Test-Equivalence Analysis for Automatic Patch Generation

SERGEY MECHTAEV, XIANG GAO, SHIN HWEI TAN, and ABHIK ROYCHOUDHURY,
National University of Singapore

Automated program repair is a problem of finding a transformation (called a patch) of a given incorrect program that eliminates the observable failures. It has important applications such as providing debugging aids, automatically grading student assignments, and patching security vulnerabilities. A common challenge faced by existing repair techniques is scalability to large patch spaces, since there are many candidate patches that these techniques explicitly or implicitly consider.

The correctness criteria for program repair is often given as a suite of tests. Current repair techniques do not scale due to the large number of test executions performed by the underlying search algorithms. In this work, we address this problem by introducing a methodology of patch generation based on a test-equivalence relation (if two programs are "test-equivalent" for a given test, they produce indistinguishable results on this test). We propose two test-equivalence relations based on runtime values and dependencies, respectively, and present an algorithm that performs on-the-fly partitioning of patches into test-equivalence classes.

Our experiments on real-world programs reveal that the proposed methodology drastically reduces the number of test executions and therefore provides an order of magnitude efficiency improvement over existing repair techniques, without sacrificing patch quality.

CCS Concepts: • **Software and its engineering → Automatic programming**; **Software testing and debugging**; *Dynamic analysis*;

Additional Key Words and Phrases: Program repair, program synthesis, dynamic program analysis

ACM Reference format:
Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Trans. Softw. Eng. Methodol.* 27, 4, Article 15 (October 2018), 37 pages.
https://doi.org/10.1145/3241980

1 INTRODUCTION

As every developer knows, debugging is difficult and extremely time-consuming. Due to the slow adoption of automated verification and debugging techniques, finding and eliminating defects remains mostly a manual process. Automated patch generation approaches can potentially alleviate this problem since they have been shown to be able to address defects in real-world programs

This work is supported in part by Office of Naval Research grant ONRG-NICOP-N62909-18-1-2052. The authors would also like to thank Airbus Helicopters for the sponsorship of this work as part of Airbus Project Skyways.
Authors' addresses: S. Mechtaev, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417; email: mechtaev@comp.nus.edu.sg; X. Gao, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417; email: gaoxiang@comp.nus.edu.sg; S. H. Tan (corresponding author), Southern University of Science and Technology, No 1088, Xueyuan Rd., Xili, Nanshan District, Shenzhen, Guangdong, China 518055; email: tansh3@sustc.edu.cn; A. Roychoudhury, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417; email: abhik@comp.nus.edu.sg.

1049-331X/2018/10-ART15 $15.00
https://doi.org/10.1145/3241980

and require minimal developer involvement. Specifically, they have been successfully applied for providing debugging hints [39], automatically grading assignments [30, 45], and patching security vulnerabilities [22]. However, the problem of huge search spaces pose serious challenges for current program repair techniques.

The goal of program repair is to modify a given incorrect program to eliminate the observable failures. Specifically, the goal of *test-driven* program repair is to modify the buggy program so that it passes all given tests. The tests are typically provided by developers, but can also be automatically generated in a counterexample-guided refinement loop [1] when a formal specification is available. Patches (program modifications) that pass all given tests are referred to as *plausible* in the program repair literature [28]. Since a test-suite is an incomplete specification, plausible patches may not coincide with user's intentions but may merely *overfit* the tests [32]. To address this problem, state-of-the-art techniques define a cost function (priority) on the space of candidate patches and search for a patch that optimizes this function. For example, changes can be prioritized based on syntactic distance [21], semantic distance [3], and information learned from human patches [17].

Patch generation systems need to consider large spaces of possible modifications in order to address many kinds of defects. One of the key challenges of program repair is scalability to large search spaces. Current techniques may require substantial time to generate patches and yet they consider and generate only relatively simple program transformations [15]. This impacts the ability of program repair to produce human-like repairs, since human patches often involve complex source code modifications. Besides that, a recent study [16] demonstrated that extending the search space with more transformations may cause repair systems to find fewer correct repairs because of the increased search time.

Although existing test-driven program repair techniques employ different methodologies (e.g., GenProg [42] uses genetic programming, SemFix [25] and Angelix [22] are based on constraint solving, Prophet [17] is based on machine learning), they all search for patches by repeatedly executing tests. Due to the high cost of test executions in large real-world programs, the number of performed test executions is the main bottleneck of many existing program repair algorithms.

Existing search methodologies can be divided into two categories: syntax-based and semantics-based. Syntax-based techniques (e.g., GenProg) explicitly generate and test syntactic changes. Thus, the number of test executions performed by such techniques is proportional to the number of explored candidate patches. Semantics-based techniques infer specification for identified statements through path exploration and synthesize changes based on this specification. For instance, Angelix explores deviations of execution paths for given tests using symbolic execution (which can be considered as a variant of test execution), and Prophet explores deviations of execution paths for given tests by enumerating sequences of condition values. Thus, the number of test executions performed by such techniques is proportional to the number of explored paths.

The purpose of this work is to improve the scalability of program repair without sacrificing the quality of generated patches. In order to achieve this, we propose a methodology based on a test-equivalence relation [8, 9]. If two programs are test-equivalent for a test, then the programs produce indistinguishable results on that test.

Definition 1.1 (Test-Equivalence). Let $\mathcal{P}$ be a set of programs, and t be a test. An equivalence relation (reflexive, symmetric, and transitive) $\overset{t}{\sim} \subset \mathcal{P} \times \mathcal{P}$ is a *test-equivalence relation* for t if it is consistent with the results of t, that is, $\forall p_1, p_2 \in \mathcal{P}$, if $p_1 \overset{t}{\sim} p_2$, then p_1 and p_2 either both pass t or both fail t.

The proposed algorithm partitions the space of candidate patches into test-equivalence classes by performing on-the-fly analysis during test execution. This enables our methodology to alleviate the limitations of previous techniques. Compared with syntax-based techniques, it reduces the

number of test executions since a single execution is sufficient to evaluate multiple patch candidates (specifically, all patches in the same test-equivalence class). Compared with semantics-based techniques, it reduces the number of test executions for two reasons. First, it avoids exploration of "infeasible" paths (sequences of values), that is, paths or sequences of values that cannot be induced by any of the considered candidate patches in the context of given tests. Second, it reuses information inferred across multiple tests to skip redundant executions, while previous semantics-based techniques perform path exploration independently for each test.

Contributions. The main contributions of this work are described in the following.

(1) We propose the use of test-equivalence relations to drastically prune the search space explored for the purpose of program repair.
(2) We adapt a test-equivalence relation used in mutation testing [8] (that we call value-based test-equivalence) to program repair by integrating it with syntax-guided program synthesis.
(3) We define a new test-equivalence relation based on dynamic data dependencies and propose a method of synthesizing assignment statements through the composition of value-based and dependency-based test-equivalence relations.
(4) We introduce a new patch space exploration algorithm that performs on-the-fly (during test execution) partitioning of patches into test-equivalence classes, thereby achieving efficient program repair that requires fewer test executions to generate a patch.
(5) We conduct an evaluation of the algorithm on real-world programs from the GenProg ICSE'12 benchmark [12]; it demonstrates that test-equivalence significantly reduces the number of required test executions and therefore increases the efficiency of test-driven program repair and scales it to larger search spaces without sacrificing patch quality.

Outline. In the next section, we provide examples demonstrating limitations of existing techniques and formulate key insights of our methodology. Section 3 formally defines the two test-equivalence relations. Section 4 introduces a repair algorithm based on these relations, Section 5 describes its implementation, and Section 6 presents its experimental evaluation. Section 7 discusses related work, Section 8 discusses future research directions, and Section 9 concludes.

2 MOTIVATING EXAMPLES

This section gives three examples demonstrating limitations of existing techniques: a large number of redundant test executions, ineffectiveness in searching for optimal repairs, and restricted applicability. We also formulate key insights that enable our method to address these limitations.

2.1 Example: Repairing Conditions

Consider a defect in the revision 0661f81 of Libtiff[1] from the GenProg ICSE'12 benchmark. The code in Figure 1(a) is responsible for flushing data written by the compression algorithm, and the defect is caused by the wrong highlighted condition. Libtiff test-suite contains 78 tests, and this defect is manifested by a failing test called "tiffcp-split." Figure 1(b) demonstrates the developer patch that modifies the wrong condition by removing the clause `tif-> tif_rawcc != orig_rawcc`.

We demonstrate how existing automated program repair algorithms generate a patch for this condition. First, repair algorithms perform fault localization to identify suspicious program statements. The number of localized statements in existing tools may vary from tens to thousands

[1]Libtiff is a software library that provides support for TIFF image format: http://simplesystems.org/libtiff/.

```
...
(*tif->tif_close)(tif);
if (tif->tif_rawcc > 0
    && tif->tif_rawcc != orig_rawcc
    && (tif->tif_flags & TIFF_BEENWRITING)!= 0
    && !TIFFFlushData1(tif)) {
      TIFFErrorExt(tif->tif_clientdata,
        module,
        "Error flushing data before
            directory write");
    return (0);
  }
...
```

(a) Incorrect condition in Libtiff (rev. 0661f81).

```
...
(*tif->tif_close)(tif);
if (tif->tif_rawcc > 0
    && (tif->tif_flags & TIFF_BEENWRITING)!= 0
    && !TIFFFlushData1(tif)) {
      TIFFErrorExt(tif->tif_clientdata,
        module,
        "Error flushing data before
            directory write");
    return (0);
  }
...
```

(b) Developer patch for incorrect condition.

Fig. 1. Defect in Libtiff library from GenProg ICSE'12 benchmark.

depending on algorithms and configurations (it can potentially include all executed statements). In this example, we consider only the location of the buggy expression highlighted in Figure 1(a).

Second, program repair algorithms define a *search space* of candidate patches. In this work, we primarily focus on two state-of-the-art approaches that have been shown to scale to large real-world programs: Angelix [22] and Prophet [17]. Specifically, our goal was to support a combination of transformations implemented in these systems. Thus, the search space for the highlighted condition includes all possible replacements of its subexpressions by expressions constructed from visible program variables and C operators, refinements (e.g., appending `&& EXPR` and `|| EXPR`), replacements of operators, and swapping arguments. In total, the search space in our synthesizer contains 56,243 modifications of the buggy condition.

Finally, program repair algorithms explore the search space in order to try to find a modification that passes all given tests. We say that an element of a search space is *explored* if the algorithm identifies if it passes all the tests or fails at least one. Existing search space exploration methods can be classified into two categories: syntax-based and semantics-based. *Syntax-based* algorithms explicitly generate and test syntactic changes. In this example, a syntax-based algorithm have to execute the failing test 56,243 times to evaluate all candidates.[2] Since there are 78 tests in the test-suite, 907,457 test executions are required to explore the search space.[3] Given the high cost of test execution, this approach has poor scalability.

Semantics-based techniques (e.g., Semfix [25], SPR [15], Angelix and Prophet) split exploration into two phases. First, they infer a synthesis specification for the identified expression through path exploration. For this example, they enumerate and execute sequences of condition values (e.g., *true*, *true*, *true*, *false*, . . .) to find those sequences that enable the program to pass the test. Second, they synthesize a modification of the condition to match the inferred specification. In this example, there are 256 possible execution paths (the condition is evaluated multiple times during the test execution), therefore a semantics-based algorithm performs 256 test executions for the failing tests, and 1,320 for the whole test-suite.[4] Although semantics-based techniques were shown to be more scalable [15], they are subject to the *path explosion problem*: the number

[2]Since the search space contains the correct patch in this example, the algorithm can stop search earlier after the patch is found. Then, the number of test executions depends on the exploration order.
[3]This data is obtained by executing our implementation of syntactic enumeration.
[4]This data is obtained by executing Angelix.

```
1. ((tif->tif_rawcc > 0) && (tif->tif_rawcc != orig_rawcc))
    || (tif->tif_flags & TIFF_BEENWRITING)
2. ((tif->tif_rawcc > 0) || (tif->tif_rawcc != orig_rawcc))
    && (tif->tif_flags & TIFF_BEENWRITING)
3. ((tif->tif_rawcc == 0) && (tif->tif_rawcc != orig_rawcc))
    && (tif->tif_flags & TIFF_BEENWRITING)
4. (((tif->tif_rawcc > 0) && (tif->tif_rawcc != orig_rawcc))
    && (tif->tif_flags & TIFF_BEENWRITING)) || (imagedone >= orig_rawcc)
5. (((tif->tif_rawcc > 0) && (tif->tif_rawcc != orig_rawcc))
    && (tif->tif_flags & TIFF_BEENWRITING)) || (tif->tif_flags >= 74)
```

Fig. 2. Each of 56,243 search space elements is test-equivalent to one of these five expressions.

of execution paths can be infinite. To address this, current systems introduce a bound for the number of explored paths; however, it may affect their effectiveness: if a path followed by the correct patch is omitted, then this correct patch cannot be generated.

The algorithm proposed in this work performs on-the-fly partitioning of program modifications into test-equivalence classes. We demonstrate the effect of the relation $\overset{t}{\sim}_{value}$ described in Section 3.3. Two modifications of a program expression are test-equivalent w.r.t. $\overset{t}{\sim}_{value}$ if they are evaluated into the same sequences of values during the test execution. Surprisingly, the space of 56,243 modifications can be partitioning into only five test-equivalence classes for the failing test "tiffcp-split" w.r.t. $\overset{t}{\sim}_{value}$; five elements of the search space that represent different test-equivalence classes are given in Figure 2. Since all patches in the same test-equivalence class exhibit the same behavior for the corresponding test, the failing test can be executed only five times to evaluate all candidates.

Our algorithm computes test-equivalence classes for each test in the test-suite. However, since test-equivalence classes for different tests may intersect, our algorithm takes advantage of this to skip redundant execution across different tests. Specifically, for each next test it only evaluates subspaces of modifications that are not included into failing test-equivalence classes of previously executed tests. Meanwhile, semantics-based techniques perform specification inference for each test independently without reusing information across tests. As a result, our algorithm requires only 103 test executions to evaluate all 56,243 modifications with the whole test-suite.

Key Insight. The key insight that enables our method to reduce the number of required test executions is that, compared with techniques that explore execution paths, it takes the expressiveness of the patch space into account (e.g., it identifies that only 5 out of 256 possible execution paths are induced by the considered set of 56,243 transformations). Compared with syntactic enumeration, it substantially reduces executions since a single execution evaluates a whole test-equivalence class.

2.2 Example: Optimal Repair

Since a test-suite is an incomplete specification, test-driven program repair suffers from the test overfitting problem [32]. To address this issue, state-of-the-art techniques define a priority (a cost function) in the space of patches and search for a program modification that optimizes this function. Ideally, this function should assign higher cost to overfitting patches. For instance, Prophet [17] demonstrates how such a cost function learned from human patches enables the generation of more correct repairs.

Consider a program p in Figure 3(a) that counts odd numbers in the interval $(0, i]$. The $*$ indicates a wrong condition that has to be modified by the repair algorithm (the correct condition is `i mod 2 = 1`). We denote a program obtained by substituting $*$ with an expression e as $p[*/e]$. The

```
while i > 0 do
  if * then
    c := c + 1
  fi;
  i := i - 1
od
```

(a) Buggy program p.

$$\mathcal{P} := \{\ p[*/\texttt{i} \geq \texttt{0}],\ p[*/\texttt{c} \geq \texttt{0}],\ p[*/\texttt{i mod 2 = 1}],\ p[*/\texttt{i mod 2 = 0}],\ p[*/\texttt{i > 2}]\ \}$$

(b) Search space.

$$\kappa(p[*/\texttt{i} \geq \texttt{0}]) := 0.1$$
$$\kappa(p[*/\texttt{c} \geq \texttt{0}]) := 0.2$$
$$\kappa(p[*/\texttt{i mod 2 = 1}]) := 0.3$$
$$\kappa(p[*/\texttt{i mod 2 = 0}]) := 0.4$$
$$\kappa(p[*/\texttt{i > 2}]) := 0.5$$

(c) Cost function.

Fig. 3. Example of optimal program repair problem.

repair algorithm searches for a plausible patch (a substitution of $*$ with a condition) from the space $\mathcal{P}$ in Figure 3(b) such that the resulting program passes the test t defined as follows:

$$t := (\{\ i \mapsto 4,\ c \mapsto 0\ \},\ \lambda\sigma.\ \sigma(c) = 2),$$

where t is pair of (1) an initial program state (mapping from variables to values) and (2) a test assertion (a Boolean function over program states) denoted using lambda notation. We assume that $*$ is such that p fails t. Besides that, we consider a cost function κ defined for the considered space of substitutions in Figure 3(c). The goal is to find a plausible patch with the lowest cost.

In order to find a patch for the example program, techniques like Angelix and Prophet enumerate possible sequences of values that a condition can take during test execution. Since there can be potentially an infinite number of such sequences, existing approaches introduce a bound for the number of explored sequences and use an exploration heuristics to choose which sequences to explore. For instance, Prophet enumerates sequences where the condition first always takes the true branch until a certain point after which it always takes the false branch. Thus, for the considered example it would enumerate the following sequences:

$$\{\ \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}\ \},$$
$$\{\ \mathit{true}, \mathit{true}, \mathit{true}, \mathit{false}\ \},$$
$$\{\ \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}\ \},$$
$$\{\ \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}\ \}.$$

For each of these sequences, Prophet executes the program with the test t in such a way that the condition $*$ takes the values as in this sequence during the execution. Only the third sequence $\{\ \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}\ \}$ enables the program to pass t, therefore it will be selected as a specification for expression synthesis. The synthesizer will find the expression $\texttt{i} > 2$ obtaining a suboptimal patch $p[*/\texttt{i} > 2]$ with the cost 0.5, since this is the only expression from the search space satisfying the specification. However, the correct expression $\texttt{i mod 2} = 1$ with a lower cost 0.3 cannot be generated, since the corresponding sequence $\{\ \mathit{false}, \mathit{true}, \mathit{false}, \mathit{true}\ \}$ is not explored by the algorithm.

In contrast to techniques like Angelix and Prophet, our algorithm iterates through the search space in such a way that at each steps it selects and evaluates an unevaluated candidate with the lowest cost. Specifically, it starts by choosing the candidate $p[*/\texttt{i} \geq \texttt{0}]$ with the cost 0.1. It executes this candidate on-the-fly computing its test-equivalence class w.r.t. $\overset{t}{\sim}_{value}$ described in Section 2.1. This class contains the program $p[*/\texttt{c} \geq \texttt{0}]$, since the conditions $\texttt{i} \geq \texttt{0}$ and $\texttt{c} \geq \texttt{0}$ produce the same sequence of values $\{\ \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}\ \}$ for t. Since $p[*/\texttt{i} \geq \texttt{0}]$ does not pass the test, the whole corresponding test-equivalence class is marked as failing. Next, it selects $p[*/\texttt{i mod 2} = \texttt{1}]$ with the cost 0.3 since $p[*/\texttt{c} \geq \texttt{0}]$ was indirectly evaluated through test-equivalence at the previous step. Since this candidate passes the test, the algorithm outputs it as a found repair.

```
...
clear_bufs();
to_stdout = 1;
part_nb = 0;
ifd = part_nb;
if (decompress) {
  method=get_method(ifd);
...
```

(a) Before if-statement.

```
...
clear_bufs();
to_stdout = 1;
part_nb = 0;
ifd = 0;
if (decompress) {
  method=get_method(ifd);
...
```

(b) Before if-statement.

```
...
clear_bufs();
to_stdout = 1;
part_nb = 0;
if (decompress) {
  ifd = part_nb;
  method=get_method(ifd);
...
```

(c) Inside if-statement.

