awareness, according to heuristics that prioritize paths with higher chances of increasing coverage [14], [15], [16]. Yet other proposals exploit the incremental analysis of the (in)feasibility conditions of the coverage targets within the test generation procedure [17], [18].

III. Subject of the Study

ITER is part of a series of experimental reactors which are meant to investigate the feasibility of using nuclear fusion as a practical source of energy and demonstrate the maintainability of such plants [23], [24]. Due to very specialized requirements, the maintenance operations of ITER reactor demand the development and testing of several new technologies related to software, mechanics, electric and control engineering. Many of these technologies are under investigation at Divertor Test Platform (DTP2) at VTT Technical Research Centre of Finland [25]. DTP2 embeds a real-time and safety critical control system for remotely operated tools and manipulation devices to handle the reactor components for maintenance [26]. The control system is implemented using C, LabVIEW and IEC 61131 programming languages.

The software component chosen for this study is part of the motion trajectory control system of the manipulation devices. The software is implemented in C. It provides an interface between the operator and the manipulator. The operator inputs the target position of the manipulator, along with the maximum velocity, initial velocity, maximum acceleration and maximum deceleration, as physical constraints on the generated trajectory. As a result, the software plans the movement of the manipulator, interpolating a trajectory between two given point in n-dimensional space, where n is the number of physical joints in manipulator. It returns outputs in the form of smooth motions, so that the manipulator's joints accelerate, move and decelerate within the physical bounds until the target position. This avoids the mechanical stress on the structure of the manipulator to ensure its integrity and safety. It also keeps the desired output forces of the joints' actuators in check. The correctness of such software plays a key role in the reliability of the control system of the ITER maintenance equipment.

The software aims to produce the trajectories in such a way that all the joints start and finish the motion at the same time. This constraint is fulfilled by slowing down the motion of certain joints and it is ensured that the acceleration and velocity constraints are not violated for any of the joints. The software ensures that all joints finish the motion at the same time by slowing down acceleration and velocities for

certain joints. The component is designed to be compiled as Dynamic Link Library (DLL) to work with Matlab or LabVIEW.

The study considers four incremental versions of the subject software. Code size ranges between 250 and 1,000 lines of code. The number of branches ranges between 36 and 74. All versions include 6 functions with maximum cyclomatic complexity equal to 11.

Baseline version: The baseline version is the main working implementation of the software, which can be compiled to run in LabVIEW real-time environment. This version was used to test the motion characteristics of a water hydraulic manipulator.

Platform change (buggy) version: The second version considered in the study is fundamentally a platform change porting of the baseline version. This version provides almost the same functionality, but is designed to compile as a DLL to work in the Matlab Simulink environment. It was implemented to simulate and plan the motions in the virtual environment before executing it on real manipulator, aiming to enhance the safety of operations.

Platform change (buggy) version: The third version considered in the study is a bug fix of the second one. In facts, the above Matlab version contains a particular bug causing manipulator to violate the maximum velocity and acceleration limits. This bug remained in the software for several years before it was detected and fixed in this version.

New implementation: The fourth version considered in the study is a new, recently proposed, implementation to obtain the same functionality, but to rectify unwanted behaviors in the previous implementations. The component has not been tested in the real environment yet, and thus it is not yet known if this new implementation entirely provides the proper functionality.

IV. Methodology

A. The Test Generator

We have instantiated automatic test generation based on the tools CREST ([14]) and ARC-B ([27]). CREST supports random testing, concolic testing and coverage-driven testing, while ARC-B specifically addresses coverage-driven testing. We have combined these tools in a four-stage test generator (concretely engineered as a shell script that calls the tools sequentially) that pursues the maximization of branch coverage. The test generator works as follows.

In the first stage, the test generator runs CREST in concolic testing mode, configured for depth-first traversal of the program paths. Offline we have verified that, for the programs considered in our study, none of the other path-crawling heuristics supported by CREST performed better than depth-first traversal. Throughout the concolic procedure, the test generator monitors the generated test cases against the program under test and retains only the test cases that result to increasing branch coverage over the previous ones. The process is continued up to saturation, defined as experiencing no coverage increase for a configurable (set to 10,000) budget of iterations. This definition of saturation applies to the next stages of the test generator too.

In the second stage, the test generator runs CREST in random testing mode until saturation, and again it retains only the test cases that result to increase of coverage.

