SBFR: A Search Based Approach for Reproducing Failures of Programs with Grammar Based Input

Fitsum Meshesha Kifetew[1], Wei Jin[2], Roberto Tiella[1], Alessandro Orso[2], Paolo Tonella[1]
[1]Fondazione Bruno Kessler, Trento, Italy
[2]Georgia Institute of Technology, USA
kifetew@fbk.eu, weijin@gatech.edu, tiella@fbk.eu, orso@gatech.edu, tonella@fbk.eu

***Abstract*—Reproducing field failures in-house, a step developers must perform when assigned a bug report, is an arduous task. In most cases, developers must be able to reproduce a reported failure using only a stack trace and/or some informal description of the failure. The problem becomes even harder for the large class of programs whose input is highly structured and strictly specified by a grammar. To address this problem, we present SBFR, a search-based failure-reproduction technique for programs with structured input. SBFR formulates failure reproduction as a search problem. Starting from a reported failure and a limited amount of dynamic information about the failure, SBFR exploits the potential of genetic programming to iteratively find legal inputs that can trigger the failure.**

I. Introduction

Due to the many limitations of in-house testing, and the increasing complexity of software, field failures—failures that occur on user machines after the deployment of software—represent a major concern for modern software systems. Reproducing field failures in house is an important step toward understanding and eliminating the bugs that cause such failures. Unfortunately, as shown in a recent survey [1], field failures are notoriously difficult to reproduce outside the time and space in which they occur.

It is therefore not surprising that several researchers, including some of the authors, have proposed techniques to address this problem and proposed possible solutions for it (*e.g.,* [2, 3, 4, 5]). These techniques, however, collect either too little information, and thus do not allow to actually and faithfully reproduce field failures, or too much information, and thus introduce problems of privacy and efficiency. In previous work, we have proposed a general framework, called BugRedux, that can faithfully reproduce field failures both effectively and efficiently using *guided symbolic execution* [6]. Intuitively, our BugRedux algorithm takes as input a sequence of key intermediate points in the observed failing execution and guides the symbolic execution search towards these points until it reaches the point of failure. Our empirical evaluation of BugRedux, performed on a set of real failures, shows that BugRedux can be extremely effective at synthesizing failing executions observed in the field. Our evaluation also shows that BugRedux is particularly effective when provided with (partial) call sequences for the failing executions.

Despite its effectiveness, BugRedux ultimately relies on symbolic execution, which is an inherently limited analysis technique. In particular, symbolic execution is notoriously ineffective when applied to (1) programs with highly structured inputs, such as inputs that must adhere to a (non-trivial) grammar, (2) programs that interact with external libraries, and (3) large complex programs in general (*e.g.,* programs that generate constraints that the constraint solver cannot handle). While there are promising approaches that address some of these issues (*e.g.,* [7, 8]), symbolic execution still has limited effectiveness on these particular types of programs.

To address these limitations, in this work we propose a novel technique for field failure reproduction, called SBFR (Search-Based approach for Failure Reproduction), that (1) is specifically designed for handling programs with complex structured inputs and (2) does not rely on symbolic execution. SBFR takes as input the failing program, a grammar describing the program input, and a (partial) call sequence for the failing execution, and uses a genetic programming approach for generating failing inputs. SBFR relies on genetic operators to manipulate the parse trees of the structured program input. Parse trees are initially constructed by the application of grammar productions, typically following some random technique. They are then evolved through mutation and crossover so as to minimize a fitness function, which measures the distance between the execution trace of each candidate input tree and the execution trace of the failing execution.

II. Background

In this section we provide the background and terminology necessary for describing our approach.

A. Evolutionary Search

Evolutionary search is an optimization heuristic that works by maintaining a population of candidate solutions that are evolved through generations via reproduction and mutation. There are a number of ways in which this evolutionary scheme can be implemented. Among them, the most dominant are Genetic Algorithm (GA) and Genetic Programming (GP).

