2016 IEEE International Conference on Software Maintenance and Evolution

Empirical Study on Synthesis Engines for Semantics-Based Program Repair

Xuan-Bach D. Le*, David Lo*, and Claire Le Goues†
*School of Information Systems, Singapore Management University, Singapore
†School of Computer Science, Carnegie Mellon University
Email: {dxb.le.2013,davidlo}@smu.edu.sg, clegoues@cs.cmu.edu

***Abstract*—Automatic Program Repair (APR) is an emerging and rapidly growing research area, with many techniques proposed to repair defective software. One notable state-of-the-art line of APR approaches is known as *semantics-based* techniques, e.g., Angelix, which extract semantics constraints, i.e., *specifications*, via symbolic execution and test suites, and then generate repairs conforming to these constraints using program synthesis. The repair capability of such approaches—expressive power, output quality, and scalability—naturally depends on the underlying synthesis technique. However, despite recent advances in program synthesis, not much attention has been paid to assess, compare, or leverage the variety of available synthesis engine capabilities in an APR context.**

In this paper, we empirically compare the effectiveness of different synthesis engines for program repair. We do this by implementing a framework on top of the latest semantics-based APR technique, Angelix, that allows us to use different such engines. For this preliminary study, we use a subset of bugs in the IntroClass benchmark, a dataset of many small programs recently proposed for use in evaluating APR techniques, with a focus on assessing output quality. Our initial findings suggest that different synthesis engines have their own strengths and weaknesses, and future work on semantics-based APR should explore innovative ways to exploit and combine multiple synthesis engines.

***Index Terms*—Automated Program Repair, Program Synthesis Engine, Empirical Study**

I. Introduction

Bugs are prevalent in software development, incurring a significant cost to software production in both commercial and open-source software. Fixing bugs is thus crucial to maintaining software quality. However, bug fixing is known to be difficult, time-consuming, laborious, and very expensive [22]. Hence, automated program repair (APR) techniques that can help developers tackle the bug-fixing challenge would be of tremendous value.

The once-futuristic idea of APR has been gradually materializing in the form of numerous recent research advances [15], [16], [11], [12], [13], [9], [7], [14]. APR techniques can be generally classified into two families, heuristic versus semantics-based, each with different strengths and weaknesses. Heuristic APR techniques typically employ various syntactic mutation operators to produce large populations of possible candidates bug-fixing patches, and then search for the best one with respect to an optimization function (typically, though not exclusively, patched program behavior on a set of test cases). Meanwhile, semantics-based techniques extract semantics constraints, i.e., *specifications* from behavior on test suites, and leverage program synthesis to synthesize repaired expressions that conform to these extracted specifications. Excitingly, semantics-based approaches have recently been shown to scale to repair bugs in large, real-world programs, comparable to those targeted by heuristic approaches [16]. In this work, we focus on semantics-based APR approaches.

Naturally, one of the main factors that influence the success of semantics-based APR is the power of the underlying program synthesis approach. Output patch quality is an especially pressing concern: Given that the specifications extracted by semantics-based approach are only partial, synthesis may generate a "plausible" but not fully correct solution—that is, a patch that satisfies the given specification, but does not generalize to the full desired specification. This phenomenon has been explored for heuristic approaches [21], but to the best of our knowledge has not been investigated for semantics-based approaches.

In this paper, we study the performance of several program synthesis engines for semantics-based APR. We propose a novel mechanism for integrating syntax-guided synthesis (SyGuS) into a semantic repair technique, and then show that the different synthesis approaches do indeed perform differently in the context of program repair. In particular, the synthesis approach employed in Angelix [16], the state-of-the-art in semantics-based APR, may not be best for all bugs. Instead, different synthesis techniques can complement one another, increasing the effectiveness of semantics-based APR. We argue that semantics-based APR can and should make use of multiple synthesis engines, and suggest future research on designing ways to *forge* multiple synthesis engines to produce an approach that can fix more bugs with less patch overfitting. We also suggest future research on benchmarks and metrics for synthesis in the context of APR, beyond the traditional benchmarks specialized for synthesis task alone [1].