In the third and fourth stage, the test generator pursues additional coverage in the way of coverage-driven testing using CREST and ARC-B, respectively. The coverage-driven testing mode of CREST extends concolic testing with a particular heuristics that weights the selectable paths according to the distance from the not-yet-covered branches, and takes into account the number of unsuccessful attempts to follow specific subpaths to not-yet-covered branches in previous iterations [14]. ARC-B addresses coverage-driven testing by incrementally computing (and then solving) the executability conditions of the not-yet-covered branches [27]. Before either stage, the test generator sets the coverage targets according to the branches that were not covered in the preceding stages, and then runs the tools until saturation, retaining the test cases that increase coverage.

Our test generator applies four specific test generation strategies in a specific order. Being primarily interested in assessing the effectiveness of automatically generable test cases, we did not look into applying these strategies in isolation or trying alternative orderings of the strategies, nor we have used all possible strategies (e.g., our test generator does not encompass search-based approaches). Any claim about the relative strengths of the test generation strategies is therefore out of the scope of this paper.

We further adapted our test generator to work in *test suite augmentation mode* [28]. In this mode, the test generator inputs an already available test suite, and runs it against the the program under test before the first stage. Throughout the test generation process, it then retains only the test cases that generate additional coverage over the input test suite. We used the test augmentation mode when generating test cases for the non-baseline versions of the subject software.

B. Dealing with Floating Point Arithmetics

The industrial software considered in our study exploits floating point arithmetics in most computations. Conversely, all above CREST's and ARC-B's test generation modes (but random testing) handle effectively only computations over integers, the reason being the limits of the underlying solvers to reason over the theory of reals with discrete floating point representation. We tackled this problem by reshaping the subject programs on top of a programming library that simulates the floating point semantics over integer-typed variables.[1] This yielded indeed an analyzable program, but costed a factor-10 increase in the number of branches in the analyzed programs.

[1] http://www.jhauser.us/arithmetic/SoftFloat.html

Worth mentioning, we have also tried the approach of approximating the floating point computations with fixed point counterparts. This had the advantage of only slightly increasing the dimension of the code. The resulting loss of precision affected the correctness of the analysis to large extent, yielding several spurious executions and crashes of the programs. We discarded this latter approach on this basis.

For all generated test suites, we finally re-computed the coverage indicators by executing the test cases against the original subject programs (with *Gcov*[2]). Our goal was to assess the absence of spurious executions caused by the inconsistencies between the original implementation and the floating point simulation library. It also served to polish the coverage data from the figures related to the library. We did not find any spurious execution. This confirms that the floating point simulation library correctly simulates the floating point computations of our subject programs.

C. Test Oracles

To assess the effectiveness of these tests, we could not rely on automatic test oracles, such as code assertions in the style of design-by-contract, since the subject programs contained no such assertions. No test case resulted to runtime exceptions or program crashes either. Even though (as we further detail below) there were happening cases of underflows and divisions by zero for some floating point computations, the standard semantics of floating point operations handles these exceptional cases by returning special values, such as, NaN (not a number) or Inf (infinity). These values were silently propagated by the subject programs.

We inspected the test outcomes by looking into the trajectories of the manipulator's joints generated by the subject programs. The subject programs yield the trajectory data of the joints as 6-tuples of floating point values. Each 6-tuple represents the trajectory of a joint by the times (three values) up to which the joint has to accelerate, cruise at peak velocity and decelerate, respectively, and the corresponding (other three values) acceleration, peak velocity and deceleration in each phase. For each test case, we collected and analyzed the trajectory data in two forms: the values of the yielded 6-tuples and the plots of the resulting movements of the joints and their velocities over time. We searched the 6-tuples for (unexpected) 0, NaN or Inf values, and the plots for unexpected or inconsistent shapes across the subject programs. We were supported by VTT experts for the analysis of the plots.

All test suites and problem reports from our testing activity have been submitted to developers of VTT to collect the feedback of domain experts on the relevance of the generated test cases and the correctness of our observations.

[2]Gcov is part of the GNU Compiler Collection.

Table I
TEST CASES (AND COVERAGE) PER TEST GENERATOR'S STAGES

		Baseline version	Upgrades over baseline version		
			Platform change (buggy)	Platform change (fixed)	New implementation
	#Branches	96	152	116	58
#Test cases from	previous version test suite	n.a.	20	32	20
	(Branch coverage)		(70%)	(82%)	(81%)
	concolic testing (stage 1)	9	9	-	2
	(Branch coverage)	(68%)	(86%)	-	(83%)
	random testing (stage 2)	8	3	-	1
	(Branch coverage)	(77%)	(86%)	-	(83%)
	coverage-driven CREST (stage 3)	1	-	-	-
	(Branch coverage)	(80%)	-	-	-
	coverage-driven ARC-B (stage 4)	2	-	-	-
	(Branch coverage)	(84%)	-	-	-
	TOTAL	20	32	32	23
	(Branch coverage)	(84%)	(86%)	(82%)	(83%)

V. RESULTS

We have run the four-stage test generator described in previous section against the subject programs. Table I summarizes the number of test cases generated for each program through the stages of the test generator, and the coverage after each stage. Recall that each stage ran incrementally over the test cases generated at the preceding stage, retaining only the test cases that increased the branch coverage.