In a GA, candidate solutions are encoded into *individuals* following various encoding schemes (*e.g.,* binary strings). Initially, a *population* of individuals is generated, usually randomly. The GA then assigns a *fitness value* to each individual in the population by evaluating it using a *fitness function*—a function that measures the "goodness" of each individual with respect to the problem being solved. After performing this evaluation, GA selects pairs of individuals (*parents*) with

978-1-4799-0215-6/13/$31.00 © 2013 IEEE

ASE 2013, Palo Alto, USA
New Ideas Track

better fitness values from the population and subjects them to reproduction (*crossover*) to generate their *offspring*. GA may further subject the offspring to a process of *mutation*, in which the individuals' encoding is modified to introduce diversity into the population. At this point, there is a new *generation* of individuals that can further reproduce. This process of evaluation, selection, and reproduction continues until either the solution is found or a given *stopping condition* (*e.g.,* maximum number of fitness evaluations) is reached.

GP [9] follows a similar process. However, the individuals manipulated by the search are *programs*, rather than encodings of solution instances. These programs usually have a well formed structure defined by a set of formal rules (*e.g.,* a grammar). While there are a number of variants of GP in the literature, in this work we focus on Grammar Guided GP (GGGP), and in particular on Grammatical Evolution (GE) [10]. In GGGP, individuals are sentences generated according to the formal rules prescribed by a grammar. Specifically, in the case of GE, sentences are generated from a Context Free Grammar (CFG), so that new individuals produced by the GP search operators (crossover and mutation) are guaranteed to be valid with respect to the associated grammar. The initial population of individuals is generated from the grammar following a number of techniques, mostly based on some form of random grammar-based generation.

1) Input Representation: An individual in GE is composed of a genotype and a phenotype. The *genotype* is an encoding of the individual that is suitable for representation and manipulation by the search. An example of such encoding is a linear representation of the grammar productions as a sequence of integers, called *codons*. The *phenotype* is the executable version of the individual, which is used as input for the software under test. A *genotype-phenotype* mapping is applied to transform the linear representation into a syntax tree (phenotype). In case of linear representation using codons, the mapping uses the *modulo* operation in conjunction with the grammar of the software being tested. Figure 1 shows an example of such derivation.

An individual—a sentence from the grammar—is obtained through the process of derivation. Starting from the root (start) symbol of the grammar, productions are applied to substitute non-terminal symbols, resulting eventually in a terminal string. The individuals in the initial population are generated from the grammar following the process of sentence derivation. The integer sequence in the genotype (see Figure 1, top) is interpreted as the sequence of productions to be applied to the leftmost non-terminal in the derivation string. Each integer is mapped to the range of actually applicable productions by computing the modulo over the total number of productions applicable to the non-terminal being expanded. Integer codon 102 from the genotype in Figure 1, for instance, is applied to the non-terminal `<atom>`, which admits four expansions. Because 102 % 4 gives 2, the third of the four possible productions is applied. The next integer in the genotype, 311, is applied to the `<t_TRIG>` non-terminal, which is the leftmost non-terminal in the current derivation

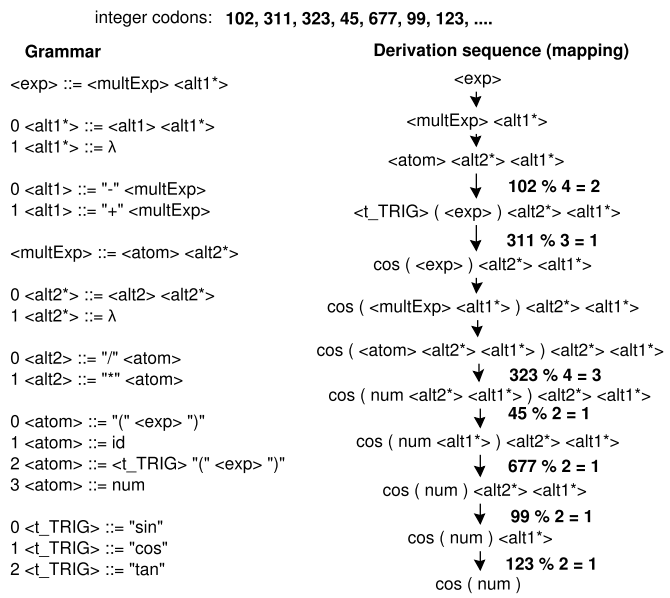

Fig. 1. An example of a Genotype-Phenotype Mapping in GE. Each non-terminal is expanded by one of its productions, whose number is obtained by applying the modulo operator on the next available codon value over the number of productions available for the non-terminal being expanded.

string `<t_TRIG> ( <exp> ) <alt2*> <alt1*>`. Because 311 % 3 = 1, the second production for `<t_TRIG>` is applied. Of course, if the leftmost non-terminal has only one production, no integer is consumed from the genotype, and the sole production is used to expand the non-terminal.