Fig. 4. Candidate patches for defect of Gzip from GenProg ICSE'12 benchmark.

Key Insight. Our algorithm guides exploration based on a given cost function and focuses on high priority areas of the space of patches. By construction, if it finds a patch, then this patch is guaranteed to be the global optimum in the search space w.r.t. the cost function. Angelix and Prophet, on the other hand, may spend executions for value sequences that correspond to suboptimal candidates or correspond to no candidates at all (e.g., { *false*, *true*, *true*, *true* }), and therefore may miss the best patch in their search space.

2.3 Example: Repairing Assignments

Although current program repair approaches have been shown to be relatively effective in modifying existing program expressions, they provide limited support for more complex transformations. In this work, we consider one such transformation that inserts a new assignment statement to the buggy program. Techniques like Prophet and GenProg can generate patches by copying/moving existing program assignments; however, this approach has limitations: (1) assignments for local variables cannot be copied from different parts of the program because of their scope and (2) each insertion of an assignment is validated separately, which yield a large number of required test executions. Existing techniques do not apply specification inference for assignment synthesis (as described in Section 2.1 for conditions) because such specification has to encode all possible side effects that can be caused by assignment insertion (for each variable that can appear in the left-hand side of the assignment), which makes inferring such specification infeasible for large programs.

We show how test-equivalence can scale assignment synthesis for a defect in Gzip[5] from the GenProg ICSE'12 benchmark. Consider three candidate patches in Figure 4 that insert the highlighted statements at several program locations. First, our algorithm identifies that the program in Figure 4(a) is test-equivalent to the program in Figure 4(b) (w.r.t. the relation $\overset{t}{\sim}_{value}$ described previously in Section 2.1) since they differ only in the right-hand side of the highlighted assignments and the corresponding expressions take the same values during test execution. Second, using a simple dynamic data dependency analysis, our algorithm identifies that the program in Figure 4(a) is test-equivalent to the program in Figure 4(c) since (1) they insert the same assignment at different program locations, (2) both these locations are executed by the test since the true branch of the if-statement is taken during the test execution, and (3) the variables `ifd` and `part_nb` are not used/modified between these locations during test execution. We refer to such a test-equivalence relation as $\overset{t}{\sim}_{deps}$. Finally, our algorithm merges the results of the two analyses (as the transitive closure of their union) and determines that the program in Figure 4(b) is test-equivalent to the program in Figure 4(c). Therefore, a single test execution is sufficient to evaluate all these patches.

[5]Gzip is a file compression/decompression application: https://www.gnu.org/software/gzip/.

$$
\begin{array}{l}
\langle Stmt\rangle ::= \langle Var\rangle := \langle AExpr\rangle \\
\quad \mid \texttt{skip} \\
\quad \mid \langle Stmt\rangle \ ; \ \langle Stmt\rangle \\
\quad \mid \texttt{if}\ \langle BExpr\rangle\ \texttt{then}\ \langle Stmt\rangle\ \texttt{else}\ \langle Stmt\rangle\ \texttt{fi} \\
\quad \mid \texttt{while}\ \langle BExpr\rangle\ \texttt{do}\ \langle Stmt\rangle\ \texttt{od}
\end{array}
$$

$$
\begin{array}{l}
\langle AExpr\rangle ::= \langle Var\rangle \\
\quad \mid \langle Num\rangle \\
\quad \mid \langle AExpr\rangle\ \langle AOp\rangle\ \langle AExpr\rangle
\end{array}
$$

$$
\begin{array}{l}
\langle BExpr\rangle ::= \texttt{true} \\
\quad \mid \texttt{false} \\
\quad \mid \langle BExpr\rangle\ \langle BOp\rangle\ \langle BExpr\rangle \\
\quad \mid \langle AExpr\rangle\ \langle ROp\rangle\ \langle AExpr\rangle
\end{array}
$$

$$\langle AOp\rangle ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \ldots$$

$$\langle BOp\rangle ::= \texttt{and} \mid \texttt{or} \mid \ldots$$

$$\langle ROp\rangle ::= \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \ldots$$

Fig. 5. Syntax of programming language $\mathcal{L}$.

$$\text{VAR}\ \frac{}{\langle v,\sigma\rangle \Downarrow \sigma(v)} \qquad \text{NUM}\ \frac{}{\langle n,\sigma\rangle \Downarrow n}$$

$$\text{OP}\ \frac{\langle e_1,\sigma\rangle \Downarrow n_1 \qquad \langle e_2,\sigma\rangle \Downarrow n_2 \qquad n_3 = n_1\ op\ n_2}{\langle e_1\ op\ e_2,\sigma\rangle \Downarrow n_3} \qquad \text{ASSIGN}\ \frac{\langle e,\sigma\rangle \Downarrow n}{\langle v := e,\sigma\rangle \Downarrow \sigma[v \mapsto n]}$$

$$\text{SEQ}\ \frac{\langle s_1,\sigma\rangle \Downarrow \sigma_1 \qquad \langle s_2,\sigma_1\rangle \Downarrow \sigma_2}{\langle s_1\ ;\ s_2,\sigma\rangle \Downarrow \sigma_2}$$

$$\text{IF-TRUE}\ \frac{\langle e,\sigma\rangle \Downarrow true \qquad \langle s_1,\sigma\rangle \Downarrow \sigma_1}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi},\sigma\rangle \Downarrow \sigma_1} \qquad \text{IF-FALSE}\ \frac{\langle e,\sigma\rangle \Downarrow false \qquad \langle s_2,\sigma\rangle \Downarrow \sigma_2}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi},\sigma\rangle \Downarrow \sigma_2}$$

$$\text{WHILE-TRUE}\ \frac{\langle e,\sigma\rangle \Downarrow true \qquad \langle s_1,\sigma\rangle \Downarrow \sigma_1 \qquad \langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma_1\rangle \Downarrow \sigma_2}{\langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma\rangle \Downarrow \sigma_2} \qquad \text{WHILE-FALSE}\ \frac{\langle e,\sigma\rangle \Downarrow false}{\langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma\rangle \Downarrow \sigma} \qquad \text{SKIP}\ \frac{}{\langle \texttt{skip},\sigma\rangle \Downarrow \sigma}$$

Fig. 6. Semantics of $\mathcal{L}$. v—variables, n—integer values, e—expressions, s—statements, and σ—program states.

Key Insight. Since test-equivalence is a weaker property than the property of "passing the test" expressed by the inferred specification in semantics-based techniques, it permits using more lightweight analysis techniques. Specifically, we demonstrate that a composition of two lightweight test-equivalence analyses enables us to scale assignment synthesis.

3 TEST-EQUIVALENCE RELATIONS

This section formally introduces two test-equivalence relations for spaces of program modifications generated through program synthesis. In the subsequent Section 4, we demonstrate how these relations can be applied for scaling patch generation. However, we believe that these relations can be also used in different domains; other potential applications are discussed in Section 7.

3.1 Preliminaries

We introduce our methodology for an imperative programming language $\mathcal{L}$. The syntax of $\mathcal{L}$ is defined in Figure 5, where $\mathbb{Z}$ is the integer domain, $\mathbb{B}$ is the Boolean domain (true and false), *Stmt* is a set of statements, *AExpr* is a set of arithmetic expressions, *BExpr* is a set of Boolean expressions, $Expr = AExpr \cup BExpr$, *Num* is a set of integer literals, and *Var* is a set of variables over $\mathbb{Z}$. A program in $\mathcal{L}$ is a sequence of statements. We denote subsets of $\mathcal{L}$ as $\mathcal{P}$, subsets of expressions *Expr* as $\mathcal{E}$, all variables from *Var* encountered in an expression e as $Var(e)$, a program that is obtained by substituting a statement (expression) s with a statement (expression) s' in a program p as $p[s/s']$.

The semantics of $\mathcal{L}$ is defined in Figure 6, where program state $\sigma : Var \to \mathbb{Z}$ is a function from program variables into values, Σ is a set of program states. We indicate a modification of a program state σ where the value of the variable v is updated to n as $\sigma[v \mapsto n]$.

Definition 3.1 (Test). Let $p \in \mathcal{L}$ be a program, $t \in \Sigma \times (\Sigma \to \mathbb{B})$ be a test, that is, a pair (σ_{in}, ϕ), where σ_{in} in the initial program state (input) and ϕ is the test assertion (a Boolean function over program states). We say that p passes t (indicated as $Pass[p,t]$) iff $\langle p, \sigma_{in}\rangle \Downarrow \sigma_{out} \wedge \phi(\sigma_{out})$.

A test execution can be represented through a derivation tree as in the following example.

Example 3.2 (Derivation Tree). Consider a program p defined as "$x := y + 1; y := x$" and a test $t := (\sigma_{in}, \lambda\sigma.\ \sigma(y) = 3)$ where the input state $\sigma_{in} := \{x \mapsto 1, y \mapsto 2\}$. According to the semantics in Figure 6, the following relation holds: $\langle p, \sigma_{in}\rangle \Downarrow \sigma_{out}$, where $\sigma_{out} := \{x \mapsto 3, y \mapsto 3\}$. This relation can be established by the following *derivation tree* obtained by applying the semantics rules:

$$\dfrac{\dfrac{\dfrac{}{\langle y, \ldots\rangle \Downarrow 2}\text{VAR} \quad \dfrac{}{\langle 1, \ldots\rangle \Downarrow 1}\text{NUM}}{\langle y + 1, \{x \mapsto 1, y \mapsto 2\}\rangle \Downarrow 3}\text{OP}}{\langle x := y + 1, \{x \mapsto 1, y \mapsto 2\}\rangle \Downarrow \{x \mapsto 3, y \mapsto 2\}}\text{ASSIGN} \quad \dfrac{\dfrac{}{\langle x, \ldots\rangle \Downarrow 3}\text{VAR}}{\langle y := x, \{x \mapsto 3, y \mapsto 2\}\rangle \Downarrow \{x \mapsto 3, y \mapsto 3\}}\text{ASSIGN}$$
$$\langle x := y + 1; y := x, \{x \mapsto 1, y \mapsto 2\}\rangle \Downarrow \{x \mapsto 3, y \mapsto 3\}\ \text{SEQ}$$

The output state $\{x \mapsto 3, y \mapsto 3\}$ satisfies the test assertion $\lambda\sigma.\ \sigma(y) = 3$, therefore p passes t.

3.2 Generalized Synthesis

The presented methodology of test-equivalence analysis is introduced for program repair techniques that rely on syntax-guided program synthesis to generate patches. Such techniques generate expressions for "holes" in programs based on the values of visible program variables (program state) and expected results for the synthesized expressions. Thus, we define *synthesis specification* as a finite set of input-output pairs (an input is represented by a program state from Σ and an output is an integer or a Boolean value); we denote the set of all specifications as $Spec := 2^{\Sigma\times(\mathbb{Z}\cup\mathbb{B})}$.

Definition 3.3 (Synthesis Procedure). A syntax-guided synthesis procedure $synthesize : 2^{Expr} \times Spec \to Expr$ is a function that takes a set of expressions (the synthesis search space) and a specification, and returns an expression from the search space that meets the specification. Specifically, for a given search space $\mathcal{E}$ and specification $spec$, if $synthesize(\mathcal{E}, spec) = e$, then $e \in \mathcal{E} \wedge \bigwedge_{\sigma,n\in spec}\langle e, \sigma\rangle \Downarrow n$.

In order to integrate synthesis with test-equivalence analysis, we impose additional requirements for the program synthesizer: it should define a value-projection operator over its search space.

The *value-projection operator* $\Pi^{value}_{\sigma,n}$ produces a maximal subset of a given set of expressions consisting only of expressions that are evaluated into n in the context σ:

$$\Pi^{value}_{\sigma,n}(\mathcal{E}) = \{e \mid e \in \mathcal{E} \wedge \langle e, \sigma\rangle \Downarrow n\}.$$

In this work, we use an enumerative synthesizer [1] that demonstrated positive results in program synthesis competitions.[6] Since it represents the search space explicitly as a set of expressions, it is straightforward to realize the value-projection operator in such a synthesizer (Algorithm 1). Other possible realizations are discussed in Section A.

3.3 Value-Based Test-Equivalence Relation

This section introduces a test-equivalence relation $\overset{t}{\sim}_{value}$ for spaces of programs that differ only in expressions. From the conceptual point of view, the introduced relation generalizes relations that have been used in mutation testing [8]. From the algorithmic point of view, our approach differs from the previous work on test-equivalence in that the relation is integrated with syntax-guided program synthesis (used for patch synthesis) via the value-projector operator.

[6] SyGuS-Comp 2014: http://www.sygus.org/SyGuS-COMP2014.html.

$$\text{EXPR-MOD}\ \frac{\mathit{Modified}(e) \qquad \langle e,\sigma\rangle \Downarrow n \qquad c' = \Pi^{value}_{\sigma,n}(c)}{\langle e,\sigma,c\rangle \Downarrow_{value} \langle n,c'\rangle} \qquad \text{EXPR-NMOD}\ \frac{\neg\mathit{Modified}(e) \qquad \langle e,\sigma\rangle \Downarrow n}{\langle e,\sigma,c\rangle \Downarrow_{value} \langle n,c\rangle} \qquad \text{ASSIGN}\ \frac{\langle e,\sigma,c\rangle \Downarrow_{value} \langle n,c'\rangle}{\langle v := e,\sigma,c\rangle \Downarrow_{value} \langle \sigma[v \mapsto n],c'\rangle}$$

$$\text{IF-TRUE}\ \frac{\langle e,\sigma,c\rangle \Downarrow_{value} \langle \mathit{true},c_1\rangle \qquad \langle s_1,\sigma,c_1\rangle \Downarrow_{value} \langle \sigma_1,c_2\rangle}{\langle \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi},\sigma,c\rangle \Downarrow_{value} \langle \sigma_1,c_2\rangle} \qquad \text{IF-FALSE}\ \frac{\langle e,\sigma,c\rangle \Downarrow_{value} \langle \mathit{false},c_1\rangle \qquad \langle s_2,\sigma,c_1\rangle \Downarrow_{value} \langle \sigma_2,c_2\rangle}{\langle \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi},\sigma,c\rangle \Downarrow_{value} \langle \sigma_2,c_2\rangle}$$

$$\text{SEQ}\ \frac{\langle s_1,\sigma,c\rangle \Downarrow_{value} \langle \sigma_1,c_1\rangle \qquad \langle s_2,\sigma_1,c_1\rangle \Downarrow_{value} \langle \sigma_2,c_2\rangle}{\langle s_1\ ;\ s_2,\sigma,c\rangle \Downarrow_{value} \langle \sigma_2,c_2\rangle} \qquad \text{SKIP}\ \frac{}{\langle \texttt{skip},\sigma,c\rangle \Downarrow_{value} \langle \sigma,c\rangle} \qquad \text{WHILE-FALSE}\ \frac{\langle e,\sigma,c\rangle \Downarrow_{value} \langle \mathit{false},c'\rangle}{\langle \texttt{while } e \texttt{ do } s \texttt{ od},\sigma,c\rangle \Downarrow_{value} \langle \sigma,c'\rangle}$$

$$\text{WHILE-TRUE}\ \frac{\langle e,\sigma,c\rangle \Downarrow_{value} \langle \mathit{true},c_1\rangle \qquad \langle s_1,\sigma,c_1\rangle \Downarrow_{value} \langle \sigma_1,c_2\rangle \qquad \langle \texttt{while } e \texttt{ do } s \texttt{ od},\sigma_1,c_2\rangle \Downarrow_{value} \langle \sigma_2,c_3\rangle}{\langle \texttt{while } e \texttt{ do } s \texttt{ od},\sigma,c\rangle \Downarrow_{value} \langle \sigma_2,c_3\rangle}$$

Fig. 7. Augmented semantics of $\mathcal{L}$ for computing test-equivalence classes w.r.t. $\overset{t}{\sim}_{value}$. v—variables, n—integer values, e—expressions, s—statements, σ—program states, c—sets of expressions, and *Modified*—predicate over expressions.

ALGORITHM 1: Value-projection operator via enumerative synthesis

Input: set of expressions $\mathcal{E}$, program state σ, value n
Output: set of expressions $\mathcal{E}'$

```
1 E' := ∅;
2 foreach e ∈ E do
3     if ⟨e, σ⟩ ⇓ n then
4         E' := E' ∪ {e};
5 return E';
```

Intuitively, two programs p and p' such that $p' = p[e/e']$ for some expressions e and e' are test-equivalent for some test t w.r.t. $\overset{t}{\sim}_{value}$ if, during the executions of p and p' with t, the expressions e and e' are evaluated into the same values. An example of applying this relation is given in Section 2.1.

We define the relation $\overset{t}{\sim}_{value}$ *constructively* using an augmented semantics of $\mathcal{L}$. We chose this presentation since it simultaneously defines an algorithm of computing test-equivalence classes in spaces of modification generation through program synthesis. The implementation of this semantics via program instrumentation is discussed in Section 5.

The semantics in Figure 7 extends the semantics in Figure 6 by defining the function $\Downarrow_{value}$. It is parameterized by a predicate $\mathit{Modified} : \mathit{Expr} \to \mathbb{B}$ that marks the modified program expression, substitutions of which are analyzed for test-equivalence. The function $\Downarrow_{value}$ additionally maintains a set of expressions (denoted as c), such that the substitutions of the modified expression with c form the computed test-equivalent class.

The augmented semantics describes an algorithm of identifying test-equivalence classes that, for a given set of expressions (the synthesis search space), "filters out" those that do not belong to the test-equivalent class of the current program by repeatedly applying the value-projection operator. The application of the value-projection operator to the current set of expressions c is highlighted in Figure 7. Thus, it identifies all expressions from c that produce the same value as the original expression at this evaluation step. Since each expression can be evaluated multiple times during test execution, the value-projection operator can also be applied multiple times. Therefore, the test-equivalence class is computed as $\Pi^{value}_{\sigma_1,n_1} \circ \Pi^{value}_{\sigma_2,n_2} \circ \ldots \circ \Pi^{value}_{\sigma_k,n_k}(\mathcal{E})$, where $\mathcal{E}$ is a set of all substitutions of the modified expression, n_i are the values of the modified expression computed during test execution, and σ_i are the corresponding program states.

Definition 3.4 (Value-Based Test-Equivalence Relation). Let $\mathcal{P}$ be a set of programs, $t = (\sigma_{in}, \phi)$ be a test. $\overset{t}{\sim}_{value} \subset \mathcal{P} \times \mathcal{P}$ is a *value-based test-equivalence relation* iff $p_1 \overset{t}{\sim}_{value} p_2$ for $p_1, p_2 \in \mathcal{P}$ if $\exists e, e' \in Expr$ such that $p_2 = p_1[e/e']$ and $\langle p_1, \sigma_{in}, \{e, e'\}\rangle \Downarrow_{value} \langle \_, \{e, e'\}\rangle$ given that $Modified := \lambda x.\ x = e$.