Below we describe the problematic behaviors revealed by the automatically generated test suites, grouped by subject program. Other than revealing the known bug, these include relevant and previously unknown problems.

Baseline version: For the *baseline version*, that is, the reference LabVIEW version of the component under test, the test generator produced 20 test cases that cover 84% of the branches. The concolic and random stages generated most test cases, 9 and 8, respectively, while the coverage-driven stages contributed 3 test cases only.

Table II reports trajectory data (columns *Output*) yielded by the baseline program for the inputs (columns *Input*) of some test cases (column *Test#*). The test cases are referred by their position in test suite according to the order in which they are yielded by the test generator. The inputs include maximum and initial velocity, maximum acceleration and maximum deceleration of the joints. Origin and destination positions are omitted for space reasons. The outputs are the 6-tuples of trajectory data.

Table II
TRAJECTORY DATA COMPUTED BY THE BASELINE PROGRAM FOR SOME AUTOMATICALLY GENERATED TEST CASES

	Input				Output					
					Accelerate		Keep peak velocity		Decelerate	
Test#	Max velocity	Initial velocity	Max acceleration	Max decelaration	for (s)	at (m/s^2)	for (s)	at (m/s)	for (s)	at (m/s^2)
6	1.4e-45	0.0	5.0	1.4e-45	0.0	5.0	inf	1.4e-45	0.0	1.4e-45
19	5.0	0.0	0.0	0.0	0.0	-0.0001	inf	-0.0	0.0	-0.0001
20	5.0	0.0	2.0	-1.0	0.0	-2.0	inf	-0.0	-0.0	1.0

Table II shows that the program fails to handle very small input values (test case 6), and combinations of the input parameters that include all zero (test case 19) or some negative (test case 20) values of the maximum acceleration/deceleration of the joints. The failures display as unexpected 0 and Inf values in the outputs. Debugging revealed that the failure of test case 6 is due to floating point underflows in a multiplication that involves the small values, while the failures of test cases 19 and 20 derive from divisions by zero, in turn caused by a program's function that returns 0 for undealt inputs. We collected from VTT experts the feedback that, although these inputs are hardly showing up (e.g., the program is currently never used with negative inputs), such (unknown) problems call for strengthening the robustness checks in the program to avoid future issues.

Platform change (buggy) version: For the version *platform change (buggy)*, that is, the incremental modification of the baseline version to migrate from LabVIEW to Matlab, the test generator augmented the test suite of the baseline version with 12 additional test cases, resulting to coverage of 86% of the branches. The concolic and random stage produced 9 and 3 additional test cases, respectively. The coverage-driven stages were not able to improve the coverage any further. The 3 test cases from the random stage covered 3 additional branches of the floating point simulation library used to facilitate the test generator, but did not result in additional coverage of the original code.

Replicating the test suite from the baseline program against the two platform change versions did not expose any problem. Conversely, the 9 additional test cases generated by the concolic stage of the test generator pinpointed the known bug. Figure 1 shows the plot of the movement of joint 2 when the test case 25 is executed on the baseline program and the buggy platform change version, respectively. The latter version clearly accelerates more than the former one.

At the code level the fault consists of a sequence of assignments that may double or triplicate the value of maximum velocity in the presence of quiet joints (same origin and destination positions in the input). The equality constraints to execute these assignments are the typical case in which concolic testing overcomes random testing: the equality constraints are easy to solve from the symbolic path conditions, while the probability of randomly generating

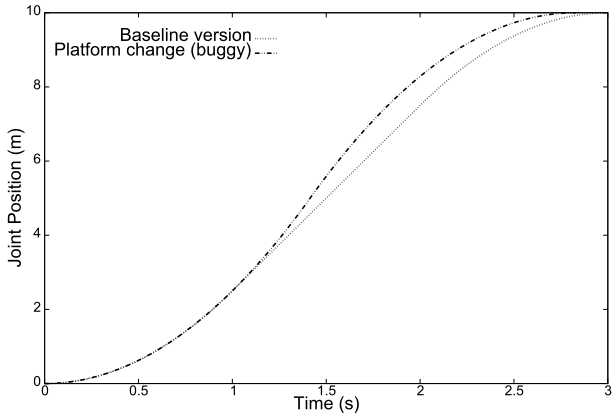

Figure 1. Movement of joint 2 when executing test case 25

equal values is infinitesimal. In facts, the 9 additional test cases from the concolic stage cover all buggy equality constraints in the platform change version.