2) Evolution Operators: Evolution operators (crossover and mutation) play a crucial role in the evolutionary search process. Crossover and mutation could be performed in a number of different ways. In GP, sub-tree crossover and mutation are commonly used. *Sub-tree crossover* between two individuals is performed by exchanging two *sub-trees*, rooted at the same non-terminal, from their tree representations. Sub-tree crossover ensures that the newly formed individuals are well formed with respect to the underlying grammar. An example of sub-tree crossover is shown in Figure 2.

Similarly, *sub-tree mutation* on an individual is performed by replacing a *sub-tree* from its tree representation with a new sub-tree of the same type generated from the grammar. Figure 3 shows an example of sub-tree mutation.

B. Terminology

A *field failure* is a failure of a deployed program while it executes on a user machine. We use the term *execution data* to refer to any runtime information collected from a program executing on a user machine. In particular, a *(partial) call sequence* is a specific type of execution data that consists of a (sub)sequence of functions invoked during the execution of a program on a user machine. We define a *field failure reproduction* technique as a technique that can synthesize, given a program P, a field execution E of P that results in a failure F, and a set of execution data D for E, an in-house

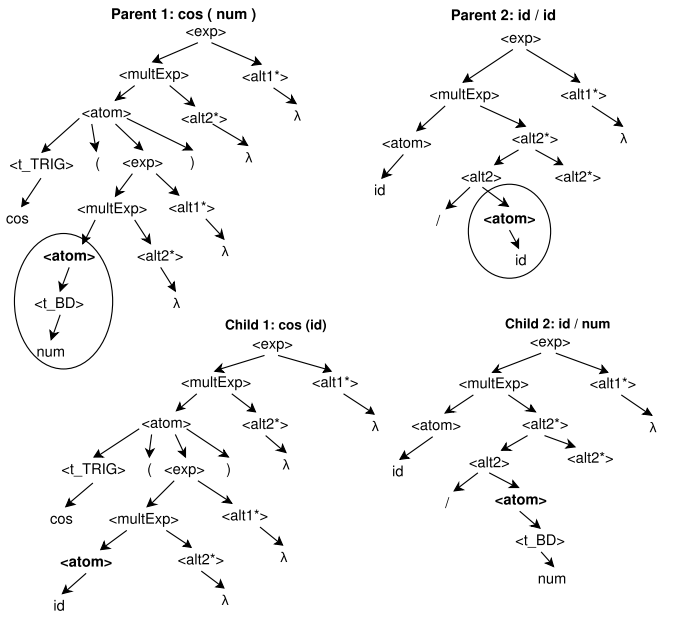

Fig. 2. Sub-tree crossover: sub-trees of the same type (in circles) from *parents* are exchanged to create *children*. See Figure 1 for the derivation process.

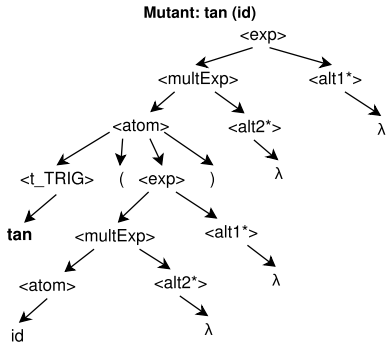

Fig. 3. Sub-tree mutation: a sub-tree is replaced by a new sub-tree of the same type generated from the grammar. *Child1* from Figure 2 is mutated.

execution E' as follows. *First*, E' should result in a failure F' that is analogous to F, that is, F' has the same observable behavior of F—it generates the same execution data D and fails "in the same way." If F results in the violation of an assertion at a given location in P, and we are using partial call sequences as execution data, for instance, F' should result in the same partial call sequence and violate the same assertion at the same point. *Second*, E' should be an actual execution of P, that is, the approach should be sound and generate an actual input that, when provided to P, results in E'. *Finally*, the approach should be able to generate E' using only P and D, without the need for any additional information.