In this definition, we call two programs that differ only in expressions to be test-equivalent if the corresponding expressions produce the same values according to the semantics in Figure 6. Specifically, by passing the program p_1, the test input σ_{in} and the set of expressions $\{e, e'\}$ as the arguments to $\Downarrow_{value}$, we obtain the same set $\{e, e'\}$ as the result.

Proposition 3.5. *The relation $\overset{t}{\sim}_{value}$ is a test-equivalence relation according to Definition 1.1.*

The proposition above formally states that (1) $\overset{t}{\sim}_{value}$ is an equivalence relation and (2) if two programs that differ only in expressions are test-equivalent according to the semantics in Figure 7, then these two programs either both pass the test or both fail the test. A proof for the above proposition is given in Section C.

Example 3.6 (Test-Equivalent Programs w.r.t. $\overset{t}{\sim}_{value}$). Consider a program p_1 defined as

$$\texttt{if } x > 0 \texttt{ then } x := y \texttt{ else skip fi},$$

a program p_2 defined as

$$\texttt{if } y = 2 \texttt{ then } x := y \texttt{ else skip fi},$$

and a test $t := (\sigma_{in}, \lambda\sigma.\ \sigma(y) = 3)$ where the input state $\sigma_{in} := \{x \mapsto 1, y \mapsto 2\}$. These programs are test-equivalent w.r.t. the value-based test-equivalence relation for the test t, since they differ only in the if-condition and the following relation holds:

$$\langle p_1, \sigma_{in}, \{\text{“}x > 0\text{”}, \text{“}y = 2\text{”}\}\rangle \Downarrow_{value} \langle \sigma_{out}, \{\text{“}x > 0\text{”}, \text{“}y = 2\text{”}\}\rangle,$$

where $\sigma_{out} := \{x \mapsto 2, y \mapsto 2\}$. This relation can be established by the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dots}{\langle x > 1, \sigma_{in}\rangle \Downarrow true}\text{OP} \quad \{\text{“}x > 0\text{”}, \text{“}y = 2\text{”}\} = \Pi^{value}_{\sigma_{in}, true}(\{\text{“}x > 0\text{”}, \text{“}y = 2\text{”}\})}{\langle x > 0, \sigma_{in}, \{\text{“}x > 0\text{”}, \text{“}y = 2\text{”}\}\rangle \Downarrow_{value} \langle true, \{\text{“}x > 0\text{”}, \text{“}y = 2\text{”}\}\rangle}\text{EXPR-MOD} \quad \dfrac{\dots}{\dots}\text{ASSIGN}}{\langle \texttt{if } x > 0 \texttt{ then } x := y \texttt{ else skip fi}, \sigma_{in}, \{\text{“}x > 0\text{”}, \text{“}y = 2\text{”}\}\rangle \Downarrow_{value} \langle \sigma_{out}, \{\text{“}x > 0\text{”}, \text{“}y = 2\text{”}\}\rangle}\text{IF-TRUE}$$

3.4 Dependency-Based Test-Equivalence Relation

This section introduces a test-equivalence relation $\overset{t}{\sim}_{deps}$ for spaces of programs that differ in locations in which an assignment statement is inserted. Let a *location* in program p be a statement of p. We say that a program p' is obtained by inserting the assignment $v := e$ at the location l iff $p' = p[l/\text{v} := \text{e};l]$. Let p be a program and programs p_1 and p_2 are obtained by inserting the assignment $v := e$ at the locations l_1 and l_2 of p, respectively. Informally, p_1 and p_2 are test-equivalent for some test t if, during an execution of p_1 with t, (1) for each occurrence of l_1 in the execution trace there is a “matching” occurrence of l_2 (the variable v is not read or overwritten between these occurrences and the variables $Var(e)$ are not overwritten between these occurrences), and (2) for each occurrence of l_2 in the execution trace there is a “matching” occurrence of l_1. An example of applying this relation is given in Section 2.3.

The relation is formally defined through an augmented semantics of $\mathcal{L}$. As in Section 3.3, we chose this representation since it simultaneously defines an algorithm of identifying test-equivalence classes. The implementation of this semantics via program instrumentation is discussed in Section 5.

$$\text{VAR-LEFT-EXE}\ \frac{\mathit{Left}(v) \wedge x}{\langle v,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma(v), l \cap c, \emptyset, \mathit{false}\rangle} \qquad \text{VAR-LEFT-NEXE}\ \frac{\mathit{Left}(v) \wedge \neg x}{\langle v,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma(v), l \setminus c, \emptyset, x\rangle} \qquad \text{VAR-NLEFT}\ \frac{\neg \mathit{Left}(v)}{\langle v,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma(v), l, c, x\rangle}$$

$$\text{OP}\ \frac{\langle e_1,\sigma,l,c,x\rangle \Downarrow_{deps} \langle n_1,l_1,c_1,x_1\rangle \qquad \langle e_2,\sigma,l_1,c_1,x_1\rangle \Downarrow_{deps} \langle n_2,l_2,c_2,x_2\rangle \qquad n_3 = n_1\ op\ n_2}{\langle e_1\ op\ e_2,\sigma,l,c,x\rangle \Downarrow_{deps} \langle n_3,l_2,c_2,x_2\rangle}$$

$$\text{ASSIGN-INS}\ \frac{\mathit{Inserted}(v := e) \qquad \langle v := e,\sigma\rangle \Downarrow \sigma'}{\langle v := e,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma', l, c, \mathit{true}\rangle} \qquad \text{ASSIGN-LR-EXE}\ \frac{\neg \mathit{Inserted}(v := e) \wedge (\mathit{Left}(v) \vee \mathit{Right}(v)) \wedge x \qquad \langle v := e,\sigma\rangle \Downarrow \sigma'}{\langle v := e,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma', l \cap c, \emptyset, \mathit{false}\rangle}$$

$$\text{ASSIGN-LR-NEXE}\ \frac{\neg \mathit{Inserted}(v := e) \wedge (\mathit{Left}(v) \vee \mathit{Right}(v)) \wedge \neg x \qquad \langle v := e,\sigma\rangle \Downarrow \sigma'}{\langle v := e,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma', l \setminus c, \emptyset, \mathit{false}\rangle}$$

$$\text{ASSIGN-NLR}\ \frac{\neg \mathit{Inserted}(v := e) \wedge \neg \mathit{Left}(v) \wedge \neg \mathit{Right}(v) \qquad \langle e,\sigma,l,c,x\rangle \Downarrow_{deps} \langle n,l_1,c_1,x_1\rangle}{\langle v := e,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma[v \mapsto n], l_1, c_1, x_1\rangle}$$

$$\text{SEQ}\ \frac{\langle s_1,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma_1,l_1,c_1,x_1\rangle \qquad \langle s_2,\sigma_1,l_1,c_1 \cup \{s_2\},x_1\rangle \Downarrow_{deps} \langle \sigma_2,l_2,c_2,x_2\rangle}{\langle s_1\ ;\ s_2,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma_2,l_2,c_2,x_2\rangle} \qquad \text{SKIP}\ \frac{}{\langle \texttt{skip},\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma,l,c,x\rangle}$$

$$\text{IF-TRUE}\ \frac{\langle e,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \mathit{true},l_1,c_1,x_1\rangle \qquad \langle s_1,\sigma,l_1,c_1 \cup \{s_1\},x_1\rangle \Downarrow_{deps} \langle \sigma_1,l_2,c_2,x_2\rangle}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi},\sigma,c\rangle \Downarrow_{deps} \langle \sigma_1,l_2,c_2,x_2\rangle}$$

$$\text{IF-FALSE}\ \frac{\langle e,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \mathit{false},l_1,c_1,x_1\rangle \qquad \langle s_2,\sigma,l_1,c_1 \cup \{s_2\},x_1\rangle \Downarrow_{deps} \langle \sigma_2,l_2,c_2,x_2\rangle}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi},\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma_2,l_2,c_2,x_2\rangle}$$

$$\text{WHILE-TRUE}\ \frac{\langle e,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \mathit{true},l_1,c_1,x_1\rangle \qquad \langle s_1,\sigma,l_1,c_1 \cup \{s_1\},x_1\rangle \Downarrow_{deps} \langle \sigma_1,l_2,c_2,x_2\rangle \qquad \langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma_1,l_2,c_2,x_2\rangle \Downarrow_{deps} \langle \sigma_2,l_3,c_3,x_3\rangle}{\langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma_2,l_3,c_3,x_3\rangle}$$

$$\text{WHILE-FALSE}\ \frac{\langle e,\sigma,l,c,x\rangle \Downarrow_{deps} \langle \mathit{false},l',c',x'\rangle}{\langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma,l,c,x\rangle \Downarrow_{deps} \langle \sigma,l',c',x'\rangle} \qquad \text{NUM}\ \frac{\langle n,\sigma\rangle \Downarrow n}{\langle n,\sigma,l,c,x\rangle \Downarrow_{deps} \langle n,l,c,x\rangle}$$

Fig. 8. Augmented semantics of $\mathcal{L}$ for computing test-equivalence classes w.r.t. $\overset{\mathrm{t}}{\sim}_{deps}$. v—variables, n—integer values, e—expressions, s—statements, σ—program states, l, c—sets of locations, x—Boolean values, *Inserted*—predicate over statements, and *Left*, *Right*—predicates over variables.

The semantics in Figure 8 extends the semantics in Figure 6 by defining the function $\Downarrow_{deps}$. It is parameterized by a predicate $\mathit{Inserted} : \mathit{Stmt} \to \mathbb{B}$ that marks the inserted assignment, a predicate $\mathit{Left} : \mathcal{V} \to \mathbb{B}$ that marks the left-hand side variable of the inserted assignment, and a predicate $\mathit{Right} : \mathcal{V} \to \mathbb{B}$ that marks the variables used in the right-hand side of the inserted assignment. $\Downarrow_{deps}$ additionally maintains

- l—a set of locations representing test-equivalent insertions;
- c—a set of locations that are executed after the last read/write of the variables involved in the inserted assignment;
- x—a Boolean value that indicates if the inserted assignment was evaluated after the last read/write of the variables involved in the inserted assignment.

The augmented semantics describes an analysis algorithm that, for a given set of locations, "filters out" those that do not correspond to the test-equivalent insertions of a given assignment. For a program with inserted assignment $v := e$, the semantics in Figure 8 computes sequences of executed locations (stored in the set c) such that each sequence contains the inserted statement $v := e$ (the rule ASSIGN-INS) and all the rest of the statements in this sequence do not read/overwrite the variable v and do not overwrite the variables in e (the rule ASSIGN-NLR). When such a sequence is found, the set of locations executed in this sequence is intersected with the current set of test-equivalent insertions ($l \cap c$ in the rules VAR-LEFT-EXE and ASSIGN-LR-EXE). When the

inserted assignment $v := e$ is not executed in such a sequence, the set of locations executed in this sequence is removed from the set of test-equivalent insertions ($l \setminus c$ in the rules VAR-LEFT-NEXE and ASSIGN-LR-NEXE), since for these locations this is no "matching" occurrence of $v := e$.

Definition 3.7 (Dependency-Based Test-Equivalence Relation). Let $\mathcal{P}$ be a set of programs, $t = (\sigma_{in}, \phi)$ be a test. $\overset{t}{\sim}_{deps} \subset \mathcal{P} \times \mathcal{P}$ is a *dependency-based test-equivalence relation* iff $p_1 \overset{t}{\sim}_{deps} p_2$ for $p_1, p_2 \in \mathcal{P}$ if there is program p with locations l_1, l_2 such that $p_1 = p[l_1/\mathsf{v} := \mathsf{e};l_1]$ and $p_2 = p[l_2/\mathsf{v} := \mathsf{e};l_2]$ and $\langle p_1, \sigma_{in}, \{l_1, l_2\}, \emptyset, \mathit{false}\rangle \Downarrow_{deps} \langle \_, \{l_1, l_2\}, \_, \_\rangle$ given that $\mathit{Inserted} := (\lambda s.\ s = \text{“}v := e'')$, $\mathit{Left} := (\lambda v'.\ v' = v)$, $\mathit{Right} := (\lambda v'.\ v' \in \mathit{Var}(e))$.

In this definition, we call two programs that differ in locations of an assignment insertion to be test-equivalent if the difference does not affect dynamic data dependencies according to the semantics in Figure 6. Specifically, by passing the program p_1, the test input σ_{in} and the set of locations $\{l_1, l_2\}$ as the arguments to $\Downarrow_{deps}$, we obtain the same set $\{l_1, l_2\}$ as the result.

Proposition 3.8. *The relation $\overset{t}{\sim}_{deps}$ is a test-equivalence relation according to Definition 1.1.*

The proposition above formally states that (1) $\overset{t}{\sim}_{deps}$ is an equivalence relation and (2) if two programs that differ only in locations in which the same assignment statement is inserted are such that these differences do not impact dynamic data dependencies (according to the semantics in Figure 8), then these two programs either both pass the test or both fail the test. A proof for the above proposition is given in Section C.

Example 3.9 (Test-Equivalent Programs w.r.t. $\overset{t}{\sim}_{deps}$). Consider a program p_1 defined as

$$
\begin{array}{l}
x := y; \\
\texttt{if } y > 0 \texttt{ then} \\
\quad y := x + 1 \\
\texttt{else skip fi},
\end{array}
$$

a program p_2 defined as

$$
\begin{array}{l}
\texttt{if } y > 0 \texttt{ then} \\
\quad x := y; \\
\quad y := x + 1 \\
\texttt{else skip fi},
\end{array}
$$

and a test $t := (\sigma_{in}, \lambda\sigma.\ \sigma(y) = 3)$ where the input state $\sigma_{in} := \{x \mapsto 1, y \mapsto 2\}$. These programs are test-equivalent w.r.t. the dependency-based test-equivalence relation for the test t, since they differ only in the location of the assignment $x := y$ and the following relation holds:

$$\langle p_1, \sigma_{in}, \{l_1, l_2\}, \emptyset, \mathit{false}\rangle \Downarrow_{deps} \langle \sigma_{out}, \{l_1, l_2\}, \_, \_\rangle,$$

where l_1 is the location of $x := y$ in p_1, l_2 is the location of $x := y$ in p_2, $\sigma_{out} := \{x \mapsto 2, y \mapsto 3\}$. This relation can be established by the derivation tree in Figure 9.

3.5 Composing Relations

In the proposed test-equivalence analysis framework, several relations can be composed in a mutually reinforcing fashion. By combining several analyses we can produce a more effective (coarse-grained) partitioning of the space of program modifications into test-equivalence classes.

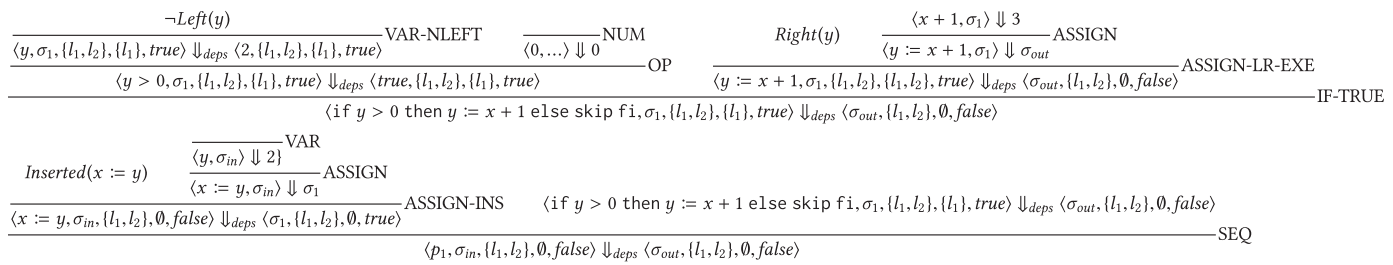

Fig. 9. Derivation tree for dependency-based test-equivalence relation in Example 3.9 ($\sigma_{in} := \{x \mapsto 1, y \mapsto 2\}$, $\sigma_1 := \{x \mapsto 2, y \mapsto 2\}$, $\sigma_{out} := \{x \mapsto 2, y \mapsto 3\}$).

$M(s_1; s_2) = \cup_{s' \in M(s_1)}\{\ s'; s_2\ \} \cup \cup_{s' \in M(s_2)}\{\ s_1; s'\ \}$

$M(\texttt{if}\ e\ \texttt{then}\ s_1 \texttt{else}\ s_2\ \texttt{fi}) = \cup_{s' \in M(s_1)}\{\ \texttt{if}\ e\ \texttt{then}\ s' \texttt{else}\ s_2\ \texttt{fi}\ \} \cup \cup_{s' \in M(s_2)}\{\ \texttt{if}\ e\ \texttt{then}\ s_1 \texttt{else}\ s'\ \texttt{fi}\ \}$

$M(\texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}) = \cup_{s' \in M(s)}\{\ \texttt{while}\ e\ \texttt{do}\ s'\ \texttt{od}\}$

$M(s \mid s \text{ is not a sequence}) = M_{EXPRESSION}(s) \cup M_{REFINEMENT}(s) \cup M_{GUARD}(s) \cup M_{ASSIGNMENT}(s)$

$M_{EXPRESSION}(v := e) = \cup_{e' \in \mathcal{E}}\{\ v := e'\ \}$

$M_{EXPRESSION}(\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}) = \cup_{e' \in \mathcal{E}}\{\ \texttt{if}\ e'\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \}$

$M_{EXPRESSION}(\texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}) = \cup_{e' \in \mathcal{E}}\{\ \texttt{while}\ e'\ \texttt{do}\ s\ \texttt{od}\}$

$M_{REFINEMENT}(\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}) = \cup_{e' \in \mathcal{E}}\{\ \texttt{if}\ e\ \texttt{and}\ e'\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}, \texttt{if}\ e\ \texttt{or}\ e'\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \}$

$M_{REFINEMENT}(\texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}) = \cup_{e' \in \mathcal{E}}\{\ \texttt{while}\ e\ \texttt{and}\ e'\ \texttt{do}\ s\ \texttt{od}, \texttt{while}\ e\ \texttt{or}\ e'\ \texttt{do}\ s\ \texttt{od}\}$

$M_{GUARD}(v := e) = \cup_{e' \in \mathcal{E}}\{\ \texttt{if}\ e'\ \texttt{then}\ v := e\ \texttt{else skip fi}\ \}$

$M_{ASSIGNMENT}(s) = \cup_{e' \in \mathcal{E}, v' \in \mathcal{V}}\{\ v' := e';\ s\ \}$

Fig. 10. Search space definition via transformation schemas M.

Definition 3.10 (Composition of Test-Equivalence Relations). Let $\mathcal{P}$ be a finite search space and $\overset{t}{\sim}_1, \overset{t}{\sim}_2, \ldots, \overset{t}{\sim}_n$ be test-equivalence relations in $\mathcal{P}$. A *composition* of $\overset{t}{\sim}_1, \overset{t}{\sim}_2, \ldots, \overset{t}{\sim}_n$ is a test-equivalence relation $\overset{t}{\sim}^*$ such that it is the transitive closure of the union of $\overset{t}{\sim}_1, \overset{t}{\sim}_2, \ldots, \overset{t}{\sim}_n$:

$$\overset{t}{\sim}^* := \left(\bigcup_i \overset{t}{\sim}_i \right)^*.$$

In this work, we define a test-equivalence relation $\overset{t}{\sim}^*$ as a composition of the relations $\overset{t}{\sim}_{value}$ and $\overset{t}{\sim}_{deps}$ introduced in Section 3.3 and Section 3.4:

$$\overset{t}{\sim}^* := \left(\overset{t}{\sim}_{value} \cup \overset{t}{\sim}_{deps} \right)^*.$$

An example of applying $\overset{t}{\sim}^*$ is given in Section 2.3.