The new test suite uncovered another (unknown) bug in the programs, due to a division by zero that produces NaN in the trajectory data of quiet joints. The NaN value interferes with the conditional control structures, such that the program fails to update the position of the joint according to the trajectory. The observed outcome is that, if the quiet joint is not first in the list, its movement is tracked exactly equal to the joint that precedes it. Figure 2 illustrates this behavior with reference to test case 25: in both the baseline version and the platform change version, the trajectory of the quiet joint is different from the expected trajectory and equal to the trajectory of the preceding joint (that is, the one illustrated in Figure 1). This bug has been confirmed and indicated as very important by VTT experts.

Platform change (fixed) version: For the version *platform change (fixed)*, that is, the version that fixes the bug introduced in the former Matlab version, the test generator did not produced any additional test case over the test suite of the previous (buggy) version. The final coverage is 82% of the branches of this version.

Replicating the test suite against the fixed platform change version confirmed the correction of the known bug and the consistency of the baseline LabVIEW program and its

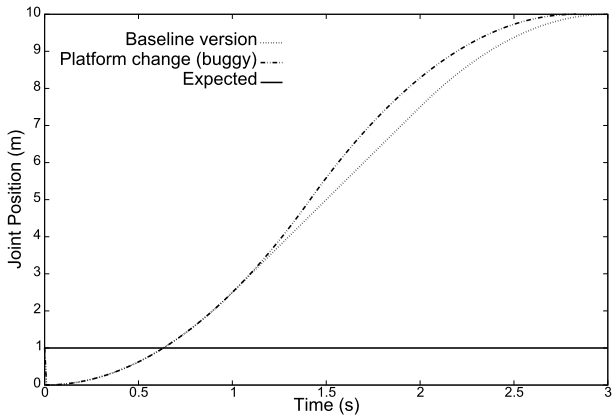

Figure 2. Movement of joint 3 when executing test case 25

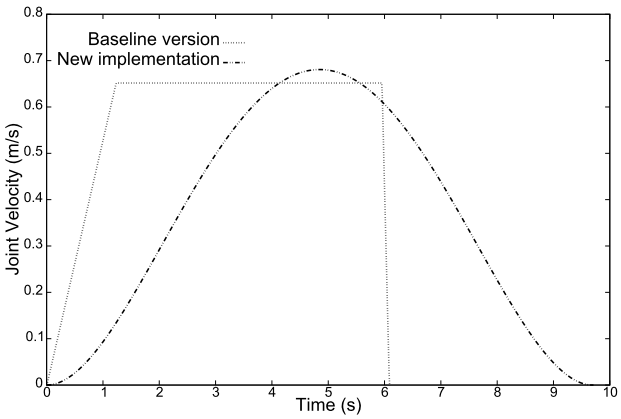

Figure 3. Velocity of joint 2 when executing test case 11

porting to Matlab. The bug in Figure 2 was present in this version too.

New implementation: For the version *new implementation*, that is, the recently proposed re-implementation of the functionality of the baseline version, the test generator produced 3 additional test cases over the test suite of the baseline version, with a total coverage of 83% of the branches. The test case from the random stage did not result in additional coverage of the original code.

We replicated the test suite from the baseline program against the proposed re-implementation of the functionality, and executed the 3 additional test cases computed by test suite augmentation. Overall, the test cases highlighted the expected change of behavior of the new implementation with respect to the baseline program. The new implementation is meant to approximate the gradual (rather than immediate) accelerations of the physical movements. This is evident if we plot the velocity of the joints, as for example in Figure 3 that relates to test case 11. The test revealed problems of the new implementation too: First, the new implementation computes incremental accelerations that always produce single instant peak velocity, and then slower movements than physically possible; Second, it does not account for the maximum deceleration if different from the maximum acceleration, which is an important practical case. These problems can be easily spot in Figure 3.

Replicating the available test cases against a new version is typical regression testing, while we did not observe any notable behavior related to the test cases specifically computed with test suite augmentation for this version. We regard however as a very positive outcome the fact that the automatically generated test cases can produce informative (and readily available) data for a new version of the software that has not been tested in the field yet.