III. Search Based Failure Reproduction

Figure 4 illustrates our overall failure reproduction approach, which we call SBFR. As we discussed earlier, SBFR's goal is to reproduce an observed field failure based on a set

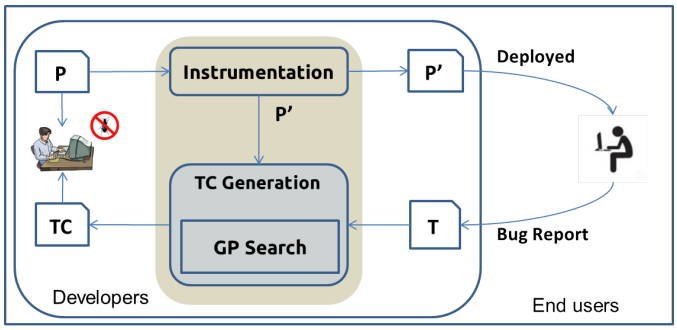

Fig. 4. Overview of SBFR: the instrumenter takes as input a Program P and generates an instrumented application P' that is deployed to the users and produces execution data. Upon failure, the execution data T are used by GP Search to generate a test case TC that reproduces the observed failure. The TC is then used by the developers to debug the failure.

of (ideally minimal) execution data (*i.e.,* runtime information about the failure). SBFR collects this execution data by instrumenting the software under test (SUT) before deploying it to the users. Upon failure, the execution data of the failing execution is used by SBFR to perform a GP based search for inputs that can trigger the observed failure while the SUT was running on the user machine. Note that this part of the approach could be performed in the field, by leveraging free cycles on the user machine, so as to send back only the generated inputs instead of any execution data.

As in our previous BugRedux work [6], the execution data is used to provide guidance to the search. However, unlike BugRedux, SBFR performs the search for the failure inducing input using evolutionary search, and the guidance is provided indirectly, through the use of the fitness function discussed in Section III-C. More precisely, the *individuals* in the GP search are candidate *test inputs* for the SUT, that is, structured input strings that adhere to a formal grammar. The search maintains a population of individuals, evaluates them by measuring how close they get to the desired solution using a fitness function, and evolves them via genetic operators (see Section III-B). If at least one candidate is able to trigger the desired failure, a solution is found and the search terminates. If SBFR finds no candidate solution after consuming the whole search budget, it deems the search unsuccessful. In the rest of this section, we present more details on this part of the approach.

A. Seeding the Search with Representative Inputs

When SBFR generates the initial population, uniform selection of the productions to use (*i.e.,* uniform generation of random integer sequences) tends to run into problems in the presence of recursive productions, which are quite frequent in commonly used grammars. In fact, frequent application of recursive productions (*e.g.,* production *0* for `<alt1*>` in Figure 1) may result in derivation strings that contain some residual non-terminals even after the entire genotype has been used. When the resulting individual contains non-terminals, the evolutionary search process discards it. If the application of recursive productions is not controlled, the individuals that

are left after discarding those containing non-terminals tend to be associated with shallow derivation trees and short, non representative strings.

In an effort to address this problem, the search can assign probabilities to the productions so that the recursive ones are applied less frequently than the non-recursive ones [11]. In this way, the sentences generated in the initial population will be of balanced depth and more representative than those generated using a uniform random approach. SBFR uses a simple, yet effective 80/20 rule to achieve this result: it uniformly distributes a total probability of 0.2 across all recursive productions, and a total probability of 0.8 across the non-recursive productions. (SBFR treats indirect recursion as plain recursion in the application of the 80/20 rule.) With reference to the example in Figure 1, SBFR would assign probability 0.2 to production 0 (a recursive production) for non-terminal `<alt1*>`, and probability 0.8 to production 1 (a non-recursive production).

B. Input Representation and Genetic Operators

Once the initial population of individuals is generated as discussed in the previous subsection, the individuals are linearly represented as a sequence of codons (integer chromosomes). A tree representation is then built from the integer codons following the derivation procedure discussed in Section II. The genetic operators manipulate this tree representation of the individuals.

SBFR applies sub-tree crossover and mutation operations on the individuals (see Section II). We choose to use tree based operations because they preserve the well formedness of the resulting individuals—if both parents are well formed individuals (according to the grammar), the offspring produced by sub-tree crossover are also well formed. Similarly, sub-tree mutation of a well formed individual results in a well formed individual. The probabilities used in sub-tree mutation are the same as those used to generate the initial population.