The remainder of this paper is structured as follows. Section II explains background on semantics-based APR. Section III describes our pluggable framework built on top of Angelix that can use different synthesis engines for APR. Experiment results are presented in Section IV, followed by threats to validity in Section IV-C. Section V concludes the paper and mentions future directions.

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.68

II. Background

A. Angelix: A State-of-the-Art Semantics-Based APR

To the best of our knowledge, Angelix is the most recently-proposed state-of-the-art semantics-based program repair approach [16]. Angelix follows a now-standard model for test-case-driven program repair, taking as input a program and a set of test cases, at least one of which is failing. The goal is to produce a small set of changes to the input program that corrects the failing test case while preserving the other correct behavior. Like many APR techniques, Angelix first uses fault localization, e.g., Ochiai [2] to identify likely-buggy expressions. Angelix then performs a "controlled" symbolic execution, wherein symbolic variables are installed only at chosen buggy expressions, to collect path constraints.[1] Angelix uses the constraints to extract *value-based specifications* for each of the candidate buggy expressions, which contain possible concrete values of variables and expressions that satisfy the constraints, thereby making all tests pass. The extracted value-based specifications take the form of a *precondition* on the values of variables before a buggy expression is executed and a *postcondition* on the values after that expression is executed. The precondition is extracted by using forward analysis on the test inputs to the point of the buggy expression; The postcondition is extracted via backward analysis from the desired test output. The problem of program repair can now be reduced to a synthesis problem: Given a precondition, Angelix synthesizes an expression that satisfies the postcondition. Angelix uses a slightly modified version of oracle-guided component-based synthesis [6] for this task. To minimize the size of the change, Angelix eagerly preserves the structure of the original expression, leveraging Partial MaxSMT [15]. This repeats until a repair that causes the program to pass all tests is synthesized.

B. Syntax-Guided Program Synthesis Engines

The primary goal of *program synthesis* is to automatically generate an implementation of a program that provably satisfies a given specification. Recent years have seen many proposed program synthesis approaches [3], [18], [17], wherein syntax-guided synthesis (SyGuS) [3] is arguably one of the most notably successful approaches. Rather than trying to synthesize a program with arbitrary syntax, SyGuS engines use a restricted grammar to describe the syntactic space of possible implementations, reducing the search space for the correct implementation. We first present the generic grammar of SyGuS's input and output, and then briefly explain a selection of existing, publicly-available SyGuS engines: *enumerative*, *stochastic*, *symbolic*, and *CVC4* [3], [20].[2]

The grammar of SyGuS's input and output is briefly described in Figure 1. An expression in SyGuS can be either an integer or a boolean expression. An integer expression contains constants, variable names, addition, or subtraction of two integer types. A boolean expression is defined similarly. A SyGuS synthesis technique will then search for solutions that conform to its provided grammar only. We note that a grammar for each synthesis problem can vary, e.g., have fewer permitted rules than those rules defined in Figure 1, depending on particular scenarios, as we will explain in Section III.

Figure 2 depicts an example script that instructs a SyGuS synthesizer to synthesize a function named *leq*, according to the grammar described by the *synth-fun* keyword, in the domain of Linear Integer Arithmetic (denoted as *LIA*). This function has two arguments of type integer, and a boolean return type. Integer and boolean expressions are constrained as shown in the example grammar (Figure 1). The specification of this function is then constrained by the input-output examples expressed via the *constraint* keyword. The first constraint says that if the value of x is 1, and the value of y is 2 (precondition), then the function *leq* over x and y will return an expression evaluated to *true* (postcondition). In this example, a SyGuS synthesizer could return an implementation of the function *leq* as $x \leq y$.