The definition above is non-constructive in that it does not define an algorithm of computing test-equivalence classes w.r.t. $\overset{t}{\sim}^*$. One possible way to compute test-equivalence classes is to separately evaluate a given program using the semantics in Figure 7 and Figure 8 and merge the results; however, this requires multiple program executions. Instead, we use a more efficient approach by defining an augmented semantics of $\mathcal{L}$ for $\overset{t}{\sim}^*$ by combining the semantics in Figure 7 and Figure 8 as shown in Section B. This semantics enables our approach to compute a test-equivalence class w.r.t. $\overset{t}{\sim}^*$ via a single execution.

4 PATCH GENERATION

Automated program repair techniques search for patches in spaces of candidate program modifications. A search space in program repair is defined as in the following.

Definition 4.1 (Search Space). A *search space* is a finite set of syntactically different programs obtained by applying a given transformation function $M : \mathcal{L} \to 2^{\mathcal{L}}$ to the buggy program.

Previous systems (e.g., SPR/Prophet [15, 17] and SemFix/Angelix [22, 25]) defined their search spaces through *parameterized transformation schemas* such that each schema transforms a given program into a program with "holes" and the "holes" are filled with expressions using a program synthesizer. We define our search space in a similar fashion via the function M in Figure 10 ($\mathcal{E}$ indicates the synthesized expressions).

ALGORITHM 2: Enumerative patch generation

Input: search space $\mathcal{P}$, cost function κ, test-suite T
Output: ordered set of repairs R

$R := \emptyset$;
while $\mathcal{P} \neq \emptyset$ **do**
  p := $pick(\mathcal{P}, \kappa)$;
  $isPassing := true$;
  foreach $t \in T$ **do**
    $isPassing := eval(p, t)$;
    if $\neg isPassing$ **then**
      break;
  if $isPassing$ **then**
    $R := R \cup \{p\}$;
  $\mathcal{P} := \mathcal{P} \setminus \{p\}$;
return R;

ALGORITHM 3: Systematic exploration with partitioning into test-equivalence classes

Input: search space $\mathcal{P}$, cost function κ, test-suite T, test-equivalence relation $\overset{t}{\sim}$
Output: ordered set of repairs R

$R := \emptyset$;
foreach $t \in T$ **do**
  $C(t), \overline{C}(t) := \emptyset, \emptyset$;
while $\mathcal{P} \neq \emptyset$ **do**
  p := $pick(\mathcal{P}, \kappa)$;
  if $\exists t. \bigvee_{c \in \overline{C}(t)} p \in c$ **then**
    $\mathcal{P} := \mathcal{P} \setminus \{p\}$;
    continue;
  foreach $t \in T$ **do**
    if $\bigvee_{c \in C(t)} p \in c$ **then**
      continue;
    $isPassing,\ [p] := \widetilde{eval}(p, t, \mathcal{P}, \overset{t}{\sim})$;
    if $isPassing$ **then**
      $C(t) := C(t) \cup \{[p]\}$;
    else
      $\overline{C}(t) := \overline{C}(t) \cup \{[p]\}$;
      break;
  if $\forall t. \bigvee_{c \in C(t)} p \in c$ **then**
    $R := R \cup \{p\}$;
  $\mathcal{P} := \mathcal{P} \setminus \{p\}$;
return R;

Definition 4.2 (Optimal Program Repair). Let T be a test-suite (a set of tests), $p \in \mathcal{L}$ be a buggy program ($\exists t \in T.\ \neg Pass[p, t]$), $M : \mathcal{L} \to 2^{\mathcal{L}}$ be a transformation function, $\mathcal{P} := M(p)$ be the corresponding search space, $\kappa : \mathcal{P} \to \mathbb{R}$ be a cost function. The goal of *optimal program repair* is to find a repair $p' \in \mathcal{P}$ such that $\forall t \in T.\ Pass[p', t]$ and $\kappa(p')$ is minimal among all such programs.

Our patch generation algorithm systematically explores the search space by (1) evaluating candidates in the order defined by the prioritization (cost function) starting from the highest priority

patch and (2) skipping redundant executions by on-the-fly identifying test-equivalence classes w.r.t. a given test-equivalence relation.

In order to abstract over various optimal synthesis methodologies (synthesis with cost), we assume that there is a function *pick* that for a given set of programs $\mathcal{P}$ and a cost function κ, returns a program from $\mathcal{P}$ with the minimal value of κ.

Consider a baseline enumerative patch generation method described in Algorithm 2. It takes a patch space (Definition 4.1), a cost function, and a test-suite as inputs, and outputs a sequence of search space elements that pass all the given tests ordered according to the cost function. First, the algorithm initializes the list of output repairs R. Second, it iterates through the search space by (1) picking the best (the lowest cost according to κ) remaining candidate using *pick* and (2) evaluating the candidate with the tests using *eval*. Finally, it outputs the list of found plausible patches R.

The overall workflow of our approach is described in Algorithm 3. Our algorithm takes a patch space, a cost function, a test-suite, and a test-equivalence relation as inputs and outputs a sequence of search space elements that pass all the given tests ordered according to the cost function. Compared with Algorithm 2, our test-equivalence based algorithm also maintains sets C and $\overline{C}$ for each test. $C(t)$ is a set of test-equivalence classes (therefore, C is a set of sets of programs) in which all candidates pass t; $\overline{C}(t)$ is the corresponding set of failing test-equivalence classes. First, our algorithm initializes the list of output repairs R and the passing and failing test-equivalence classes C and $\overline{C}$ for all tests t. Second, it iterates through the search space by (1) picking the best (the lowest cost according to κ) remaining candidate using *pick* and (2) evaluating the candidate with the tests and computing test-equivalence classes.

For a given candidate, in order to identify the result of a test execution and the corresponding test-equivalence class, the algorithm evaluates the candidate using the function $\widetilde{eval}$. The function $\widetilde{eval}$ takes a program p, a test t, a search space $\mathcal{P}$, and a test-equivalence relation $\overset{t}{\sim}$ and returns the result of executing p with t (as a Boolean value *isPassing*) and a set of programs $[p]$ such that $[p]$ is a test-equivalence class of p in $\mathcal{P}$ w.r.t. the relation $\overset{t}{\sim}$. The concrete implementation of $\widetilde{eval}$ depends on the relation $\overset{t}{\sim}$ and is formally described for the relations $\overset{t}{\sim}_{value}$ and $\overset{t}{\sim}_{deps}$ in Definition 4.3 and Definition 4.4, respectively.

The test-equivalence classes are used at two steps of search space exploration. First, after a next candidate is picked, the algorithm checks if the candidate belongs to any of the existing failing classes (line 6). If the candidate is in a failing class of at least one test, evaluation of this candidate is omitted. Second, after a next test is selected for evaluating a candidate, the algorithm checks if the candidate is in a passing class of the given test (line 10). If a candidate is in a passing class of a test, then the algorithm omits execution of this candidate with this test. We now discuss the function $\widetilde{eval}$ specifically for the two notions of test-equivalence we have studied: value-based test-equivalence and dependency-based test-equivalence.

Definition 4.3 (Value-Based Test-Equivalence Analysis). Let $\mathcal{P}$ be a search space, $p \in \mathcal{P}$ be a program, $t = (\sigma_{in}, \phi)$ be a test. Let e be an expression in p such that $\exists p' \in \mathcal{P} \exists e' \in Expr.\ p' = p[e/e']$. Then, value-based test-equivalence analysis $\widetilde{eval}$ is defined as follows:

$$\widetilde{eval}(p, t, \mathcal{P}, \overset{t}{\sim}_{value}) = \left(\phi(\sigma_{out}), \bigcup_{e' \in C} \{p[e/e']\}\right), \text{ given that}$$
$$Modified := \lambda x.\ x = e,$$
$$\mathcal{E} := \{e' \mid \exists p' \in \mathcal{P}.\ p' = p[e/e']\},$$
$$\langle p, \sigma_{in}, \mathcal{E}\rangle \Downarrow_{value} \langle \sigma_{out}, C\rangle.$$

In this analysis, we identify a test-equivalence class of a program with an expression e in the space of all programs in $\mathcal{P}$ that differ only in e. The test-equivalence class is computed by passing the set of all "alternative" expressions $\mathcal{E}$ as an argument of $\Downarrow_{value}$. Note that in this definition we explicitly select an expression e, substitution of which is analyzed for test-equivalence. For each element of a search space produced by the transformation function M in Figure 10, there is always at most one such e. We now discuss the function $\widetilde{eval}$ for dependency-based test-equivalence.

Definition 4.4 (Dependency-Based Test-Equivalence Analysis). Let $\mathcal{P}$ be a search space, $p \in \mathcal{P}$ be a program, $t = (\sigma_{in}, \phi)$ be a test. Let p' be a program, l_1 be a location such that $p = p'[l_1/\mathsf{v} := \mathsf{e};l_1]$, and $\exists p'' \in \mathcal{P}.\ \exists l_2 \in p''.\ p'' = p'[l_2/\mathsf{v} := \mathsf{e};l_2]$. Then, dependency-based test-equivalence analysis $\widetilde{eval}$ is

$$\widetilde{eval}(p, t, \mathcal{P}, \overset{t}{\sim}_{deps}) = \left(\phi(\sigma_{out}), \bigcup_{l \in L'} \{p'[l/\mathsf{v} := \mathsf{e};l]\}\right), \text{ given that}$$

$$Inserted := \lambda s.\ s = \text{“}v := e\text{”},\ Left := \lambda v'.\ v' = v,\ Right := \lambda v'.\ v' \in Var(e),$$

$$L := \{\ l \mid l \in p' \wedge \exists p'' \in \mathcal{P}.\ p'' = p'[l/\mathsf{v} := \mathsf{e};l],$$

$$\langle p, \sigma_{in}, L, \emptyset, false\rangle \Downarrow_{deps} \langle \sigma_{out}, L', \_, \_\rangle.$$

In this analysis, we identify a test-equivalence class of a program with an assignment $v := e$ in the space of all programs in $\mathcal{P}$ that differ only in locations of this assignment. The test-equivalence class is computed by passing the set of all "alternative" locations L as an argument of $\Downarrow_{deps}$.

Finally, note that Algorithm 3 can be used in different ways. The output of the algorithm is a sequence of plausible patches R ordered according to the function κ. A sequence of repairs can be used to provide several patch suggestions for developers. The number of suggested repairs can be controlled by introducing a limit and breaking from the main loop when the required number of plausible patches is found. Certain applications may require generation of all plausible patches (e.g., in order to narrow candidates through test generation [31]). In this case, the algorithm can be modified so that it outputs whole test-passing partitions instead of single patches.

5 IMPLEMENTATION

We have implemented the described approach in a tool called f1x (pronounced as [ɛf-wʌn-ɛks]) for the C programming language.

Analysis. Our implementation of the proposed test-equivalence analyses is built upon a combination of static (source code) and dynamic instrumentation. Specifically, to implement the augmented semantics in Section 3.3 for the relation $\overset{t}{\sim}_{value}$, we apply the transformation schemas M (Figure 10) to the source code of the buggy program and replace "holes" with calls to a procedure implementing the value-projection operator. To implement the augmented semantics in Section 3.4 for the relation $\overset{t}{\sim}_{deps}$, we implemented a dynamic instrumentation using Pin [18] that tracks reads and writes of the variables involved in assignment synthesis.

Search Space. The goal of this work was to design and evaluate test-equivalence relations for transformations used in existing program repair systems. Our system combines the transformation schemas of SPR/Prophet and Angelix (we studied implementation of these systems in order to closely reproduce their search spaces), described as follows.

EXPRESSION Modify an existing side-effect free integer expression or condition (adopted from Angelix). A variant of $M_{EXPRESSION}$ in Figure 10 for C programs. Partitioned into test-equivalence classes based on the expression values using $\overset{t}{\sim}_{value}$.

REFINEMENT Append a disjunct/conjunct to an existing condition (adopted from Prophet). A variant of $M_{REFINEMENT}$ in Figure 10 for C programs. Partitioned into test-equivalence classes based on the condition values using $\overset{t}{\sim}_{value}$.

GUARD Add an if-guard for an existing statement (adopted from Angelix and Prophet). A variant of M_{GUARD} in Figure 10 for C programs. Partitioned into test-equivalence classes based on the condition values using $\overset{t}{\sim}_{value}$.

ASSIGNMENT Insert an assignment statement (adopted from Prophet[7]). A variant of $M_{ASSIGNMENT}$ in Figure 10 for C programs. Partitioned into test-equivalence classes using $\overset{t}{\sim}^{*}$.

INITIALIZATION Insert memory initialization (adopted from Prophet). Not partitioned.

FUNCTION Replace a function call with another function (adopted from Prophet). Not partitioned.

The two last transformation schemas adopted from Prophet are not partitioned by our algorithm, since they generate relatively small search spaces. Our transformations differ from that of SPR/Prophet in the following ways: (1) Prophet implements a transformation schema for inserting guarded return statements. Although our algorithm can partition these transformations using the relation $\overset{t}{\sim}_{value}$, such transformations were shown to frequently generate overfitting patches [38] and therefore we exclude them from our search space; (2) Prophet implements a transformation that copies existing program statements. Since such statements can be arbitrarily complex and they cannot be partitioned by our algorithm, we do not include this transformation.

Cost Function. Several techniques have been proposed to increase the probability of generating correct repairs by prioritizing patches [3, 17, 21]. For our system, we implement an approach that assigns higher priority to smaller changes [21]:

$$\kappa(p) := distance(p,\ p_{orig}),$$

where p is a patched program (an element of the search space), p_{orig} is the original program, and *distance* is defined as the number of added, modified, and deleted AST nodes.

6 EXPERIMENTAL EVALUATION

We evaluate our approach in terms of the following research questions:

(RQ1) What are the effectiveness and efficiency of our approach compared with state-of-the-art program repair systems?

(RQ2) Does our approach scale to larger search spaces compared with state-of-the-art systems? Does test-equivalence relation enable higher scalability of our implementation?

(RQ3) What is the impact of each test-equivalence relation on the number of test executions performed by our algorithm?

6.1 Evaluation Setup

Our evaluation compares f1x against three repair approaches: Angelix, Prophet, and GenProg-AE. These repair techniques are chosen as they use different repair algorithms including symbolic analysis (Angelix), machine learning (Prophet), and genetic algorithm (GenProg). We evaluate all repair approaches on the GenProg ICSE'12 benchmark [12] for our evaluation because it includes defects from large real-world projects, and was designed for systematic evaluation of program

[7]Prophet generates new assignments by copying and modifying existing assignments. Instead, f1x *synthesizes* assignments and therefore its search space includes a superset of assignments that can be generated by Prophet.

Table 1. Subject Programs and Their Basic Statistics

Program	Description	LOC	Defects	Tests	Execution cost (sec)
libtiff	Image processing library	77K	24	78	8.18
lighttpd	Web server	62K	9	295	29.75
php	Interpreter	1,046K	44	8,471	427.25
gmp	Math library	145K	2	146	53.52
gzip	Data compression utility	491K	5	12	0.43
python	Interpreter	407K	11	35	156.46
wireshark	Network packet analyzer	2,814K	7	63	9.92
fbc	Compiler	97K	3	773	240.27

repair tools. Moreover, the test suites in this benchmark were independently augmented to prevent repair tools from generating implausible patches [28]. The benchmark consists of 105 defects from eight subjects (i.e., libtiff, lighttpd, PHP, gmp, gzip, python, wireshark, and fbc) which have developer-written test suites. Table 1 shows the statistics of each evaluated subject. The column "Execution cost" denotes the time taken to execute the test-suite for a given subject.

We selected the following systems and their configurations for evaluation:

F1X f1x that implements search with test-equivalence partitioning described in Algorithm 3.

F1X^E f1x^E is a variant of f1x that enumerates changes without test-equivalence partitioning (Algorithm 2). It is considered to evaluate implementation-independent effect of partitioning.

ANG Angelix 1.1 [22] that implements a symbolic path exploration and prioritizes syntactically small changes.

PR Prophet 0.1 [17] that implements value search (a variant of path exploration) for conditional expressions and patch prioritization based on machine learning.

PR* Prophet* that is a variant of Prophet that disables transformations for (1) inserting overfitting return insertions and (2) copying complex statements except for assignments. This variant is considered to match the transformation implemented in F1X/F1X^E, since the search space of F1X/F1X^E is effectively the combination of the search spaces of PR* and ANG.

GP GenProg-AE 3.0 [42] that implements a group of analysis techniques to avoid evaluating functionally equivalent patches (as opposite to test-equivalent as in our approach). Compared to the earlier version of GenProg that uses genetic algorithm [14] that is inherently stochastic, GenProg-AE leverages a deterministic repair algorithm.

We run all the configurations (F1X, F1X^E, ANG, PR, PR*, GP) in two modes:

Stop-after-first-found The algorithm terminates after finding the first patch. This mode represents the usual program repair usage scenario.

Full exploration The algorithm terminates after searching through the entire search space. This mode allows us to obtain data that is independent on (1) the exploration order and (2) whether a plausible patch is present in the search space.

We reuse the configurations from previous studies for running Angelix, Prophet, and GenProg-AE [28, 42]. As Prophet takes a correctness model as input to prioritize patches akin to the provided model, we used the default model that is publicly available.[8]

[8]Prophet website: http://rhino.csail.mit.edu/prophet-rep/.

Table 2. Effectiveness of Program Repair Approaches

Subject	Plausible						Equivalent to human					
	F1X	F1X^E	ANG	PR	PR*	GP	F1X	F1X^E	ANG	PR	PR*	GP
libtiff	13	10	10	5	3	5	5	3	3	2	1	0
lighttpd	5	3	-	4	4	4	0	0	-	0	0	0
php	15	7	10	18	15	7	6	3	4	10	6	2
gmp	2	1	2	2	2	1	2	1	2	1	1	0
gzip	3	2	2	2	2	2	2	0	1	1	1	0
python	5	1	-	6	5	3	0	0	-	0	0	1
wireshark	4	4	4	4	4	4	0	0	0	0	0	0
fbc	1	1	-	1	1	1	1	1	-	1	1	0
Overall	49	29	28	42	36	27	16	8	10	15	10	3

We conduct all experiments on Intel® Xeon™ CPU E5-2660 machines running Ubuntu 14.04, and use a 10 hours timeout for running each configuration.

6.2 Effectiveness and Efficiency (RQ1)

Table 2 summarizes the effectiveness results for F1X, F1X^E, ANG, PR, PR*, and GP executed in the stop-after-first-found mode. The second through seventh columns denote the number of plausible patches generated by each repair approach, while the eighth through thirteenth columns represent the number of patches syntactically equivalent to the human patches. As Angelix does not support lighttpd, python, and fbc, the corresponding cells for these subjects are marked with "-." The overall results illustrate that F1X generates the highest number of plausible patches compared to all other evaluated repair approaches. The "Equivalent to human" column in Table 2 shows that F1X generates eight more human-like patches than F1X^E, six more human-like patches than ANG, one more human-like patch than PR, two more human-like patches than PR*, and 13 more human-like patches than GP.