To evaluate the fitness of an individual, SBFR transforms its tree representation into a string representation and passes the resulting string to the SUT as input. Based on the execution of the SUT on the input string, SBFR then computes the fitness of the individual, as explained in the next subsection.

C. Fitness Computation and Search Termination

The success of a metaheuristic search heavily depends on the fitness function used to evaluate the candidate solutions. The fitness function is expected to provide a ranking among individuals with respect to their closeness to the desired solution.

SBFR evaluates candidate solutions based on the trace obtained when executing them against the SUT. To evaluate how good a candidate individual is, the instrumented SUT is executed using the candidate individual as input, resulting in a set of execution data. In this work, we consider execution data that consist of call sequences and refer to a call sequence using the term *trajectory*. More formally, we define a trajectory as a sequence $T = \langle c_1, ..., c_n \rangle$, where each c_i is a function/method call. We make this choice because our findings in previous work show that call sequences can provide the best tradeoffs in terms of cost benefit for synthesizing in-house executions [6]. Further, call sequences are unlikely to reveal sensitive or confidential information about the original execution.

In our customized GP approach, we define a *fitness function* based on the *distance* between the trajectory of the failing execution and the trajectory of a candidate individual. Hence, our GP approach tries to *minimize* this distance with the objective of finding individuals that generate trajectories *similar* to that of the failing execution.

To compute the distance between two trajectories, we rely on sequence alignment algorithms that determine the similarity of the two trajectories. The distance between two trajectories T_1 and T_2 can be defined as:

$$distance(T_1, T_2) = |T_1| + |T_2| - 2 * |LCS(T_1, T_2)| \quad (1)$$

where LCS stands for Longest Common Subsequence [12], and $|T|$ is the length of trajectory T.

For instance, $T_1 = \langle f, g, g, h, m, n \rangle$ and $T_2 = \langle f, g, h, m, m \rangle$ have $LCS = \langle f, g, h, m \rangle$. Hence, their distance is 6 + 5 - 8 = 3, which corresponds to the number of calls that appear only in T_1 (second g, n) or in T_2 (second m).

The search stops when either a desired solution is found or the search budget, expressed as the number of fitness evaluations, is exhausted. If successful, the search will produce an individual (*i.e.,* an input) that causes the program to follow a trajectory similar to that of the observed failure, reaching the point of failure. Failure reproduction is deemed successful if the SUT fails at that point with the same observable failing behavior as the original failure.

It is worth noting that, even when unsuccessful, SBFR can still give developers valuable information about the structure of the failure inducing input. In these cases, the evolutionary search would still produce a set of "best" possible solutions, which may be close to the actually failing executions.

IV. Related Work

In this section, we focus on two research topics that are most closely related to our approach: test input generation and field failure reproduction.

A. Test Input Generation

Symbolic execution is a systematic approach for generating test inputs that traverse as many different control flow paths as possible (all paths, asymptotically). Symbolic execution was first introduced almost four decades ago [13]. The dramatic growth in the computational power of today's computers, together with the availability of increasingly powerful decision procedures, has resulted in a renewed interest in using symbolic execution for test-input generation and in the development of a number of new symbolic execution techniques and tools (*e.g.,* [14, 15, 16, 17, 18, 19]). Despite these recent advances, however, symbolic execution is still an inherently limited technique, mostly due to the path explosion problem (*i.e.,* the virtually infinite number of paths in the code),

the environment problem (*i.e.,* the challenges involved with handling interactions between the code and its environment, such as external libraries), and the limitations of constraint solvers in handling complex constraints and theories.

Application of symbolic execution to programs with complex, structured inputs that must adhere to a formal language specification (*e.g.,* the media format accepted by a media player or the input program accepted by an interpreter) is a more recent topic of investigation. Existing approaches to this problem (*e.g.,* [7, 8]) rely on the creation of symbolic tokens associated with the non terminals in the grammar productions. Such tokens are propagated through the program when it is executed symbolically. While promising, these approaches also suffer from some of the same limitations of traditional symbolic execution.