Different SyGuS engines then employ different search strategies to generate or synthesize a solution conforming to such a specification. The *enumerative* engine generates candidate expressions in increasing size, and leverages the specification to prune the search space of possible candidates. The *stochastic* search engine searches for a correct implementation among the search space using an optimization function indicating a probability that a candidate satisfies the provided specification. The *symbolic* approach uses a constraint solver both to search for a candidate expression satisfying a set of concrete input examples, and to verify the validity of an expression for all possible inputs. *CVC4* is the first synthesizer implemented inside an SMT solver, via a slight modification of the solver's background theory. To synthesize an implementation that satisfies all possible inputs, it translates the challenging problem of solving universal quantifier over all inputs into showing the unsatisfiability of the negation of the given specification. It then synthesizes a desired solution based on the unsatisfiability proof. Recent competitions of SyGuS techniques showed that *CVC4* and *enumerative* engines are the among the best engines on benchmarks specialized for assessing SyGuS [1].

$$
\begin{aligned}
&\mathit{IntExpr} := \mathcal{N} \mid \mathit{Var} \mid \mathit{IntExpr}\ \mathit{BinOp}\ \mathit{IntExpr}\\
&\mathit{BinOp} := + \mid -\\
&\mathit{BoolExpr} := \mathit{true} \mid \mathit{false} \mid \mathit{Var} \mid \neg\ \mathit{BoolExpr}\\
&\qquad \mid \mathit{IntExpr}\ \mathit{RelOp}\ \mathit{IntExpr} \mid \mathit{BoolExpr}\ \mathit{LogOp}\ \mathit{BoolExpr} \mid\\
&\qquad \mathit{IntExpr}\ \mathit{EqOp}\ \mathit{IntExpr} \mid \mathit{BoolExpr}\ \mathit{EqOp}\ \mathit{BoolExpr} \mid\\
&\mathit{RelOp} := > \mid < \mid \leq \mid \geq\\
&\mathit{LogOp} := \wedge \mid \vee\\
&\mathit{EqOp} := =
\end{aligned}
$$

Fig. 1: Generic SyGuS's Input and Output Grammar

[1] Angelix can target multiple buggy expressions at once and scales with the number of locations considered; we explain the process with respect to a single buggy expression for clarity, but the process generalizes naturally.

[2] We refer interested readers to [3], [20] for additional details.

```
(set-logic LIA)
(synth-fun leq ((x Int) (y Int)) Bool
((Start Int (1 2 5 x y
      (+ Start Start)
      (- Start Start)))
 (StartBool Bool (
      (and StartBool StartBool)
      (or StartBool StartBool)
      (not StartBool)
      (≤ Start Start)
      (< Start Start)))
)); end definition of function leq
(declare-var x Int)
(declare-var y Int)
(constraint (⇒ (and (= x 1) (= y 2)) (= (leq x y) true)))
(constraint (⇒ (and (= x 2) (= y 2)) (= (leq x y) true)))
(constraint (⇒ (and (= x 5) (= y 2)) (= (leq x y) false)))
(check-synth)
```

Fig. 2: Example of a SyGuS Script

III. Pluggable Framework for Semantics-Based Program Repair

In this section, we describe our framework that allows Angelix to use different SyGuS synthesizers.[3] We explain our translation from Angelix's value-based specifications to a SyGuS script, and our heuristic to allow SyGuS synthesizers to generate solutions that are minimally different from the original buggy expression, based on the given value-based specifications.

From value-based specifications to SyGuS script: Angelix infers a value-based specification for each buggy location/-expression, that it then uses to inform the synthesis of a possibly multi-line patch. We need to translate this value-based specification into a SyGuS synthesis problem, wherein each of the locations under repair corresponds to a function to be synthesized with respect to its corresponding specification. We must further map the output of successful SyGuS synthesis back to its corresponding buggy locations in the original program to construct a repair patch.

Our translation proceeds as follows: Each buggy location is associated with a to-be-synthesized function. Each function is constrained by its own grammar, which we "dynamically" generate based on the value-based specification, the original buggy expression, and synthesis level.[4] Consider the example in Figure 2. If the original buggy expression is $x == y$, the test cases/specification for this expression is as shown in Table I, and the synthesis level is "integer-constants" and "alternatives", we generate the grammar for the function corresponding to this expression as in Figure 2. This function takes as input two integer arguments x and y, and returns an expression evaluated to a boolean type. Since the function *leq* requires both types *Int* and *Bool*, we generate the definition of both types in the function's grammar. The integer expression in this grammar permits constants: $1, 2$, and 5, as taken from the specification, and variable x and y as taken from the arguments. The boolean expression in this grammar allows operators $\leq$ and $<$, that are alternatives for the buggy operator $==$ in the original buggy expression, following the "alternatives" synthesis level. The set of *constraint* statements is generated by traversing the specification, in which each constraint encodes the desired behavior of each test case (recall that the specification contains possible concrete values of variables and expressions that make all tests pass).