VI. Discussion and Lessons Learned

Overall, the results of our study support the conclusion that test generators can be effective in the considered industrial setting, where they were able to unveil important bugs of the software under test. The considered subject programs are part of a safety critical system and have been extensively used within a prototype deployment of that system. Nonetheless, the test cases delivered by our test generator pinpointed unknown robustness issues with unchecked implicit preconditions and possible floating point underflows, exposed known and unknown subtle bugs, and provided valuable feedback on a recently developed new version of the software. All test outcomes have been reviewed, discussed and confirmed with domain experts.

During the study we learned lessons that is useful to share.

Being able to effectively analyze programs that exploit non-linear and floating point arithmetics was a strong requirement in our study. This likely generalizes to many other relevant industrial domains. We used a floating point simulation library to mitigate the limitations of existing SMT solvers in this respect. Our experience indicates the support for floating point arithmetics and non-linear computations as an important milestone in the path to exploiting the full potential of concolic-based tools in industry.

Another indication that can be drawn out of our study is that test generators must be able to integrate with *manual oracles*, since addressing program crashes or uncaught runtime exceptions only can be insufficient. Our study provides evidence that even low level violations, such as floating point divisions by zero, can result to silent failures.

Integrating with manual oracles is the main reason why we instructed our test generator to retain only the test cases that resulted to increase of coverage: We regarded this as an inexpensive method of controlling the size of the test suites, while delivering the test cases with good chances of capturing behaviors not yet seen. It is however easy to think of other, possibly more effective, methods to achieve a similar

goal, and we do not want to claim any superiority of our method over others. We only remark that industrial users call for prioritized test suites, and exception/crash counting is not always the proper way to achieve a successful prioritization. Test generators must be assessed in this respect as well.

ACKNOWLEDGMENT

This work is partially supported by the European Community under the call FP7-ICT-2009-5 - project PINCETTE 257647. The authors would like to thank Luca Trovato that implemented part of the test cases for execution in LabView.

REFERENCES

[1] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java," *Software—Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004.

[2] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society, 2007, pp. 75–84.

[3] P. Godefroid, A. Kieżun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, 2008, pp. 206–215.

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

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, 2005, pp. 213–223.

[6] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the European Software Engineering Conference joint with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, 2005, pp. 263–272.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, 2006, pp. 322–335.

[8] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF-SE: A symbolic execution extension to Java Pathfinder," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, Braga, Portugal, March 2007, pp. 134–138.

[9] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, 2008.

[10] N. Tillmann and J. de Halleux, "Pex — white box test generation for .NET," in *Proceedings of the 2nd International Conference on Tests and Proofs (TAP 2008)*, 2008, pp. 134–153.

[11] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870 –879, aug 1990.

[12] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, pp. 1085–1110, December 2001.

[13] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*. ACM, 2004, pp. 119–128.

[14] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, 2008, pp. 443–446.

[15] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, 2008, pp. 297–306.

[16] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, June-July 2009, pp. 359–368.

[17] K. L. McMillan, "Lazy annotation for program testing and verification," in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, 2010, pp. 104–118.

[18] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Structural coverage of feasible code," in *Proceedings of the Fifth International Workshop on Automation of Software Test (AST 2010)*, 2010.

[19] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society, 2007, pp. 416–426.

[20] K. Lakhotia, P. McMinn, and M. Harman, "Automated test data generation for coverage: Haven't we solved this problem yet?" in *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*. IEEE Computer Society, 2009, pp. 95–104.

[21] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2008)*, 2008, pp. 151–166.

[22] 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 (ISSTA 2011)*, 2011, pp. 35–44.

[23] Y. Shimomura, "The present status and future prospects of the ITER project," *Journal of Nuclear Materials*, vol. 329-333, no. 1, pp. 5–11, 2004.

[24] M. Keilhacker, "JET deuterium-tritium results and their implications," in *17th IEEE/NPSS Symposium on Fusion Engineering*, vol. 2, 1997, pp. 3–9.

[25] A. Muhammad, S. Esque, M. Tolonen, J. Mattila, P. Nieminen, O. Linna, and M. Vilenius, "Water hydraulic teleoperation system for ITER," in *Proceedings of the 10th Scandinavian International Conference on Fluid Power*, vol. 3, 2007, pp. 263–276.

[26] T. Honda, Y. Hattori, C. Holloway, E. Martin, Y. Matsumoto, T. Matsunobu, T. Suzuki, A. Tesini, V. Baulo, R. Haange, J. Palmer, and K. Shibanuma, "Remote handling systems for ITER," *Fusion Engineering and Design*, vol. 63-64, pp. 507–518, 2002.

[27] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Enhancing structural software coverage by incrementally computing branch executability," *Software Quality Journal*, vol. 19, no. 4, pp. 725–751, 2011.

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