A type of test input generation techniques alternative to symbolic execution are those that build upon search based algorithms, and in particular genetic algorithms (*e.g.,* [20, 21]). In these approaches, an initial population of test cases, usually generated randomly, is evolved by means of mutation and crossover operators. The evolution is performed so as to optimize a fitness function that typically accounts for the degree of coverage achieved or for the distance from the coverage target of each test case. The main limitation of search based techniques, when compared to symbolic execution, is that they can get stuck in local optima and can only be indirectly (*i.e.,* through the fitness function) guided towards a target of interest. However, search based techniques have the great advantage that they can scale to much larger programs and are not affected by the environment problem or by the complexity of the path constraints. In addition, genetic programming is particularly suitable for generating complex, structured inputs (*e.g.,* [10, 9]).

B. Field Failure Reproduction

One type of techniques that can be used for field failure reproduction are those capturing and recording program behaviors by monitoring or sampling field executions (*e.g.,* [3, 22, 4]). These techniques, however, tend to either record too much information to be practical or too little information to be effective.

For this reason, researchers started investigating more sophisticated approaches to reproduce field failures using more limited information. Some debugging techniques, for instance, leverage weakest-precondition computation to generate inputs that can trigger certain types of exceptions in Java programs (*e.g.,* [23, 24, 25]). Although potentially promising, these approaches tend to handle limited types of exceptions and to operate mostly at the module level. SherLog [26] and its follow-up work LogEnhancer [27] use runtime logs to reconstruct and infer paths close to logging statements to help developers identify bugs. These techniques have shown to be effective, but they aim to highlight potential faulty code, rather than synthesizing failing executions.

ReCrash [28] records partial object states at the method level dynamically to recreate an observed crash. It inspects the call stack (collected upon a crash) at different levels of stack depth and tries to call each method in the stack with parameters capable of reproducing the failure. Although this approach can help reproduce a field failure, it either captures large amounts of program states, which makes it impractical, or reproduces the crash in a shallow way, at the module or even method level, which has limited usefulness (*e.g.,* making a method fail by calling it with a *null* parameter does not provide useful information for the developer, who is rather interested in knowing why a *null* value reached the method).

Both ESD [29] and CBZ [30] leverage symbolic execution to generate program inputs that reproduce an observed field failure. Specifically, ESD aims at reaching the point of failure (POF), whereas CBZ improves ESD by reproducing executions that follow partial branch traces, where the relevant branches are identified by different static and dynamic analyses. However, as some of the authors have shown in a previous paper [6], POFs and partial traces are unlikely to be successful for some failures.

Similar to ESD and CBZ, BugRedux [6] is a general approach for synthesizing, in-house, executions that mimic observed field failures. The core of BugRedux is a guided symbolic execution algorithm that aims to reach a sequence of intermediate points in an execution. Although an empirical evaluation of BugRedux has shown that it can reproduce real-world field failures effectively and efficiently, given a suitable set of field execution data, the approach is based on symbolic execution and suffers from the inherent problems of these kinds of techniques. In fact, when we tried to apply BugRedux to more and larger programs, and in particular to programs with grammar based input, we found that its effectiveness is limited.

Another approach, RECORE [31], applies genetic algorithms to synthesize executions from crash call stacks. However, the current empirical evaluation of RECORE focuses on partial executions (*i.e.,* executions of standalone library classes), so it is unclear whether the approach would be able to reproduce complete executions. Failures in library classes usually result in shallow crash stacks, and in our experience execution synthesis approaches based on symbolic execution tend to work well in these cases.

Some techniques specifically focus on reproducing concurrency-related failures. Among those, two worth mentioning are ESD [29], which we just discussed, and PRES [32]. The approach we propose in this paper focuses on failures of sequential programs and leave concurrency-related failures for future work. In this sense, our technique is orthogonal and complementary to these techniques, and we could combine these techniques with our approach.

In summary, our careful study of (and extensive experimentation with) state-of-the-art techniques confirmed that test input generation techniques based on symbolic analysis are effective but have a number of practical limitations, and are thus ideal for reproducing failure in programs of limited size and complexity. Conversely, approaches based on search based algorithms are often not as powerful as approaches based on

symbolic analysis but can scale to larger and more complex programs that symbolic analysis cannot handle—and are thus amenable to failure reproduction for this kind of programs. In fact, to the best of our knowledge, SBFR is the first approach that aims at *fully reproducing system-level* field failures for programs with *complex, structured input*.