Constructing likely closest solution to original buggy expression: Angelix uses Partial MaxSMT to minimize the amount of behavioral change in a produced repair, in the interest of increasing the probability of a higher-quality patch. SyGuS synthesis techniques, unfortunately, can not inherently handle this problem directly. Thus, we heuristically constrain candidate repairs in the tranlation from the value-based specification to a SyGuS. Two ways to do this are to: (1) Dynamically force the functions' grammar to mimic the structure of the original expression, or (2) identify subexpressions that are unlikely to change in the original expression over the course of the repair, to help reduce the amount of new code to be synthesized. We adopt the second approach, leaving the first approach for future work.

To find subexpressions that are unlikely to change in the original expression e, we take into account the value-based specifications, which contains values of variables involved in e. That is, we filter out subexpressions in e that involve variables whose values do not change in the specification. To illustrate, consider an original buggy expression e: $(x \leq 5) \wedge (y == 2)$, with a specification as described in Table I, and a SyGuS script shown in Figure 2. The expected correct expression that satisfies the specification in Table I is $(x \leq 2) \wedge (y == 2)$. Without any optimization, a SyGuS synthesizer could return the expression: $x \leq 2$ as a solution. Although this solution satisfies the specification, it does not minimally change the behavior of the original expression e. Our optimization, on the other hand, identifies that the subexpression $y == 2$ is very unlikely to be changed, since the value of y is unchanged through out the specifications. Our optimization further identifies that the expression $x \leq 5$ is likely to be changed, since the value of x changes, and also the value of its "neighbor" (the constant 5 involved in the same operator with x) never changes through out the specification. We therefore instruct a SyGuS synthesizer to synthesize a function involving only variable x, and various constants. After receiving the result from the synthesizer, such as the expression $x \leq 2$, we map it back to original expression e, leaving the unchanged subexpressions $(y == 2)$ intact. This way, we can generate the expected correct expression, close to the original expression e.

More formally, we define subexpression(s) e_{sub} in an original buggy expression that are unlikely to be changed if e_{sub} involves constants or variables whose values are unchanged throughout the specification. If e_{sub} is composed of several subexpressions e^i_{sub}, each e^i_{sub} must also be unchanged. This

[3]Our framework is built on top of Angelix, and is available here: https://github.com/xuanbachle/syntax-guided-synthesis-repair/.

[4]At each synthesis level, particular set of components will be permitted in synthesis task via the dynamically generated grammar.

TABLE I: Value-based specifications for expression: $x == y$

Test #	Value of x	Value of y	Expected Output
1	1	2	true
2	2	2	true
3	5	2	false

TABLE II: Subset of IntroClass Dataset. Each column shows the name of the program and the numer of versions of that program that fail on black-box tests.

Median	Smallest	Digits
61	67	60
Total: 188 programs		

way, we eagerly identify the largest possible set of unchanged subexpressions in the original buggy expression, reducing the burden of synthesizing large expression on the SyGuS synthesizers.

IV. Experiments and Analysis

A. Benchmark Dataset and Evaluation Metrics

We evaluate four SyGuS synthesis engines including *enumerative*, *stochastic*, *symbolic*, and *CVC4*, and Angelix's Partial MaxSMT-based synthesis engine, on programs from the IntroClass benchmark [10], which consists of student-written programs with defects submitted as homework to a freshman programming class. Each program in the benchmark has two independent high-coverage test suites: a black-box test suite written by the course instructor, and a white-box test suite generated by the test generation tool KLEE [4] on a reference solution. The dataset is described in Table II.