We attribute the high number of patches generated by F1X to the larger patch space supported by F1X compared to other approaches. Since F1X combines the search spaces of ANG and PR*, it fixes all defects that are fixed by either of these tools. Note that F1X finds more patches than F1X^E within the time limit due to the performance gain from our partitioning.

Figure 11 illustrates the average patch generation time for the configurations. The x-axis of Figure 11 represents the eight subjects in the benchmark, while the y-axis shows the average time taken to generate a patch for all defects for a given subject where each bar depicts a patch generation approach. Overall, the average patch generation time for F1X is significantly shorter than all other repair approaches. For instance, F1X requires only 121 seconds on average to generate a patch for libtiff, while ANG takes 1,262 seconds (F1X is $\frac{1262}{121}$=10.5× faster than ANG). Meanwhile, PR* takes 1,701 seconds on average to produce a patch for libtiff (F1X is $\frac{1701}{121}$=14× faster than PR*). Notably, F1X is 16× faster than GP for libtiff (GP takes 1,940 seconds on average to generate a patch for libtiff). The average patch generation time for PR is slightly higher compared to PR* as it searches through a slightly larger patch space.

The results shown in Figure 11 validate our claim that F1X is able to achieve significant improvement on the patch generation time due to its efficient search algorithm. F1X and F1X^E demonstrate a comparable average time of patch generation. Note that for some subjects (e.g., python), the average time of F1X^E is lower than that of F1X. This is because F1X^E finds a subset of patches found by F1X, and patches found exclusively by F1X contribute to its higher average time. However, when

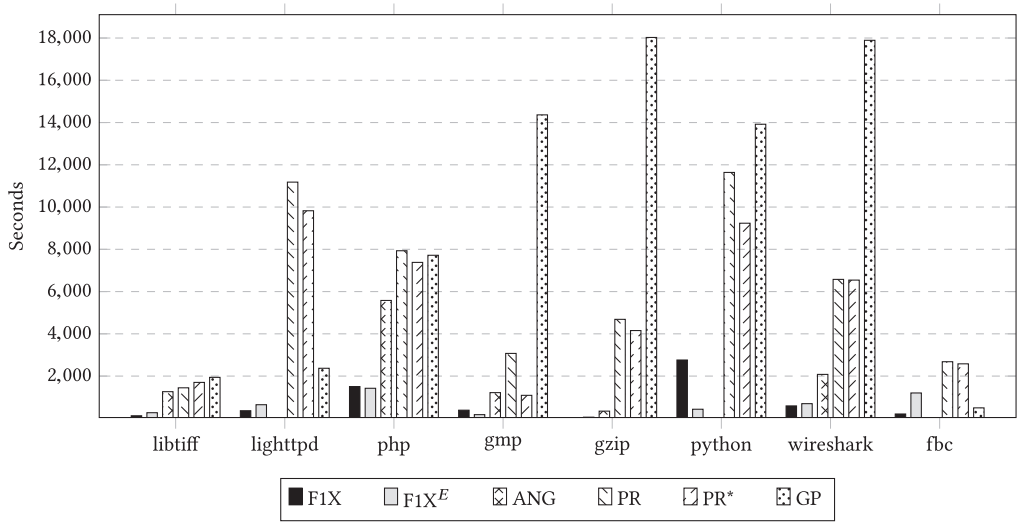

Fig. 11. Average patch generation time.

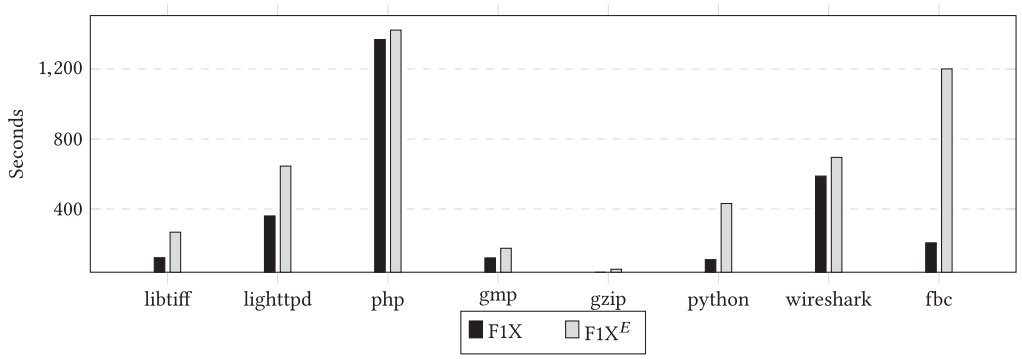

Fig. 12. Average time for patches generated by both F1X and F1X^E.

considering only patches found by both F1X and F1X^E, F1X shows consistently better performance, as shown in Figure 12.

RQ1: Compared with existing automated program repair tools, f1x generates more patches since its search space is a combination of the search spaces of ANG and PR*. Despite a larger search space, it finds patches in an order of magnitude faster.

6.3 Exploration Speed (RQ2)

Definition 6.1 (Explored Candidates). We say that a candidate patch is *explored* if the algorithm identified whether the patch passes all given tests or fails at least one. Note that we only consider candidate patches in which the source code modification is executed by all given failing tests.

Table 3 shows the exploration statistics for F1X, F1X^E, PR, PR*, and GP (we exclude ANG because the search space for Angelix is encoded via logical constraints). The second through sixth columns depict the data for the stop-after-first-found mode, while the seventh through eleventh columns

Table 3. Exploration Statistics of Program Repair Tools in Stop-After-First-Found and Full Exploration Modes

Subject	Stop-after-first-found mode					Full exploration mode				
	F1X	F1X^E	PR	PR*	GP	F1X	F1X^E	PR	PR*	GP
libtiff	$\frac{44,675}{243}$=183.8	$\frac{1,241}{1,272}$=1.0	$\frac{14,980}{2,946}$=5.1	$\frac{6,995}{4,625}$=1.5	$\frac{1,371}{1,302}$=1.1	$\frac{40,0786}{759}$=528	$\frac{23,872}{24,512}$=1.0	$\frac{62,521}{23,763}$=2.6	$\frac{31,403}{16,777}$=1.9	$\frac{4,564}{4,657}$=1.0
lighttpd	$\frac{1,085}{220}$=4.9	$\frac{512}{616}$=0.8	$\frac{4,645}{1,295}$=3.6	$\frac{6,269}{2,563}$=2.4	$\frac{359}{383}$=0.9	$\frac{112,667}{850}$=132	$\frac{10,283}{10,580}$=1.0	$\frac{45,338}{8,377}$=5.4	$\frac{29,106}{10,126}$=2.9	$\frac{2,337}{2,371}$=1.0
php	$\frac{2,843}{1,717}$=1.7	$\frac{3,407}{5,184}$=0.7	$\frac{1,186}{48,070}$=0.0	$\frac{2,566}{56,492}$=0.0	$\frac{88}{7,736}$=0.0	$\frac{77472}{193}$=401	$\frac{9,801}{12,139}$=0.8	$\frac{6,378}{58,671}$=0.1	$\frac{4,942}{73,898}$=0.1	$\frac{2,057}{3,497}$=0.6
gmp	$\frac{5,517}{173}$=31.9	$\frac{30}{31}$=1.0	$\frac{3,215}{2,002}$=1.6	$\frac{2,064}{1,260}$=1.6	$\frac{9,128}{9,172}$=1.0	$\frac{80,448}{1,140}$=70	$\frac{16,934}{23,994}$=0.7	$\frac{17,526}{14,523}$=1.2	$\frac{11,423}{10,156}$=1.1	$\frac{2,841}{2,841}$=1.0
gzip	$\frac{13,071}{123}$=106.3	$\frac{241}{251}$=1.0	$\frac{5,892}{1,340}$=4.4	$\frac{5,284}{890}$=5.9	$\frac{7,605}{5,735}$=1.3	$\frac{518,803}{568}$=913	$\frac{18,501}{24,789}$=0.7	$\frac{59,467}{28,716}$=2.1	$\frac{48,491}{40,574}$=1.2	$\frac{9,007}{9,098}$=1.0
python	$\frac{940}{231}$=4.1	$\frac{29}{37}$=0.8	$\frac{4,211}{7,389}$=0.6	$\frac{6,928}{8,771}$=0.8	$\frac{3,327}{3,319}$=1.0	$\frac{105,842}{1,617}$=65	$\frac{8,381}{12,494}$=0.7	$\frac{13,367}{9,213}$=1.5	$\frac{9,928}{9,050}$=1.1	$\frac{7,823}{7,824}$=1.0
wshark	$\frac{828}{139}$=6.0	$\frac{595}{597}$=1.0	$\frac{3,744}{2,460}$=1.5	$\frac{3,948}{2,434}$=1.6	$\frac{4,253}{4,297}$=1.0	$\frac{689,925}{620}$=1{,}112	$\frac{14,094}{14,003}$=1.0	$\frac{23,043}{13,099}$=1.8	$\frac{18,554}{17,442}$=1.1	$\frac{2,008}{2,213}$=0.9
fbc	$\frac{948}{434}$=2.1	$\frac{650}{891}$=0.7	$\frac{1,111}{672}$=1.65	$\frac{524}{394}$=1.33	$\frac{22}{788}$=0.0	$\frac{50,195}{1,312}$=38	$\frac{20,195}{24,521}$=0.8	$\frac{892}{589}$=1.51	$\frac{766}{499}$=1.54	$\frac{852}{852}$=1.0

represent the data for the full exploration mode. Each cell in the second through eleventh columns is of the form $\frac{X}{Y}$=Z where X represents the average number of "explored candidates" by a repair approach, Y represents the average number of test executions performed by a repair approach, Z denotes the exploration speed (computed by the ratio of the number of "explored candidates" over the number of test executions). The average for the stop-after-first-found is computed among the fixed defects, whereas the average for the full exploration mode is computed among all defects.

In general, F1X has an order of magnitude higher exploration speed compared to all other patch generation approaches in both the stop-after-first-found mode and the full exploration mode. For example, in the full exploration mode, F1X requires on average only 620 test executions to explore 689,925 candidates for wireshark. For the same subject, PR requires 13,099 test executions for exploring 23,043 candidates, PR* requires 17,442 test executions for exploring 18,554 candidates, and GP requires 2,213 test executions for exploring 2,008 candidates. The efficiency of exploration can also be indirectly shown by comparing the average number of plausible patches found in full exploration mode. For this, we compared the results of F1X and PR, since these configurations found the largest number of patches. F1X generates 2,265 plausible patches on average, while PR generates only 9 plausible patches on average.

To enable generation of more patches, an ideal repair approach should scale to larger spaces by exploring more candidates within the time budget. The data in Table 3 shows that F1X scales to larger search spaces, since it explores more candidates within the time limit due to fewer number of test executions. This explains the effectiveness and the efficiency of F1X compared with other tools shown in Table 2 and Figure 11. Recall that F1X^E is a variant of F1X without test-equivalence partitioning. For the same search space, F1X^E explores less candidates than F1X within the time limit since it requires more test executions. From this observation, we conclude that test-equivalence partitioning is responsible for the higher scalability of F1X.

RQ2: f1x scales to larger search space by exploring more candidates with fewer number of test executions.

Table 4. Effect of Equivalence Relations on the Number of Test Executions on Libtiff

Transformation	Relation	Locations	Candidates	Partitions	Test executions
$M_{EXPRESSION}$	$\overset{t}{\sim}_{value}$	109	428,641	424	739
$M_{REFINEMENT}$	$\overset{t}{\sim}_{value}$	45	48,942	87	154
M_{GUARD}	$\overset{t}{\sim}_{value}$	87	106,389	199	347
$M_{ASSIGNMENT}$	$\overset{t}{\sim}_{value}$	121	17,181	3,436	5,983
	$\overset{t}{\sim}_{deps}$	121	1,7181	1,809	2,308
	$\overset{t}{\sim}^{*}$	121	1,7181	480	671

6.4 Effect of Equivalent Relation (RQ3)

To investigate the effect of each equivalence relation on the number of test executions, we conduct another experiment on all 24 defects in Libtiff. We selected Libtiff, because due to the structure of its source code, Libtiff subjects have the largest total number of candidate patches in the search space (9,618,864 = 400,786 ∗ 24), where 400,786 is the average number of candidates per version, 24 is the number of versions (see Table 3). Our goal is to determine if there is a single equivalence relation that dominantly contributes to the reduction in the number test executions or if a composition of these relations will result in greater reduction.

Table 4 shows the effect of the three equivalent relations ($\overset{t}{\sim}_{value}$, $\overset{t}{\sim}_{deps}$, and $\overset{t}{\sim}^{*}$) in F1X on the average number of test executions for Libtiff. For each transformation from Figure 10, the table demonstrates the number of locations in which the transformation was applied (the "Locations" column), the test-equivalence relations that were applied for the search space produced by this transformation (the "Relation" column), the number of different candidate patches generated (the "Candidates" column), the number of partitions that were identified for the failing test (the "Partitions" column), and the number of tests required to explore all the corresponding candidates with the whole test suite (the "Test executions" column).

These results demonstrate that for the transformations $M_{EXPRESSION}$, $M_{REFINEMENT}$, and M_{GUARD} our algorithm produces a small number of test-equivalence classes for the failing tests (with 500–1,000 elements in each partition on average) which also resulted in a small number of executions for the whole test suite. For the relation $M_{ASSIGNMENT}$, the reduction is less significant; however, the composition of relations $\overset{t}{\sim}^{*}$ is significantly more efficient (the number of test executions is significantly less) than the individual relations $\overset{t}{\sim}_{value}$, $\overset{t}{\sim}_{deps}$.

RQ3: f1x partitions a large number of patches into a small number of test-equivalence classes using two relations: $\overset{t}{\sim}_{value}$ and $\overset{t}{\sim}_{deps}$. The composition of these two relations is more effective than the individual relations.

7 RELATED WORK

Program Synthesis. Existing program synthesis techniques can be used to generate patches by directly searching in patch spaces; however, this approach has limitations as explained in the following. First, the cost of test execution in a typical program repair problem is substantially higher than that in program synthesis, since the subjects on which program repair is carried out today are significantly larger (e.g., a single test execution for PHP interpreter from GenProg benchmark [12] takes 1–10 seconds on commodity hardware, while a typical solution in SyGuS-Comp competition

can be executed in 10^{-6} seconds). Second, the complexity of repaired programs makes it infeasible to apply precise deductive techniques. For instance, Sketch [33] fills "holes" in sketches (partial programs), and can be potentially applied to generate repairs for identified suspicious statements. However, it translates programs into Boolean formulas and therefore can repair only relatively small programs [5]. Since program synthesis algorithms may not be directly applicable to program repair, they are used as parts of program repair algorithms for filling "holes" in programs based on inferred specification [15, 22]. In our technique, we do not use program synthesis as a black box, but integrate synthesis with program analysis by imposing additional requirements on the underlying synthesizer: support for the value-projection operator (Section 3.2). Since our implementation uses an enumerative program synthesis, it is straightforward to realize such operators. However, other techniques can also be used for this purpose. For instance, FlashMeta [26] compactly represents its search space as version space algebra (VSA) [23]. Moreover, it defines the operation *Filter* over this representation that is effectively the value-projection operator, therefore it can be potentially used in our algorithm as a more efficient representation of the space of program modifications.

Program Repair Search Algorithms. Syntax-based techniques generate patches by enumerating and testing syntactic changes. Since (1) repair tools have to explore large search spaces to address many classes of defects and (2) test execution has high cost for large real-world programs, they scale to relatively small search spaces. GenProg-AE [42] eliminates redundant executions by identifying functionally equivalent patches via lightweight analyses. Instead of functional equivalence, our technique applies test-equivalence, which is a weaker and therefore a more effective relation (produces larger equivalence classes). *Semantics-based* techniques split search into two phases. First, they localize suspicious statements and *infer specification* for the identified statements that captures the property of "passing the test suite." Such specification can be expressed as logical constraints [25] or *angelic values* [15, 44]. Second, they apply off-the-shelf program synthesizers in order to modify the selected statements according to this specification. To infer specification, existing techniques perform path exploration by altering test executions. Since the number of execution paths in programs can be infinite, these methods are subject to the *path explosion problem*. For instance, Nopol [44], SPR [15], and Prophet [17] enumerate values of conditional expressions (which is a special case of path exploration) in order to find angelic values that enable the program to pass the failing test. As shown in Section 2.1, such techniques may perform a large number of redundant executions that are avoided by our algorithm. SemFix [25] and Angelix [22] are semantics-based techniques relying on symbolic execution and SMT-based synthesis. Since our methodology does not use symbolic methods, it is orthogonal to SemFix and Angelix from the point of view of underlying analysis. Existing syntax and semantic-based techniques are limited to modifying side-effect free expressions. For instance, they can only generate new assignments by copying them from other parts of the program. Meanwhile, we demonstrate that test-equivalence can scale assignment synthesis using a combination of value- and dependency-based analyses.

Program Repair Prioritization Approaches. In order to address the test over-fitting problem [32], various techniques have been proposed to prioritize patches that are more likely to be correct. For instance, DirectFix [21] prioritizes candidate patches based on syntactic distance, Qlose [3] prioritizes patches bases on semantic distance, Prophet [17] utilizes information learned from human patches, ACS [43] prioritizes patches based on information mined from previous versions and API documentation, and S3 [11] prioritizes patches based on a combination of syntactic and semantic properties. Meanwhile, relifix [35] leverages different program versions for generating

patches, whereas SEMGRAFT [20] uses references implementation as implicit correctness criteria. Droix [36] fixes crashes in Android apps using Android activity lifecycle management rules for patch prioritization. Our technique finds the best patch (the global optimum) in its search space according to an arbitrary cost function. Meanwhile, previous techniques applied to large real-world programs did not provide such guarantees. Techniques based on genetic programming [2, 14] and random search [27] guarantee only a local optimum by definition. Semantics-based repair techniques may miss the global optimum, which is shown in Section 2.2.

Program Repair Using Types and Formal Specification. Several approaches utilize temporal logic formulas [7], contracts [41], and types [29] to guide program repair. Our test-equivalence analysis can potentially optimize these approaches. Besides, our test-driven patch generation algorithm can be used in a counterexample-guided refinement loop [1] to repair programs based on given formal specification.

Mutation Testing. The scalability of program repair is related to scalability of mutation testing, since mutation testing also evaluates a large number of program modifications. To address this problem in mutation testing, a common approach is to reduce the number of mutation operators (transformations) to avoid redundant executions [24]. This approach may not be suitable for program repair, because program repair search spaces have to be rich enough to enable generation of non-trivial human-like repairs. Test-equivalence has been applied to scale mutation testing. Mutant analysis by Just et al. [8] performs a pre-pass that partitions mutants based on infected states, which can thought of as a variant of the $\overset{t}{\sim}_{value}$ relation. More recent techniques [19, 40] extend this approach by performing more fine-grained partitioning. Our method adapts a value-based test-equivalence relation that has been used in mutation testing to program repair by integrating it with syntax-guided program synthesis through a value-projector operator (Section 3.2). Apart from that, we introduce a new dependency-based test-equivalence relation for partitioning more complex program modifications that might be required to fix real bugs.