V. Conclusion and Future Work

We proposed the use of evolutionary search for the generation of complex and structured test inputs capable of reproducing failures observed in the field. Our approach, called SBFR, exploits the potential of genetic programming to evolve candidate solutions using genetic operators that manipulate parse-tree representations of the inputs. The fitness function that guides the evolution of these trees rewards test inputs associated with execution traces that are similar to the execution trace of the observed failure.

We are currently conducting an empirical evaluation of SBFR to assess its effectiveness by applying it to a set of programs with complex input grammars, and our initial results are promising. In future work, besides continuing our experiments, we will investigate whether our fitness function can be further improved to increase the probability of failure reproduction and/or to achieve failure reproduction more quickly. We will also explore the impact of using grammars learned from a corpus of inputs for the generation of the initial population used in the search.

Acknowledgments

This work was partially supported by NSF awards CCF-1320783, CCF-1161821, and CCF-0964647 to Georgia Tech, and by funding from IBM Research and Microsoft Research.

References

[1] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss, "What Makes a Good Bug Report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, Sep. 2010.

[2] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective Statistical Debugging via Efficient Path Profiling," in *ICSE 2009*, 2009, pp. 34–44.

[3] J. Clause and A. Orso, "A Technique for Enabling and Supporting Debugging of Field Failures," in *ICSE 2007*, 2007, pp. 261–270.

[4] "The Amazing VM Record/Replay Feature in VMware Workstation 6," http://communities.vmware.com/community/vmtn/cto/steve/blog/2007/04/18/the-amazing-vm-recordreplay-feature-in-vmware-workstation-6, Apr. 2012.

[5] L. Jiang and Z. Su, "Context-aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 184–193.

[6] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *Proc. of the 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 474–484.

[7] R. Majumdar and R.-G. Xu, "Directed Test Generation Using Symbolic Grammars," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 134–143.

[8] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based Whitebox Fuzzing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 206–215.

[9] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'Neill, "Grammar-based Genetic Programming: a survey," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 365–396, May 2010.

[10] M. O'Neill and C. Ryan, "Grammatical Evolution," *Evolutionary Computation, IEEE Transactions on*, vol. 5, no. 4, pp. 349–358, Aug. 2001.

[11] K. Lari and S. J. Young, "The Estimation of Stochastic Context-free Grammars Using the Inside-outside Algorithm," *Computer speech & language*, vol. 4, no. 1, pp. 35–56, 1990.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[13] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[14] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proc. of the 13th ACM Conference on Computer and Communications Security*, 2006, pp. 322–335.

[15] W. Grieskamp, N. Tillmann, and W. Schulte, "XRT–Exploring Runtime for .NET Architecture and Applications." *Electr. Notes Theor. Comp. Sci.*, vol. 144, no. 3, pp. 3–26, 2006.

[16] 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 Conference on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[17] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test Input Generation with Java PathFinder," *SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[18] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005, pp. 263–272.

[19] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[20] M. Harman and P. McMinn, "A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2010.

[21] P. McMinn, "Search-based Software Test Data Generation: a survey," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[22] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," in *PLDI 2005*, 2005, pp. 15–26.

[23] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: A Powerful Approach to Weakest Preconditions," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 363–374.

[24] M. G. Nanda and S. Sinha, "Accurate Interprocedural Null-Dereference Analysis for Java," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 133–143.

[25] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002, pp. 234–245.

[26] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: Error Diagnosis by Connecting Clues from Run-time Logs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 143–154.

[27] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving Software Diagnosability via Log Enhancement," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 3–14.

[28] S. Artzi, S. Kim, and M. D. Ernst, "ReCrash: Making Software Failures Reproducible by Preserving Object States," in *Proceedings of the 22nd European Conference on Object-Oriented Programming*, 2008, pp. 542–565.

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

[30] O. Crameri, R. Bianchini, and W. Zwaenepoel, "Striking a New Balance Between Program Instrumentation and Debugging Time," in *Proceedings of the 6th European Conference on Computer Systems*, 2011, pp. 199–214.

[31] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea, "Reconstructing Core Dumps," in *Proc. of the 6th International Conference on Software Testing*, 2013, pp. 114–123.

[32] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: Probabilistic Replay with Execution Sketching on Multiprocessors," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009, pp. 177–192.