Although IntroClass contains small programs, it is a particularly suitable benchmark for assessing repair quality via overfitting, because of the two high-coverage test suites associated with each program [21]. Our experimental data is a subset of the IntroClass benchmark, because Angelix can only handle programs whose output are of boolean, integer, or character types. We therefore do not consider IntroClass programs whose outputs cannot be handled by Angelix, e.g., those with string output.

For each of the 188 programs, we run each synthesis technique on the black-box tests to generate repairs, and use the white-box tests as *held-out* tests to assess the quality of generated repairs. If a generated repair does not pass all of the held-out tests, we say that it *overfits* to the training test cases and is not fully general; this is a proxy for repair quality (or lack thereof). We assess the synthesis techniques based on *success count*, defined as the number of programs for which a technique generates a patch that generalizes/does not overfit. Higher is better.

B. Results

Success Counts, Overlaps, and Union: Figure 5 shows the number of non-overfitting patches generated by Angelix using each of the synthesis engines. Figure 5 shows that Angelix' default synthesis engine can fix more bugs than any other SyGuS techniques. Our observation is that taking into account the original buggy expression to generate minimal fixes gives Angelix an advantage over SyGuS techniques, especially when the buggy expression is large. This suggests that future improvements of SyGuS techniques should make use of the original buggy expression rather than trying to synthesize a fix from scratch. Interestingly, the results also show that Angelix's default synthesis engine alone is not the best for all bugs. For example, there are 7 non-overfitting patches generated by CVC4 that Angelix cannot produce. Overall, combining the results of all synthesis engines together could increase the success by 50%, as compared to Angelix alone. This suggests an interesting angle for future research in semantics-based APR, whereby successfully forging many synthesis techniques would potentially enhance repair capability of APR.

Case studies: We now show case studies where SyGus engines outperform Angelix's default synthesis technique. Figure 3 depicts an example from a `median` program, which attempts to find the median of three integers. A correct patch for this bug is to replace line 4 with a statement at line 6; this is the patch generated by the CVC4 and enumerative SyGuS engines. The "plausible" but overfitting patch generated by Angelix's synthesis engine is depicted at line 5, which replaces the variable *small* with a constant 6. This patch forces a particular set of tests to pass, but will fail on a second set of tests. We believe that this overfitting issue in Angelix's synthesis engine is due to the fact that it does not force generalization during synthesis process, where a generalized solution should involve as small number of constants as possible [5]. SyGuS engines, on the other hand, are more flexible in forcing generalization by simply emphasizing permitted constants after variables in the grammar.

```
1    if (num1 > num2) {...}
2    else {
3      big = num2;
4  -   small = num2;
5  +   small = 6; // by Angelix
6  +   small = num1 // by SyGuS
7    }
```

Fig. 3: Patches generated by Angelix's synthesis engine and SyGuS engines for a median program

Figure 4 shows a defect from a `smallest` program, which returns the smallest of four integer numbers. A correct patch, generated by all four SyGuS techniques, is to replace the incorrect assignment at line 2 with the correct assignment at line 3. Despite the simplicity of the defect, Angelix could not generate any patch for it. The SyGus techniques, with our heuristic, on the other hand, identify which variables should be involved in the synthesis process, reducing the search space for correct solutions.

```
1  else if ((num4 <= num1) && (num4 <= num2) && (num4 <= num3
       )) {
2  -    num_smallest = num1;
3  +    num_smallest = num4; // by all 4 SyGuS techniques
4       printf("%d\n", num_smallest);
5  }
```

Fig. 4: Correct Patch generated SyGuS engines for a smallest program.

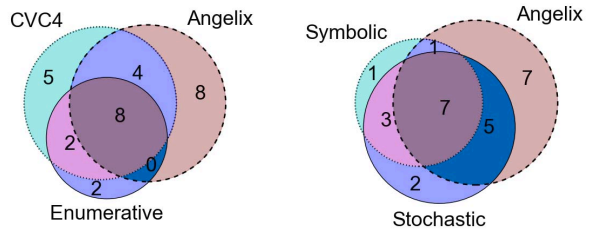

Fig. 5: Non-overfitting patches generated by Angelix, CVC4, Enumerative, Stochastic, and Symbolic synthesis engines.