Compiler Testing. Equivalence modulo inputs (EMI) [9, 10] (a variant of test-equivalence) was successfully applied to compiler testing. Specifically, for a compiler, a program, and an input, it generates input-equivalent variants of the program by altering unexecuted statements and checks that these variants compiled by the compiler produce the same outputs. A recent extension of this approach [34] modifies executed statements by inserting random code guarded by a condition that is evaluated into *false* in the context of the given test, which can be considered as an application of value-based test-equivalence relation. Our dependency-based test-equivalence relation and the proposed composition of several analyses might be used to increase the effectiveness of compiler testing by synthesizing non-trivial input-equivalent program modifications.

8 DISCUSSION AND FUTURE WORK

We envision the following design of a future *general-purpose program repair system* (a system that is able to address many kinds of defects in commodity software). This system will (1) implement a large number of transformations to address many kinds of defects, (2) implement test-equivalence analyses for these transformations to ensure scalability, and (3) implement intelligent search space prioritization strategies over the patch space, to address the overfitting problem. This work is a step toward such a design. In our future works, we plan to investigate the following aspects.

Test-Equivalence Relations. The effectiveness of the relation $\overset{t}{\sim}_{value}$ (the size of test-equivalence classes it induces) may depend on the size of the output domain of modified expressions. Since

it identifies equivalence based on concrete values, it may not be effective for expressions of large output domains (e.g., strings). In future, we plan to investigate a generalization of this relation that defines two changes to be test-equivalent iff they drive test execution along the same path. Such a relation will be computed using dynamic symbolic execution [4].

Transformations. Existing program repair systems provide very limited support for repairing function calls. We hypothesize that a generalization of the relation $\overset{t}{\sim}_{deps}$ may enable repair systems to extend search spaces by incorporating function call transformations in a scalable fashion.

Current Limitations. Although we demonstrated that our approach based on test-equivalence significantly outperforms previous repair techniques, it has several limitations. The proposed analysis assumes deterministic test execution. Besides, our algorithm is designed to search in finite spaces of candidate patches. It can be potentially generalized for infinite spaces by encoding them symbolically. Finally, the current algorithm is designed to synthesize single-line patches (involving a single modification). It can be potentially extended for multi-line modifications using the approach of Angelix [22].

Subject Programs. The subjects for the evaluation (GenProg ICSE'12 benchmark [12]) were previously used for evaluating related approaches [15, 17, 22]. GenProg ICSE'12 benchmark was constructed to address generalizability concerns in evaluation of repair tools [12]. Moreover, the test suites were independently augmented [28] to avoid generation of implausible patches. Nevertheless, a possible threat to validity is that our results may not generalize for other programs and defects. In the future, it may be worthwhile to evaluate our approaches on other relevant benchmarks [13, 37].

9 CONCLUSION

Traditionally, many problems in software testing which involve exploring huge search spaces, such as mutation testing, have been shown to benefit from test equivalence [8, 9]. In this article, we have adopted the notion of test-equivalence for program repair, and provided several test-equivalence relations: based on runtime values, based on dynamic data dependencies and their composition. We also proposed an algorithm of automatic patch generation based on on-the-fly test-equivalence analysis. This enables us to explore larger patch spaces in substantially less time, thereby significantly enhancing the practicality of modern day program repair technology.

APPENDICES

A ALTERNATIVE IMPLEMENTATIONS OF VALUE-PROJECTION OPERATOR

We define the value-projection operator $\Pi^{value}_{\sigma,n}$ (Section 3.2) for SMT-based synthesis [6] and DSL-based synthesis [26].

A.1 Symbolic Synthesis

In SMT-based component-based synthesis [6], the search space is represented implicitly through logical constraints. To synthesize an expression from a given specification, it solves the following formula:

$$\exists e \in \mathcal{E}. \bigwedge_{\sigma, n \in spec} \langle e, \sigma \rangle \Downarrow n.$$

To support quantification over expressions, it uses *location variables* (denoted via the prefix l) to encode all expressions constructed from given components C (e.g., +, −, variables) as follows.

$$\phi_{wpf} := \phi_{range} \wedge \phi_{cons} \wedge \phi_{acyc},$$

$$\phi_{range} := \bigwedge_{c \in C} \left(0 \leq lc^{out} < |C| \quad \wedge \bigwedge_{k \in [1, NI(c)]} 0 \leq lc^{in}_k < |C| \right),$$

$$\phi_{cons} := \bigwedge_{(c,s) \in C \times C, c \neq s} lc^{out} \neq ls^{out},$$

$$\phi_{acyc} := \bigwedge_{c \in C, k \in [1, NI(c)]} lc^{out} > lc^{in}_k,$$

$$\phi_{conn} := \bigwedge_{\substack{(c,\, s) \,\in\, C \times C \\ k \,\in\, [1..NI(s)]}} lc^{out} = ls^{in}_k \Rightarrow c^{out} = s^{in}_k.$$

The algorithm also imposes library constraint (ϕ_{lib}) that captures semantics of given components. For example, for the component $c := h_1 + h_2$, the library constraint is $c^{out} = c^{in}_1 + c^{in}_1$. Given the above constraints, the search space is encoded through the formula ϕ defined as $\phi := \phi_{wpf} \wedge \phi_{lib} \wedge \phi_{conn}$.

In this context, the value-projection operator can be implemented as follows:

$$\Pi^{value}_{\sigma,n}(\phi) := \phi \wedge e^{out} = n \wedge \bigwedge_{\text{variable} v \in C} v^{out} = \sigma(v),$$

where e^{out} captures the output of the synthesized expression. Effectively, this operator conjoins the formula representing the search space with the input-output relation represented via σ and n.

A.2 DSL-Based Synthesis

In the FlashMeta [26] synthesis framework, the search space is compactly represented via version space algebra (VSA) [23]. Assume that the set of expression $\mathcal{E}$ is defined through applications of operators to a given set of variables; we denote an operator as F.

The grammar for a version space algebra $\widetilde{N}$ is defined as

$$\widetilde{N} := \{ e_1, \ldots, e_k \} \mid U(\widetilde{N}_1, \ldots, \widetilde{N}_k) \mid F_{\triangleright\triangleleft}(\widetilde{N}_1, \ldots, \widetilde{N}_k)$$

such that

- $e \in \{ e_1, \ldots, e_k \}$ if $\exists i.\ e = e_i$;
- $e \in U(\widetilde{N}_1, \ldots, \widetilde{N}_k)$ if $\exists i.\ e \in \widetilde{N}_i$;
- $e \in F_{\triangleright\triangleleft}(\widetilde{N}_1, \ldots, \widetilde{N}_k)$ if $e = F(e_1, \ldots e_n) \wedge \forall i.\ e_i \in \widetilde{N}_i$.

For version space algebra $\widetilde{N}$, FlashMeta implements a clustering operator denoted as $\widetilde{N}|_\sigma$. $\widetilde{N}|_\sigma$ is a mapping from values to version space algebras $\{n_1 \mapsto \widetilde{N}_1, \ldots, n_k \mapsto \widetilde{N}_k\}$ such that

- $\widetilde{N} = \widetilde{N}_1 \cup \ldots \cup \widetilde{N}$;
- $\widetilde{N}_i \cap \widetilde{N}_j = \emptyset$ for all $i \neq j$;
- $\forall e \in \widetilde{N}_i.\ \langle e, \sigma \rangle \Downarrow n_i$;
- $\forall i, j.\ i \neq j \rightarrow n_i \neq n_j$.

In this context, the value-projection operator can be implemented as follows:

$$\Pi^{value}_{\sigma,n}(\widetilde{N}) := \widetilde{N}|_\sigma(n).$$

$$\text{VAR-LEFT-EXE}\ \frac{\mathit{Left}(v) \wedge x}{\langle v,\sigma,l,c,x\rangle \Downarrow_* \langle \sigma(v), l \cap c, \emptyset, \mathit{false}\rangle} \qquad \text{VAR-LEFT-NEXE}\ \frac{\mathit{Left}(v) \wedge \neg x}{\langle v,\sigma,l,c,x\rangle \Downarrow_* \langle \sigma(v), l \setminus c, \emptyset, x\rangle} \qquad \text{VAR-NLEFT}\ \frac{\neg \mathit{Left}(v)}{\langle v,\sigma,l,c,x\rangle \Downarrow_* \langle \sigma(v), l, c, x\rangle}$$

$$\text{OP}\ \frac{\langle e_1,\sigma,l,c,x\rangle \Downarrow_* \langle n_1,l_1,c_1,x_1\rangle \qquad \langle e_2,\sigma,l_1,c_1,x_1\rangle \Downarrow_* \langle n_1,l_1,c_1,x_1\rangle \qquad n_3 = n_1\ op\ n_2}{\langle e_1\ op\ e_2,\sigma,l,c,x\rangle \Downarrow_* \langle n_3,l_2,c_2,x_2\rangle}$$

$$\text{ASSIGN-INS}\ \frac{\mathit{Inserted}(v := e) \qquad \langle v := e,\sigma\rangle \Downarrow \sigma'}{\langle v := e,\sigma,l,c,x\rangle \Downarrow_* \langle \sigma', l, c, \mathit{true}\rangle} \qquad \text{ASSIGN-LR-EXE}\ \frac{\neg \mathit{Inserted}(v := e) \wedge (\mathit{Left}(v) \vee \mathit{Right}(v)) \wedge x \qquad \langle v := e,\sigma\rangle \Downarrow \sigma'}{\langle v := e,\sigma,l,c,x\rangle \Downarrow_* \langle \sigma', l \cap c, \emptyset, \mathit{false}\rangle}$$

$$\text{ASSIGN-LR-NEXE}\ \frac{\neg \mathit{Inserted}(v := e) \wedge (\mathit{Left}(v) \vee \mathit{Right}(v)) \wedge \neg x \qquad \langle v := e,\sigma\rangle \Downarrow \sigma'}{\langle v := e,\sigma,l,c,x\rangle \Downarrow_* \langle \sigma', l \setminus c, \emptyset, \mathit{false}\rangle}$$

$$\text{ASSIGN-NLR}\ \frac{\neg \mathit{Inserted}(v := e) \wedge \neg \mathit{Left}(v) \wedge \neg \mathit{Right}(v) \qquad \langle e,\sigma,l,c,x\rangle \Downarrow_* \langle n,l_1,c_1,x_1\rangle}{\langle v := e,\sigma,l,c,x\rangle \Downarrow_* \langle \sigma[v \mapsto n], l_1, c_1, x_1\rangle}$$

$$\text{SEQ}\ \frac{\langle s_1,\sigma,l,c,x\rangle \Downarrow_{value*} \langle \sigma_1,l_1,c_1,x_1\rangle \qquad \langle s_2,\sigma_1,l_1,c_1,x_1\rangle \Downarrow_{value*} \langle \sigma_2,l_2,c_2,x_2\rangle}{\langle s_1\ ;\ s_2,\sigma,l,c,x\rangle \Downarrow_* \langle \sigma_2,l_2,c_2,x_2\rangle} \qquad \text{SKIP}\ \frac{}{\langle \texttt{skip},\sigma,l,c,x\rangle \Downarrow_* \langle \sigma,l,c,x\rangle}$$

$$\text{IF-TRUE}\ \frac{\langle e,\sigma,l,c,x\rangle \Downarrow_* \langle \mathit{true},l_1,c_1,x_1\rangle \qquad \langle s_1,\sigma,l_1,c_1,x_1\rangle \Downarrow_{value*} \langle \sigma_1,l_2,c_2,x_2\rangle}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi},\sigma,c\rangle \Downarrow_* \langle \sigma_1,l_2,c_2,x_2\rangle}$$

$$\text{IF-FALSE}\ \frac{\langle e,\sigma,l,c,x\rangle \Downarrow_* \langle \mathit{false},l_1,c_1,x_1\rangle \qquad \langle s_2,\sigma,l_1,c_1,x_1\rangle \Downarrow_{value*} \langle \sigma_2,l_2,c_2,x_2\rangle}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi},\sigma,l,c,x\rangle \Downarrow_* \langle \sigma_2,l_2,c_2,x_2\rangle}$$

$$\text{WHILE-TRUE}\ \frac{\langle e,\sigma,l,c,x\rangle \Downarrow_* \langle \mathit{true},l_1,c_1,x_1\rangle \qquad \langle s_1,\sigma,l_1,c_1,x_1\rangle \Downarrow_{value*} \langle \sigma_1,l_2,c_2,x_2\rangle \qquad \langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma_1,l_2,c_2,x_2\rangle \Downarrow_{value*} \langle \sigma_2,l_3,c_3,x_3\rangle}{\langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma,l,c,x\rangle \Downarrow_* \langle \sigma_2,l_3,c_3,x_3\rangle}$$

$$\text{WHILE-FALSE}\ \frac{\langle e,\sigma,l,c,x\rangle \Downarrow_* \langle \mathit{false},l',c',x'\rangle}{\langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma,l,c,x\rangle \Downarrow_* \langle \sigma,l',c',x'\rangle} \qquad \text{NUM}\ \frac{\langle n,\sigma\rangle \Downarrow n}{\langle n,\sigma,l,c,x\rangle \Downarrow_* \langle n,l,c,x\rangle}$$

$$\text{STMT-L}\ \frac{\langle s,\sigma,l,c,x\rangle \Downarrow_* \langle \sigma',l',c',x'\rangle \qquad c'' = \lambda x.\ \text{if } x = s \text{ then } c'(x) \cap \Pi^{value}_{\sigma,n}(\varepsilon) \text{ else } c'(x)}{\langle s,\sigma,l,c,x\rangle \Downarrow_{value*} \langle \sigma',l',c'',x'\rangle}$$

Fig. 13. Augmented semantics of $\mathcal{L}$ for computing test-equivalence classes w.r.t. $\overset{t}{\sim}{}^*$. v—variables, n—integer values, e—expressions, s—statements, σ—program states, l, c—functions from locations to sets of expressions, x—Boolean values, *Inserted*—predicate over statements, *Left*, *Right*—predicates over variables, and ε—right-hand side of inserted assignment.

B SEMANTICS FOR COMPOSITION OF TEST-EQUIVALENCE RELATIONS

Section 3.5 presents a non-constructive definition of the composition of test-equivalence relations $\overset{t}{\sim}{}^*$ (it does not define an algorithm for computing test-equivalence partitions). Although it is possible to compute test-equivalence partitions w.r.t. $\overset{t}{\sim}{}^*$ by separately applying the value-based test-equivalence analysis in Figure 7 and dependency-based test-equivalence analysis in Figure 8, this approach would require performing multiple program executions. A more efficient approach is to compute test-equivalence partitions for $\overset{t}{\sim}{}^*$ directly thorough a combination of the semantics in Figure 7 and Figure 8 as shown in Figure 13. For a given program p with an assignment $v := \varepsilon$, this semantics identifies a mapping $\{\ l_1 \mapsto \mathcal{E}_1, \ldots, l_k \mapsto \mathcal{E}_k\ \}$ that denotes all test-equivalent insertions of assignments of $v := e'$ at each location l_i so that $e' \in \mathcal{E}_i$. In this semantics, l and c denote such mappings, $l \cap c$ denotes $\lambda x.\ l(x) \cup c(x)$ and $l \setminus c$ denotes $\lambda x.\ l(x) \setminus c(x)$.

For this semantics of $\overset{t}{\sim}{}^*$, test-equivalence analysis for the patch generation algorithm in Section 4 can be defined as follows.

$$Trace\Big(\frac{}{\langle v,\sigma\rangle \Downarrow \sigma(v)}\text{VAR}^i\Big) := \{\ \lambda\sigma.\ \sigma[x_i \mapsto \sigma(v)]\ \} \qquad Trace\Big(\frac{}{\langle n,\sigma\rangle \Downarrow n}\text{NUM}^i\Big) := \{\ \lambda\sigma.\ \sigma[x_i \mapsto n]\ \}$$

$$Trace\Big(\frac{\langle e_1,\sigma\rangle \Downarrow n_1 \qquad \langle e_2,\sigma\rangle \Downarrow n_2 \qquad n_3 = n_1\ op\ n_2}{\langle e_1\ op\ e_2,\sigma\rangle \Downarrow n_3}\text{OP}^i\Big) := Trace(\langle e_1,\sigma\rangle \Downarrow n_1)@Trace(\langle e_2,\sigma\rangle \Downarrow n_2)@\{\ \lambda\sigma.\ \sigma[x_i \mapsto \sigma(x_l)\ op\ \sigma(x_r)]\ \}$$

$$Trace\Big(\frac{}{\langle \texttt{skip},\sigma\rangle \Downarrow \sigma}\text{SKIP}^i\Big) := \{\ \lambda\sigma.\ \sigma\ \} \qquad Trace\Big(\frac{\langle e,\sigma\rangle \Downarrow n}{\langle v := e,\sigma\rangle \Downarrow \sigma[v \mapsto n]}\text{ASSIGN}^i\Big) := Trace(\langle e,\sigma\rangle \Downarrow n)@\{\ \lambda\sigma.\ \sigma[v \mapsto n]\ \}$$

$$Trace\Big(\frac{\langle s_1,\sigma\rangle \Downarrow \sigma_1 \qquad \langle s_2,\sigma_1\rangle \Downarrow \sigma_2}{\langle s_1\ ;\ s_2,\sigma\rangle \Downarrow \sigma_2}\text{SEQ}^i\Big) := Trace(\langle s_1,\sigma\rangle \Downarrow \sigma_1)@Trace(\langle s_2,\sigma_1\rangle \Downarrow \sigma_2)$$

$$Trace\Big(\frac{\langle e,\sigma\rangle \Downarrow true \qquad \langle s_1,\sigma\rangle \Downarrow \sigma_1}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi},\sigma\rangle \Downarrow \sigma_1}\text{IF-TRUE}^i\Big) := Trace(\langle e,\sigma\rangle \Downarrow true)@Trace(\langle s_1,\sigma\rangle \Downarrow \sigma_1)$$

$$Trace\Big(\frac{\langle e,\sigma\rangle \Downarrow false \qquad \langle s_2,\sigma\rangle \Downarrow \sigma_2}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi},\sigma\rangle \Downarrow \sigma_2}\text{IF-FALSE}^i\Big) := Trace(\langle e,\sigma\rangle \Downarrow false)@Trace(\langle s_2,\sigma\rangle \Downarrow \sigma_2)$$

$$Trace\Big(\frac{\langle e,\sigma\rangle \Downarrow true \qquad \langle s_1,\sigma\rangle \Downarrow \sigma_1 \qquad \langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma_1\rangle \Downarrow \sigma_2}{\langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma\rangle \Downarrow \sigma_2}\text{WHILE-TRUE}^i\Big) := Trace(\langle e,\sigma\rangle \Downarrow true)@Trace(\langle s_1,\sigma\rangle \Downarrow \sigma_1)@Trace(\langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma_1\rangle \Downarrow \sigma_2)$$

$$Trace\Big(\frac{\langle e,\sigma\rangle \Downarrow false}{\langle \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od},\sigma\rangle \Downarrow \sigma}\text{WHILE-FALSE}^i\Big) := Trace(\langle e,\sigma\rangle \Downarrow false)$$

Fig. 14. Definition of *Trace*. @—list concatenation operator, v—variables, n—integer values, e—expressions, s—statements, σ—program states, i—index of current rule application, l, r—indexes of child rules application (left and right), x_i—variable introduced for each rule application, and $Trace(\langle e_1,\sigma\rangle \Downarrow n_1)$ indicates an application of *Trace* to the derivation tree for $\langle e_1,\sigma\rangle \Downarrow n_1$.

Definition B.1 (Composed Test-Equivalence Analysis). Let $\mathcal{P}$ be a search space, $p \in \mathcal{P}$ be a program, $t = (\sigma_{in}, \phi)$ be a test. Let p' be a program, l_1 be a location such that $p = p'[l_1/\mathsf{v} := \mathsf{e};l_1]$ and $\exists p'' \in \mathcal{P}.\ \exists l_2 \in p''.\ \exists e'.\ p'' = p'[l_2/\mathsf{v} := \mathsf{e}';l_2]$. Then, dependency-based test-equivalence analysis $\widetilde{eval}$ is

$$\widetilde{eval}(p,t,\mathcal{P},\overset{\mathrm{t}}{\sim}_{deps}) = \left(\phi(\sigma_{out}), \bigcup_{l\in p}\bigcup_{e\in L'(l)}\{p'[l/\mathsf{v} := \mathsf{e};l]\}\right), \text{ given that}$$

$$Inserted := \lambda s.\ s = \text{“}v := e\text{”},\ Left := \lambda v'.\ v' = v,\ Right := \lambda v'.\ v' \in Var(e),$$

$$L := \lambda l.\ \{e \mid \exists p'' \in \mathcal{P}.\ p'' = p'[l/\mathsf{v} := \mathsf{e};l]\},$$

$$\langle p, \sigma_{in}, L, \lambda x.\ \emptyset, false\rangle \Downarrow_* \langle \sigma_{out}, L', \_, \_\rangle.$$

In this analysis, we identify a test-equivalence class of a program with an assignment $v := e'$ in the space of all programs in $\mathcal{P}$ that differ only in the expressions e' and the locations of this assignment. The test-equivalence class is computed by passing the “alternative” mapping from locations to expressions L as an argument of $\Downarrow_*$.

C PROPERTIES OF TEST-EQUIVALENCE SEMANTICS

We introduce a linear representation of a derivation tree that we refer to as a *trace*. A trace of a derivation tree D (for the semantics in Figure 6) is denoted as $Trace(D)$ (Figure 14). The trace of the derivation tree in Example 3.2 is a sequence of functions

$$\left\{\ f^0_{\text{VAR}},\ f^1_{\text{NUM}},\ f^2_{\text{OP}},\ f^3_{\text{ASSIGN}},\ f^4_{\text{VAR}},\ f^5_{\text{ASSIGN}},\ f^6_{\text{SEQ}}\ \right\},$$

where f^i_{RULE} is a *state transformation function* for the rule RULE with the index i in the derivation tree computed in depth-first order from left to right. Note that $\sigma_{out} = f^0_{\text{VAR}} \circ f^1_{\text{NUM}} \circ f^2_{\text{OP}} \circ f^3_{\text{ASSIGN}} \circ f^4_{\text{VAR}} \circ f^5_{\text{ASSIGN}} \circ f^6_{\text{SEQ}}(\sigma_{in})$ (by construction).

C.1 Value-Based Test-Equivalence Relation

This section describes properties and their proofs for the semantics of value-based test-equivalence relation in Figure 7. In an analogous manner to the previous section, we define the function *Trace* for the value-based semantics in such a way that it transforms each rule application into a function $f^i_{\text{RULE}} : \Sigma \times \mathcal{E} \to \Sigma \times \mathcal{E}$, where the seconds element of the tuple denotes the computed class of expressions.

Lemma C.1. *Let p be a program, e be an expression, $t := (\sigma_{in}, \phi)$ be a test, Modified $:= \lambda x.\ x = e$. If $\langle p, \sigma_{in}, C_{in}\rangle \Downarrow_{value} \langle \_, C_{out}\rangle$ (with the corresponding derivation tree D) and $e \in C_{out}$, then for each application of the rule EXPR-MOD in the derivation tree D with premises $\langle e, \sigma\rangle \Downarrow n$ and $c' = \Pi^{value}_{\sigma,n}(c)$, the expression e is in the set c'.*

Proof. The rule EXPR-MOD is either encountered in D or not. If it is not applied in the derivation tree, then $C_{out} = C_{in}$ since EXPR_MOD is the only rule that affects c. If it is applied in the derivation tree, represent the derivation tree as a trace:

$$\left\{ f^0_{\text{R\_0}},\ f^1_{\text{R\_1}}, \ldots,\ f^k_{\text{R\_k}} \right\}.$$

Then, $(\sigma_{out}, C_{out}) = f^0_{\text{R\_0}} \circ f^1_{\text{R\_1}} \circ \cdots \circ f^k_{\text{R\_k}}(\sigma_{in}, C_{in})$. Therefore, the set C_{out} can be represented as $\Pi^{value}_{\sigma_1,n_1} \circ \Pi^{value}_{\sigma_2,n_2} \circ \cdots \circ \Pi^{value}_{\sigma_m,n_m}(C_{in})$, where each application of the value-projector operator corresponds to an application of the rule EXPR-MOD. Since the value-projector operator outputs a subset of its argument set, $\forall i.\ e \in \Pi^{value}_{\sigma_i,n_i} \circ \cdots \circ \Pi^{value}_{\sigma_m,n_m}(C_{in})$. Therefore, $e \in c'$ for each application of the rule EXPR-MOD in D. □

Lemma C.2. *Let p be a program, e be an expression, $t := (\sigma_{in}, \phi)$ be a test, Modified $:= \lambda x.\ x = e$. If $\langle p, \sigma_{in}, C^1_{in}\rangle \Downarrow_{value} \langle \_, C^1_{out}\rangle$ and $\langle p, \sigma_{in}, C^2_{in}\rangle \Downarrow_{value} \langle \_, C^2_{out}\rangle$ and $C^1_{in} \subseteq C^2_{in}$, then $C^1_{out} \subseteq C^2_{out}$.*

Proof. It is trivial that for any n and σ if $C^1_{in} \subseteq C^2_{in}$, then $\Pi^{value}_{\sigma,n}(C^1_{in}) \subseteq \Pi^{value}_{\sigma,n}(C^2_{in})$. Then, the lemma follows from the fact that C_{out} can be represented as $\Pi^{value}_{\sigma_1,n_1} \circ \Pi^{value}_{\sigma_2,n_2} \circ \cdots \circ \Pi^{value}_{\sigma_k,n_k}(C_{in})$. □

Lemma C.3. *Let p_1 and p_2 be programs such that $\exists e, e' \in Expr.\ p_2 = p_1[e/e']$, $t := (\sigma_{in}, \phi)$ be a test. If $p_1 \overset{t}{\sim}_{value} p_2$ and $\langle p_1, \sigma_{in}, \{e, e'\}\rangle \Downarrow_{value} \langle \sigma_{out}, \{e, e'\}\rangle$ (for Modified $:= \lambda x.\ x = e$), then $\langle p_2, \sigma_{in}, \{e, e'\}\rangle \Downarrow_{value} \langle \sigma_{out}, \{e, e'\}\rangle$ (for Modified $:= \lambda x.\ x = e'$).*

Proof. $\langle p_1, \sigma_{in}, \{e, e'\}\rangle \Downarrow_{value} \langle \sigma_{out}, \{e, e'\}\rangle$ for *Modified* $:= \lambda x.\ x = e$ with the corresponding derivation tree D. Therefore, in each application of the rule EXPR-MOD of the derivation tree, $c' = \{e, e'\}$ (Lemma C.1). Consequently, in each occurrence of EXPR-MOD, $\langle e, \sigma\rangle \Downarrow n \Leftrightarrow \langle e', \sigma\rangle \Downarrow n$. Assume that D' is obtained by replacing all occurrences of e with e' in the derivation tree D. Since $\langle e, \sigma\rangle \Downarrow n \Leftrightarrow \langle e', \sigma\rangle \Downarrow n$ in each application of the rule EXPR-MOD, D' is a valid derivation tree for $\langle p_2, \sigma_{in}, \{e, e'\}\rangle \Downarrow_{value} \langle \sigma_{out}, \{e, e'\}\rangle$ with *Modified* $:= \lambda x.\ x = e'$. □

Lemma C.4. *Let p be program, $t := (\sigma_{in}, \phi)$ be a test. If $\langle p, \sigma_{in}, \_\rangle \Downarrow_{value} \langle \sigma_{out}, \_\rangle$, then $\langle p, \sigma_{in}\rangle \Downarrow \sigma_{out}$.*

Proof. A derivation tree for the semantics in Figure 7 can be transformed into a valid derivation tree of the standard semantics (Figure 6) by removing the argument c and changing the applications of EXPR-MOD and EXPR-NMOD into corresponding applications of EXPR. □

Proposition C.5. *Let $\mathcal{P}$ be a set of programs such that for any $p_1, p_2 \in \mathcal{P}$, $\exists e, e' \in Expr$ such that $p_2 = p_1[e/e']$, $t := (\sigma_{in}, \phi)$ be a test. Then, $\overset{t}{\sim}_{value}$ as in Definition 3.4 is an equivalence relation in $\mathcal{P}$.*

Proof. **Reflexivity.** Let p be a program, e be an expression in p, *Modified* $:= \lambda x.\ x = e$. Since $\forall e.\ \langle e, \sigma\rangle \Downarrow n \Leftrightarrow \{e\} = \Pi^{value}_{\sigma,n}(\{e\})$, $c' = \{e\}$ in each application of EXPR-MOD for $c = \{e\}$.

Therefore, $\langle p_1, \sigma_{in}, \{e\}\rangle \Downarrow_{value} \langle \_, \{e\}\rangle$. Consequently, $p \overset{t}{\sim}_{value} p$. **Symmetry.** Let p_1, p_2 be programs and e_1, e_2 be expressions such that $p_2 = p_1[e_1/e_2]$ and $p_1 \overset{t}{\sim}_{value} p_2$. Assume $\langle p_1, \sigma_{in}, \{e, e'\}\rangle \Downarrow_{value} \langle \sigma_{out}, \{e, e'\}\rangle$. Then, $\langle p_2, \sigma_{in}, \{e, e'\}\rangle \Downarrow_{value} \langle \sigma_{out}, \{e, e'\}\rangle$ (Lemma C.3). Therefore, $p_2 \overset{t}{\sim}_{value} p_1$. **Transitivity.** Let p_1, p_2, p_3 be programs and e_1, e_2, e_3 be expressions such that $p_2 = p_1[e_1/e_2]$ and $p_3 = p_1[e_1/e_3]$. Assume $p_1 \overset{t}{\sim}_{value} p_2$ and $p_1 \overset{t}{\sim}_{value} p_3$. Note that for any σ, if $\langle e_1, \sigma\rangle \Downarrow n \wedge c' = \Pi^{value}_{\sigma,n}(\{e_1, e_2, e_3\})$, then $e_2 \in c' \wedge e_3 \in c'$ (Lemma C.1 and Lemma C.2). Thus, $\langle e_1, \sigma\rangle \Downarrow n \Leftrightarrow \{e_1, e_2, e_3\} = \Pi^{value}_{\sigma,n}(\{e_1, e_2, e_3\})$. Therefore, $\forall\sigma.\ \langle e_2, \sigma\rangle \Downarrow n \Leftrightarrow \{e_2, e_3\} = \Pi^{value}_{\sigma,n}(\{e_2, e_3\})$. Consequently, $\langle p_2, \sigma_{in}, \{e_2, e_3\}\rangle \Downarrow_{value} \langle \_, \{e_2, e_3\}\rangle$ for $Modified := \lambda x.\ x = e_2$ and $p_2 \overset{t}{\sim}_{value} p_3$. □

Proposition C.5 can be trivially generalized for spaces involving modifications of multiple expressions since the union of several equivalence relations is an equivalence relation.

Proposition C.6. *Let p_1 and p_2 be programs such that $\exists e, e' \in Expr.\ p_2 = p_1[e/e']$, $t := (\sigma_{in}, \phi)$ be a test. If $p_1 \overset{t}{\sim}_{value} p_2$, then p_1 and p_2 either both pass t or both fail t.*

Proof. Assume that $\langle p_1, \sigma_{in}, \{e, e'\}\rangle \Downarrow_{value} \langle \sigma_{out}, \{e, e'\}\rangle$ for $Modified := \lambda x.\ x = e$. Therefore, $\langle p_2, \sigma_{in}, \{e, e'\}\rangle \Downarrow_{value} \langle \sigma_{out}, \{e, e'\}\rangle$ for $Modified := \lambda x.\ x = e'$ (Lemma C.3), $\langle p_1, \sigma_{in}\rangle \Downarrow \sigma_{out}$ (Lemma C.4), and $\langle p_2, \sigma_{in}\rangle \Downarrow \sigma_{out}$ (Lemma C.4). Since p_1 and p_2 produce the same output state for σ_{in}, they either both pass t or both fail t. □

Proposition C.7. *The relation $\overset{t}{\sim}_{value}$ is a test-equivalence relation according to Definition 1.1.*

Proof. Proposition C.5 states that $\overset{t}{\sim}_{value}$ is an equivalence relation and Proposition C.6 states that any two programs equivalent w.r.t. $\overset{t}{\sim}_{value}$ produce indistinguishable results for the given test. Therefore, $\overset{t}{\sim}_{value}$ is a test-equivalence relation. □

C.2 Dependency-Dased Test-Equivalence Relation

This section describes properties and their proofs for the semantics of dependency-based test-equivalence relation in Figure 8. In an analogous manner to the previous section, we define the function *Trace* for the dependency-based semantics in such a way that it transforms each rule application into a function $f^i_{\text{RULE}} : \Sigma \times Stmt \times Stmt \times \mathbb{B} \to \Sigma \times Stmt \times Stmt \times \mathbb{B}$, where the second and the third elements of the tuple denote global and current test-equivalence classes, and the last Boolean element indicates if the inserted assignment has been executed.

Lemma C.8. *Let p be a program, e be an expression and v be a variable, such that the assignment "$v := e$" is at the location s of p, $Inserted := \lambda s.\ s =$ "$v := e$", $Left := \lambda v'.\ v' = v$, $Right := \lambda v'.\ v' \in Var(e)$. Then, $\langle p, \sigma_{in}, \{s\}, \emptyset, false\rangle \Downarrow_{deps} \langle \_, \{s\}, \_, \_\rangle$.*

Proof. Consider a trace of the derivation tree of the relation $\langle p, \sigma_{in}, \{s\}, \emptyset, false\rangle \Downarrow_{deps} \langle \_, \_, \_, \_\rangle$:

$$\{\ f_{\text{R\_0}},\ f_{\text{R\_1}}, \ldots,\ f_{\text{R\_k}}\ \}.$$

For each i, if $(\_, l, c, x) = f_{\text{R\_i}} \circ \cdots \circ f_{\text{R\_k}}(\sigma_{in}, \{s\}, \emptyset, false)$, then $x = true \Leftrightarrow s \in c$ since x is set to *true* only when the rule ASSIGN-INS is applied (that adds s to c), and x is set to *false* in the rules VAR-LEFT-EXE, VAR-LEFT-NEXE, ASSIGN-LR-EXE, and ASSIGN-LR-NEXE (that make $c = \emptyset$). Therefore, when the rules VAR-LEFT-NEXE and ASSIGN-LR-NEXE are applied, $s \notin c$. Thus, s is never removed from l in the derivation tree. Therefore, $\langle p, \sigma_{in}, \{s\}, \emptyset, false\rangle \Downarrow_{deps} \langle \_, \{s\}, \_, \_\rangle$. □

Assume that $\{\ f_{\text{R\_0}},\ f_{\text{R\_1}}, \ldots,\ f_{\text{R\_k}}\ \}$ is a trace of some program p in a derivation tree for the semantics in Figure 8. We call a sequence of rule applications $I := \{f_{\text{R\_l}}, \ldots, f_{\text{R\_r}}\}$ a *test-equivalent interval* iff

— it does not include the rules VAR-LEFT-EXE, VAR-LEFT-NEXE, ASSIGN-LR-EXE, and ASSIGN-LR-NEXE;
— it contains an application of the rule ASSIGN-INS;
— it is maximal (not a sub-interval of another test-equivalent interval).

Note that each application of the rule ASSIGN-INS is included into a test-equivalent interval (possibly consisting only of this rule). We enumerate all test-equivalent intervals in the trace as $I_0, I_1, \ldots, I_k$ from left to right.