C. Threats to Validity

Threats to internal validity relate to errors in our implementation and experiments. There could be hidden errors that we did not notice despite our effort on rechecking our implementation and experiments. Threats to external validity correspond to the generalizability of our findings. We have analyzed 188 bugs from 3 different C programs that have been used to evaluate past search-based APR techniques, e.g., [21]. More programs with real bugs would further help mitigate this threat. Threats to construct validity correspond to the suitability of our evaluation metrics. We use success count to assess the synthesis techniques. This is the main metric used in prior studies [21], [19]. There are other criteria that could help in the assessment as well, e.g., time needed for repair. We leave the consideration of other criteria to a future work.

V. Conclusion and Future Work

Semantics-based APR approaches have shown promising results, e.g., by generating high-quality patches for software bugs in large open-source systems. The strengths of such approaches come from the advances of many other fields, especially program synthesis, which plays a crucial role. Given that specifications inferred via test suites are incomplete, program synthesis techniques employed for repair may generate "plausible" but insufficiently general, or correct solutions, depending at least in part on the strategies behind the underlying synthesis. In this paper, we performed the first study on the effectiveness of program synthesis techniques from the program repair point of view. We showed that the existing synthesis technique used for program repair is not best for all cases. Instead, forging the results of many synthesis techniques together can increase the effectiveness of program repair, e.g., generally fixing more bugs, by 50%.

Beyond our empirical results on synthesis techniques, we suggest untapped potential for future research in semantics-based APR, such as in designing ways to forge many synthesis techniques into a more effective approach for APR. We also suggest that APR techniques and benchmarks can serve as a way for assessing synthesis techniques' strengths and weaknesses. Additionally, predicting effectiveness of synthesis techniques to suggest the best technique for APR in particular scenarios, as similarly suggested by [8], would also be an interesting future work. We also plan to strengthen our study by expanding our dataset with more bugs from real-world software. This is possible since our approach is built upon Angelix, which has shown its good scalability [16].

Acknowledgement: This research was funded in part by the Air Force under Contract #FA8750-15-2-0075 and by the US Department of Defense through the Systems Engineering Research Center (SERC), Contract H98230-08-D-0171.

References

[1] Syntax-guided synthesis. [Online]. Available: http://www.sygus.org/

[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION 2007*.

[3] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," *Dependable Software Systems Engineering*, 2015.

[4] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008.

[5] S. Gulwani, J. Esparza, O. Grumberg, and S. Sickert, "Programming by examples (and its applications in data wrangling)," *Verification and Synthesis of Correct and Secure Systems*, 2016.

[6] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE*. IEEE, 2010.

[7] X. B. D. Le, Q. L. Le, D. Lo, and C. L. Goues, "Enhancing automated program repair with deductive verification," in *ICSME*, 2016.

[8] X.-B. D. Le, T.-D. B. Le, and D. Lo, "Should fixing these failures be delegated to automated program repair?" in *ISSRE, 2015*.

[9] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd SANER*. IEEE.

[10] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE TSE*, 2015.

[11] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *TSE*, 2012.

[12] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *FSE*.

[13] ——, "Automatic patch generation by learning correct code," in *ACM SIGPLAN Notices*, 2016.

[14] S. Ma, D. Lo, T. Li, and R. H. Deng, "Cdrep: Automatic repair of cryptographic misuses in android applications," in *ASIACCS*, 2016.

[15] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *ICSE*, 2015.

[16] ——, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*. ACM, 2016.

[17] D. Perelman, S. Gulwani, D. Grossman, and P. Provost, "Test-driven synthesis," in *ACM SIGPLAN Notices*. ACM, 2014.

[18] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," *ACM SIGPLAN Notices*, 2015.

[19] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," ser. ISSTA, 2013.

[20] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett, "Counterexample-guided quantifier instantiation for synthesis in smt," in *CAV*, 2015.

[21] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *FSE*. ACM, 2015.

[22] G. Tassey, "The economic impacts of inadequate infrastructure for software testing." *Planning Report, NIST*, 2002.