Lemma C.9. *Let p_1 and p_2 be programs, e be an expression, and v be a variable, such that there is program p with locations l_1, l_2 so that $p_1 = p[l_1/\mathsf{v} := \mathsf{e};l_1]$, $p_2 = p[l_2/\mathsf{v} := \mathsf{e};l_2]$, Inserted $:= \lambda s.\ s =$ "$v := e''$, Left $:= \lambda v'.\ v' = v$, Right $:= \lambda v'.\ v' \in Var(e)$, $p_1 \overset{t}{\sim}_{deps} p_2$. Assume that $I_0^1, I_1^1, \ldots, I_k^1$ are the test-equivalent intervals in the trace of p_1 for a derivation tree $\langle p_1, \sigma_{in}, \_, \_, \_\rangle \Downarrow_{deps} \langle \_, \_, \_, \_\rangle$. Then, for each I_i^1 there is a corresponding test-equivalent interval I_i^2 in the trace of $\langle p_2, \sigma_{in}, \_, \_, \_\rangle \Downarrow_{deps} \langle \_, \_, \_, \_\rangle$ such the first rules in the intervals receive the same inputs state for the same program location and the last rules in the intervals produce the same state for the same program location in the trace.*

Proof. Assume $I_i^1 := \{f^1_{\mathrm{R\_0}}, \ldots, f^1_{\mathrm{R\_k}}\}$ and $I_i^2 := \{f^2_{\mathrm{R\_0}}, \ldots, f^2_{\mathrm{R\_l}}\}$. We prove that for any state σ, if $(\sigma', \_, \_, \_) = f^1_{\mathrm{R\_0}} \circ \ldots \circ f^1_{\mathrm{R\_k}}(\sigma, \_, \_, \_)$ and the rules $f^1_{\mathrm{R\_0}}$ and $f^2_{\mathrm{R\_0}}$ are applied for the same program location, then $(\sigma', \_, \_, \_) = f^2_{\mathrm{R\_0}} \circ \ldots \circ f^2_{\mathrm{R\_l}}(\sigma, \_, \_, \_)$ and $f^1_{\mathrm{R\_l}}$ and $f^2_{\mathrm{R\_r}}$ are applied for the same program location. This fact is proved by induction on the rule index in the test-equivalent interval, excluding the applications of ASSIGN-INS. Then, this fact can be used to prove the lemma by induction on the index of the test-equivalent interval. □

Lemma C.10. *Let p be program, $t := (\sigma_{in}, \phi)$ be a test. If $\langle p, \sigma_{in}, \_, \_, \_\rangle \Downarrow_{deps} \langle \sigma_{out}, \_, \_, \_\rangle$, then $\langle p, \sigma_{in}\rangle \Downarrow \sigma_{out}$.*

Proof. A derivation tree for the semantics in Figure 8 can be transformed into a valid derivation tree of the standard semantics (Figure 6) by removing the arguments l, c, and x, changing the applications of VAR-LEFT-EXE, VAR-LEFT-NEXE, and VAR-NLEFT into corresponding applications of VAR and the applications of ASSIGN-INS, ASSIGN-LR-EXE, ASSIGN-LR-NEXE, and ASSIGN-NLR into corresponding applications of ASSIGN. □

Proposition C.11. *Let $\mathcal{P}$ be a set of programs, e be an expression, and v be a variable, such that for any $p_1, p_2 \in \mathcal{P}$, there is program p with locations l_1, l_2 such that $p_1 = p[l_1/\mathsf{v} := \mathsf{e};l_1]$ and $p_2 = p[l_2/\mathsf{v} := \mathsf{e};l_2]$, $t := (\sigma_{in}, \phi)$ be a test. Then, $\overset{t}{\sim}_{deps}$ as in Definition 3.7 is an equivalence relation in $\mathcal{P}$.*

Proof. **Reflexivity.** Let $p \in \mathcal{P}$ be a program, s is the location of "$v := e$" in p, *Inserted* $:= \lambda s.\ s =$ "$v := e$", *Left* $:= \lambda v'.\ v' = v$, *Right* $:= \lambda v'.\ v' \in Var(e)$. $\langle p, \sigma_{in}, \{s\}, \emptyset, false\rangle \Downarrow_{deps} \langle \_, \{s\}, \_, \_\rangle$ (Lemma C.8). Consequently, $p \overset{t}{\sim}_{deps} p$. **Symmetry.** Let p_1, p_2 be programs such that there is program p with locations l_1, l_2 such that $p_1 = p[l_1/\mathsf{v} := \mathsf{e};l_1]$, $p_2 = p[l_2/\mathsf{v} := \mathsf{e};l_2]$, and $p_1 \overset{t}{\sim}_{deps} p_2$. Since $\langle p_1, \sigma_{in}, \{l_1, l_2\}, \emptyset, false\rangle \Downarrow_{deps} \langle \_, \{l_1, l_2\}, \_, \_\rangle$, $\langle p_2, \sigma_{in}, \{l_1, l_2\}, \emptyset, false\rangle \Downarrow_{deps} \langle \_, \{l_1, l_2\}, \_, \_\rangle$ (Lemma C.9). Therefore, $p_2 \overset{t}{\sim}_{deps} p_1$. **Transitivity.** Let p_1, p_2, p_3 be programs such that there is program p with locations l_1, l_2, l_3 such that $p_1 = p[l_1/\mathsf{v} := \mathsf{e};l_1]$, $p_2 = p[l_2/\mathsf{v} := \mathsf{e};l_2]$ and $p_3 = p[l_3/\mathsf{v} := \mathsf{e};l_3]$, and $p_1 \overset{t}{\sim}_{deps} p_2$, $p_1 \overset{t}{\sim}_{deps} p_3$. Therefore, $\langle p_1, \sigma_{in}, \{l_1, l_2, l_3\}, \emptyset, false\rangle \Downarrow_{deps} \langle \_, \{l_1, l_2, l_3\}, \_, \_\rangle$. Therefore, the locations $\{l_1, l_2, l_3\}$ belong to each test-equivalent interval of the corresponding trace. Consequently, $\langle p_2, \sigma_{in}, \{l_2, l_3\}, \emptyset, false\rangle \Downarrow_{deps} \langle \_, \{l_2, l_3\}, \_, \_\rangle$. Therefore, $p_2 \overset{t}{\sim}_{deps} p_3$. □

Proposition C.12. *Let p_1 and p_2 be programs such that there is program p with locations l_1, l_2 such that $p_1 = p[l_1/\mathsf{v} := \mathsf{e};l_1]$ and $p_2 = p[l_2/\mathsf{v} := \mathsf{e};l_2]$, $t := (\sigma_{in}, \phi)$ be a test. If $p_1 \overset{t}{\sim}_{deps} p_2$, then p_1 and p_2 either both pass t or both fail t.*

Proof. Without loss of generality we assume that the last statement of p_1 and p_2 is $v := v$. Then, p_1 and p_2 produce the same output states for the inputs σ_{in} (Lemma C.9 and Lemma C.10). Therefore, p_1 and p_2 either both pass t or both fail t. □

Proposition C.13. *The relation $\overset{t}{\sim}_{deps}$ is a test-equivalence relation according to Definition 1.1.*

Proof. Proposition C.11 states that $\overset{t}{\sim}_{deps}$ is an equivalence relation and Proposition C.12 states that any two programs equivalent w.r.t. $\overset{t}{\sim}_{deps}$ produce indistinguishable results for the given test. Therefore, $\overset{t}{\sim}_{deps}$ is a test-equivalence relation. □

C.3 Composition of Test-Equivalence Relations

This section describes properties and their proofs for the semantics in Figure 13. In an analogous manner to the previous sections, we define the function *Trace* for the dependency-based semantics in such a way that it transforms each rule application into a function $f^i_{\text{RULE}} : \Sigma \times (Stmt \to 2^{Expr}) \times (Stmt \to 2^{Expr}) \times \mathbb{B} \to \Sigma \times (Stmt \to 2^{Expr}) \times (Stmt \to 2^{Expr}) \times \mathbb{B}$, where the second and the third elements of the tuple denote global and current test-equivalence classes, and the last Boolean element indicates if the inserted assignment has been executed.

Lemma C.14. *Let p be a program, e be an expression, and v be a variable, such that the assignment "$v := e$" is at the location s of p, Inserted $:= \lambda s.\ s =$ "$v := e$", Left $:= \lambda v'.\ v' = v$, Right $:= \lambda v'.\ v' \in Var(e)$. Then, $\langle p, \sigma_{in}, l_{in}, \lambda x.\ \emptyset, false\rangle \Downarrow_* \langle \_, l_{in}, \_, \_\rangle$, where $l_{in} := \lambda x.$ if $x = s$ then $\{e\}$ else $\emptyset$.*

Proof. Consider a trace of the derivation tree of the relation $\langle p, \sigma_{in}, l_{in}, \lambda x.\ \emptyset, false\rangle \Downarrow_* \langle \_, \_, \_, \_\rangle$:

$$\{\ f_{\text{R\_0}},\ f_{\text{R\_1}}, \ldots,\ f_{\text{R\_k}}\ \}.$$

For each i, if $(\_, l, c, x) = f_{\text{R\_i}} \circ \cdots \circ f_{\text{R\_k}}(\sigma_{in}, l_{in}, \lambda x.\ \emptyset, false)$, then $x = true \Leftrightarrow e \in c(s)$ since x is set to *true* only when the rule ASSIGN-INS is applied (that updates c for the argument s), and x is set to *false* in the rules VAR-LEFT-EXE, VAR-LEFT-NEXE, ASSIGN-LR-EXE, and ASSIGN-LR-NEXE (that make $c = \lambda x.\ \emptyset$). Therefore, when the rules VAR-LEFT-NEXE and ASSIGN-LR-NEXE are applied, $e \notin c(s)$. Thus, $l(s)$ always includes at least e in the derivation tree. Therefore, $\langle p, \sigma_{in}, l_{in}, \lambda x.\ \emptyset, false\rangle \Downarrow_* \langle \_, l_{in}, \_, \_\rangle$. □

Lemma C.15. *Let p_1 and p_2 be programs, e_1 and e_2 be expressions, and v be a variable, such that there is program p with locations l_1, l_2 so that $p_1 = p[l_1/\mathsf{v} := \mathsf{e}_1;l_1]$, $p_2 = p[l_2/\mathsf{v} := \mathsf{e}_2;l_2]$, Inserted $:= \lambda s.\ s =$ "$v := e$", Left $:= \lambda v'.\ v' = v$, Right $:= \lambda v'.\ v' \in Var(e)$, $p_1 \overset{t}{\sim}_* p_2$. Assume that $I^1_0, I^1_1, \ldots, I^1_k$ are the test-equivalent intervals in the trace of p_1 for a derivation tree $\langle p_1, \sigma_{in}, \_, \_, \_\rangle \Downarrow_* \langle \_, \_, \_, \_\rangle$. Then, for each I^1_i there is a corresponding test-equivalent interval I^2_i in the trace of $\langle p_2, \sigma_{in}, \_, \_, \_\rangle \Downarrow_* \langle \_, \_, \_, \_\rangle$ such that the first rules in the intervals receive the same inputs state for the same program location and the last rules in the intervals produce the same state for the same program location in the trace.*

Proof. Assume $I^1_i := \{f^1_{\text{R\_0}}, \ldots, f^1_{\text{R\_k}}\}$ and $I^2_i := \{f^2_{\text{R\_0}}, \ldots, f^2_{\text{R\_l}}\}$. We prove that for any state σ, if $(\sigma', \_, \_, \_) = f^1_{\text{R\_0}} \circ \cdots \circ f^1_{\text{R\_k}}(\sigma, \_, \_, \_)$ and the rules $f^1_{\text{R\_0}}$ and $f^2_{\text{R\_0}}$ are applied for the same program location, then $(\sigma', \_, \_, \_) = f^2_{\text{R\_0}} \circ \cdots \circ f^2_{\text{R\_l}}(\sigma, \_, \_, \_)$ and $f^1_{\text{R\_l}}$ and $f^2_{\text{R\_r}}$ are applied for the same program location. This fact is proved by induction on the rule index in the test-equivalent

interval, excluding the applications of ASSIGN-INS. Then, this fact can be used to prove the lemma by induction on the index of the test-equivalent interval. □

Lemma C.16. *Let p be program, $t := (\sigma_{in}, \phi)$ be a test. If $\langle p, \sigma_{in}, \_, \_, \_\rangle \Downarrow_* \langle \sigma_{out}, \_, \_, \_\rangle$, then $\langle p, \sigma_{in}\rangle \Downarrow \sigma_{out}$.*

Proof. A derivation tree for the semantics in Figure 13 can be transformed into a valid derivation tree of the standard semantics (Figure 6) by removing the arguments l, c, and x, changing the applications of VAR-LEFT-EXE, VAR-LEFT-NEXE, and VAR-NLEFT into corresponding applications of VAR and the applications of ASSIGN-INS, ASSIGN-LR-EXE, ASSIGN-LR-NEXE, and ASSIGN-NLR into corresponding applications of ASSIGN, removing applications of STMT-L. □

Proposition C.17. *Let $\mathcal{P}$ be a set of programs, e be an expression, and v be a variable, such that for any $p_1, p_2 \in \mathcal{P}$, there is program p with locations l_1, l_2 and expressions e_1 and e_2 such that $p_1 = p[l_1/\mathsf{v} := \mathsf{e}_1;l_1]$ and $p_2 = p[l_2/\mathsf{v} := \mathsf{e}_2;l_2]$, $t := (\sigma_{in}, \phi)$ are a test. Then, $\overset{\mathrm{t}}{\sim}_*$ as in Definition 3.7 is an equivalence relation in $\mathcal{P}$.*

Proof. **Reflexivity.** Let $p \in \mathcal{P}$ be a program, s is the location of "$v := e$" in p, *Inserted* := $\lambda s.\ s =$ "$v := e$", *Left* := $\lambda v'.\ v' = v$, *Right* := $\lambda v'.\ v' \in Var(e)$. $\langle p, \sigma_{in}, l_{in}, \lambda x.\ \emptyset, false\rangle \Downarrow_* \langle \_, l_{in}, \_, \_\rangle$ where $l_{in} := \lambda x.$ If $x = s$, then $\{e\}$ else $\emptyset$ (Lemma C.14). Consequently, $p \overset{\mathrm{t}}{\sim}_* p$. **Symmetry.** Let p_1, p_2 be programs and e_1 and e_2 be expressions such that there is program p with locations l_1, l_2 such that $p_1 = p[l_1/\mathsf{v} := \mathsf{e}_1;l_1]$, $p_2 = p[l_2/\mathsf{v} := \mathsf{e}_2;l_2]$, and $p_1 \overset{\mathrm{t}}{\sim}_* p_2$. Since $\langle p_1, \sigma_{in}, \{l_1 \mapsto \{e_1\}, l_2 \mapsto \{e_2\}, \_ \mapsto \emptyset\}, \emptyset, false\rangle \Downarrow_* \langle \_, \{l_1 \mapsto \{e_1\}, l_2 \mapsto \{e_2\}, \_ \mapsto \emptyset\}, \_, \_\rangle$, $\langle p_2, \sigma_{in}, \{l_1 \mapsto \{e_1\}, l_2 \mapsto \{e_2\}, \_ \mapsto \emptyset\}, \emptyset, false\rangle \Downarrow_* \langle \_, \{l_1 \mapsto \{e_1\}, l_2 \mapsto \{e_2\}, \_ \mapsto \emptyset\}, \_, \_\rangle$ (Lemma C.15). Therefore, $p_2 \overset{\mathrm{t}}{\sim}_* p_1$. **Transitivity.** Let p_1, p_2, p_3 be programs and e_1, e_2, e_3 be expressions such that there is program p with locations l_1, l_2, l_3 such that $p_1 = p[l_1/\mathsf{v} := \mathsf{e}_1;l_1]$, $p_2 = p[l_2/\mathsf{v} := \mathsf{e}_2;l_2]$ and $p_3 = p[l_3/\mathsf{v} := \mathsf{e}_3;l_3]$, and $p_1 \overset{\mathrm{t}}{\sim}_* p_2$, $p_1 \overset{\mathrm{t}}{\sim}_* p_3$. Therefore, $\langle p_1, \sigma_{in}, \{l_1 \mapsto \{e_1\}, l_2 \mapsto \{e_2\}, l_3 \mapsto \{e_3\}, \_ \mapsto \emptyset\}, \emptyset, false\rangle \Downarrow_* \langle \_, \{l_1 \mapsto \{e_1\}, l_2 \mapsto \{e_2\}, l_3 \mapsto \{e_3\}, \_ \mapsto \emptyset\}, \_, \_\rangle$. Therefore, the locations $\{l_1, l_2, l_3\}$ belong to each test-equivalent interval of the corresponding trace. Consequently, $\langle p_2, \sigma_{in}, \{l_2 \mapsto \{e_2\}, l_3 \mapsto \{e_3\}, \_ \mapsto \emptyset\}, \emptyset, false\rangle \Downarrow_* \langle \_, \{l_2 \mapsto \{e_2\}, l_3 \mapsto \{e_3\}, \_ \mapsto \emptyset\}, \_, \_\rangle$. Therefore, $p_2 \overset{\mathrm{t}}{\sim}_* p_3$ □

Proposition C.18. *Let p_1 and p_2 be programs and e_1 and e_2 be expressions such that there is program p with locations l_1, l_2 such that $p_1 = p[l_1/\mathsf{v} := \mathsf{e}_1;l_1]$ and $p_2 = p[l_2/\mathsf{v} := \mathsf{e}_2;l_2]$, $t := (\sigma_{in}, \phi)$ be a test. If $p_1 \overset{\mathrm{t}}{\sim}_* p_2$, then p_1 and p_2 either both pass t or both fail t.*

Proof. Without loss of generality, we assume that the last statement of p_1 and p_2 is $v := v$. Then, p_1 and p_2 produce the same output states for the inputs σ_{in} (Lemma C.15 and Lemma C.16). Therefore, p_1 and p_2 either both pass t or both fail t. □

Proposition C.19. *The relation $\overset{\mathrm{t}}{\sim}_*$ is a test-equivalence relation according to Definition 1.1.*

Proof. Proposition C.17 states that $\overset{\mathrm{t}}{\sim}_*$ is an equivalence relation and Proposition C.18 states that any two programs equivalent w.r.t. $\overset{\mathrm{t}}{\sim}_*$ produce indistinguishable results for the given test. Therefore, $\overset{\mathrm{t}}{\sim}_*$ is a test-equivalence relation. □

ACKNOWLEDGMENTS

The authors would like to thank Julia Lawall for her comments on earlier drafts of this article. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*. IEEE, 1–8.

[2] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *CEC*. IEEE, 162–168.

[3] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *CAV*. Springer, 383–401.

[4] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *PLDI*. ACM, 213–223.

[5] Jinru Hua and Sarfraz Khurshid. 2016. A sketching-based approach for debugging using test cases. In *ATVA*. Springer, 463–478.

[6] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *ICSE*. 215–224.

[7] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program repair as a game. In *CAV*. Springer, 226–238.

[8] René Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *ISSTA*. ACM, 315–326.

[9] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*. ACM, 216–226.

[10] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*. ACM, 386–399.

[11] Xuan Bach Dinh Le, Duc Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and semantic-guided repair synthesis via programming by example. In *FSE*. ACM.

[12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each. In *ICSE*. IEEE, 3–13.

[13] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[14] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *TSE* 38, 1 (2012), 54–72.

[15] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *FSE*. ACM, 166–178.

[16] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *ICSE*. ACM, 702–713.

[17] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*. ACM, 298–312.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*. ACM, 190–200.

[19] Yu-Seung Ma and Sang-Woon Kim. 2016. Mutation testing cost reduction by clustering overlapped mutants. *Journal of Systems and Software* 115 (2016), 18–30.

[20] Sergey Mechtaev, Manh-Dung Nguyen, Lars Noller, Yannic Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *ICSE*.

[21] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *ICSE*. IEEE, 448–458.

[22] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*.

[23] Tom M. Mitchell. 1982. Generalization as search. *Artificial intelligence* 18, 2 (1982), 203–226.

[24] Elfurjani S. Mresa and Leonardo Bottaci. 1999. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability* 9, 4 (1999), 205–232.

[25] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *ICSE*. 772–781.

[26] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. *OOPSLA* (2015), 107–126.

[27] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *ICSE*. ACM, 254–265.

[28] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*. ACM, 24–36.

[29] Alex Reinking and Ruzica Piskac. 2015. A type-directed approach to program repair. In *Computer Aided Verification (1)*. 511–517.
[30] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *ICSE*. IEEE Press, 404–415.
[31] David Shriver, Sebastian Elbaum, and Kathryn T. Stolee. 2017. At the end of synthesis: Narrowing program candidates. In *ICSE NIER*. IEEE Press, 19–22.
[32] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *FSE*. ACM, 532–543.
[33] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. University of California, Berkeley.
[34] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *OOPSLA*. ACM, 849–863.
[35] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated repair of software regressions. In *ICSE*. IEEE, 471–482.
[36] Shin Hwei Tan, Abhik Roychoudhury, Zhen Dong, and Xiang Gao. 2018. Repairing crashes in android apps. In *ICSE*.
[37] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. 2017. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *ICSE Companion*. 180–182.
[38] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE*. ACM, 727–738.
[39] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically generated patches as debugging aids: A human study. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 64–74.
[40] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster mutation analysis via equivalence modulo states. *ISSTA*.
[41] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *ISSTA*. ACM, 61–72.
[42] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*. IEEE, 356–366.
[43] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE*. IEEE Press, 416–426.
[44] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime ClÃl'ment, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering*.
[45] Jooyong Yi, Umair Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *FSE*. ACM.

Received October 2017; revised May 2018; accepted July 2018