Automatic Inference of Code Transforms and Search Spaces for Automatic Patch Generation Systems

Fan Long, Peter Amidon, and Martin Rinard
MIT EECS and MIT CSAIL
fanl@csail.mit.edu, peter@picnicpark.org, rinard@csail.mit.edu

Abstract

We present a new system, Genesis, that processes sets of human patches to automatically infer code transforms and search spaces for automatic patch generation. We present results that characterize the effectiveness of the Genesis inference algorithms and the resulting complete Genesis patch generation system working with real-world patches and errors collected from top 1000 github Java software development projects. To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from successful patches.

1. Introduction

Automatic patch generation systems [33, 38–41, 45, 54, 61, 64, 65] hold out the promise of significantly reducing the human effort required to diagnose, debug, and fix software errors. The standard *generate and validate* approach starts with a set of test cases, at least one of which exposes the error. It deploys a set of *transforms* to generate a *search space* of *candidate patches*, then runs the resulting patched programs on the test cases to find *plausible patches* that produce correct outputs for all test cases.

All previous generate and validate systems work with a set of manually crafted transforms [38–41, 54, 61, 64, 65]. This approach limits the system to fixing only those bugs that fall within the scope of the transforms that the developers of the patch generation system decided to provide. This limitation is especially unfortunate given the widespread availability (in open-source software repositories) of patches developed by many different human developers. Together, these patches embody a rich variety of different patching strategies developed by a wide range of human developers, and not just the patch generation strategies encoded in a set of manually crafted transforms from the developers of the patch generation system.

1.1 Genesis

We present Genesis, a novel system that automatically infers transforms and resulting search spaces for automatic patch generation systems. Given a set of successful human patches drawn from available revision histories, Genesis automatically generalizes subsets of patches to infer transforms that together generate a productive search space of candidate patches. Genesis can therefore leverage the combined patch generation expertise of many different developers to capture a wide range of productive patch generation strategies. It can then automatically apply the resulting transforms to successfully correct errors in multiple previously unseen applications. To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from successful patches.

Transforms: Each Genesis transform has two *template abstract syntax trees (ASTs)*. One template AST matches code in the original program. The other template AST specifies the replacement code for the generated patch. Template ASTs contain *template variables*, which match subtrees or subforests in the original or patched code. Template variables enable the transforms to abstract away patch- or application-specific details to capture common patch patterns implemented by multiple patches drawn from different applications.

Generators: Many useful patches do not simply rearrange existing code and logic; they also introduce new code and logic. Genesis transforms therefore implement *partial pattern matching* in which the template AST for the patch contains free template variables that are not matched in the original code. Each of the free template variables is associated with a *generator*, which systematically generates new candidate code components for the free variable. This new technique, which enables Genesis to synthesize new code and logic in the candidate patches, is essential to enabling Genesis to generate correct patches.

Search Space Inference with ILP: A key challenge in patch search space design is navigating an inherent tradeoff between coverage and tractability [42]. On one hand, the search space needs to be large enough to contain correct patches for the target class of errors (coverage). On the other hand, the search space needs to be small enough so that the patch generation system can efficiently explore the space to find the correct patches (tractability) [42].

Genesis navigates this tradeoff by formulating an integer linear program (ILP) whose solution maximizes the number of training patches covered by the inferred search space while acceptably bounding the number of candidate patches that the search space can generate (Section 3.5).

The ILP operates over a collection of subsets of patches drawn from a set of training patches. Each subset generalizes to a Genesis transform, with the final search space generated by the set of transforms that the solution to the ILP selects. Genesis uses a sampling algorithm to tractably derive the collection of subsets of patches for the ILP. This sampling algorithm incrementally builds up larger subsets of patches from smaller subsets, using a fitness function to identify promising candidate subsets (Section 3.4). Together, the sampling algorithm and final ILP formulation of the search space selection problem enable Genesis to scalably infer a set of transforms with both good coverage and good tractability.

1.2 Experimental Results

We use Genesis to infer patch search spaces and generate patches for three classes of errors of Java programs: null pointer errors (NPE), out of bounds errors (OOB), and class cast errors (CCE). Our training set includes 483 NPE patches, 199 OOB patches, and 287 CCE patches from 356 open source applications. Our benchmark set includes 20 NPE errors, 13 OOB errors, and 16 CCE errors from 41 open source applications. All of the benchmark applications are systematically collected from github [9] with up to 235K lines of code. Genesis generates correct patches for 21 out of the 49 errors (13 NPE errors, 6 OOB errors, and 5 CCE errors). Genesis significantly outperforms PAR [36], a previous patch generation system that works with manually defined mutation templates. For the same benchmark set, PAR generates correct patches only for 11 errors (7 NPE errors and 4 OOB errors). The reason is that the Genesis inference algorithm more successfully navigates the trade-off between coverage and complexity — it infers a richer but still effectively targeted patch search space (see Section 5.4).

1.3 Contributions

This paper makes the following contributions:

- **Transforms with Template ASTs and Generators:** We present novel transforms with template ASTs and generators for free template variables. These transforms enable Genesis to abstract away patch- and application-specific details to capture common patch patterns and strategies implemented by multiple patches drawn from different applications. Generators enable Genesis to synthesize the new code and logic required to obtain correct patches for errors that occur in large real-world applications.
- **Patch Generalization:** We present a novel patch generalization algorithm that, given a set of patches, automatically derives a transform that captures the common patch generation pattern present in the patches. This transform can generate all of the given patches as well as other patches with the same pattern in the same or other applications.
- **Search Space Inference:** We present a novel search space inference algorithm. Starting with a set of training patches, this algorithm infers a collection of transforms that together generate a search space of candidate patches with good coverage and tractability. The inference algorithm includes a novel sampling algorithm that identifies promising subsets of training patches to generalize and an ILP-based solution to the final search space selection problem.
- **Complete System and Experimental Results:** We present a complete patch generation system, including error localization and candidate patch evaluation algorithms, that uses the inferred search spaces to automatically patch errors in large real-world applications. We also present experimental results from this complete system.

Automatic patch generation systems have great potential for automatically eliminating errors in large software systems. By inferring transforms and search spaces from sets of previous successful patches, Genesis can automatically derive patch generation strategies that leverage the combined insight and expertise of developers worldwide. To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from previous successful patches.

2. Example

We next present a motivating example of using Genesis to generate a correct patch for a null-pointer exception (NPE) error from DataflowJavaSDK [5] revision c06125 (shown at the bottom of Figure 1).

Collect and Split Training Set: Genesis works with a training set of successful human patches to infer a search space for patch generation. In our example, the training set consists of 963 human patches collected from 356 github repositories. The training set contains patches for multiple kinds of errors. Specifically, 483 out of the 963 patches in the training set are for NPE errors and the remaining 480 patches are for out-of-bound (OOB) errors and class-cast exception (CCE) errors. To control overfitting, Genesis reserves 241 (25%) human patches from the training set as a validation set (Section 3.4). This leaves 722 human patches remaining in the training set.

Generalize Patches: The Genesis inference algorithm works with sampled subsets of patches from the training

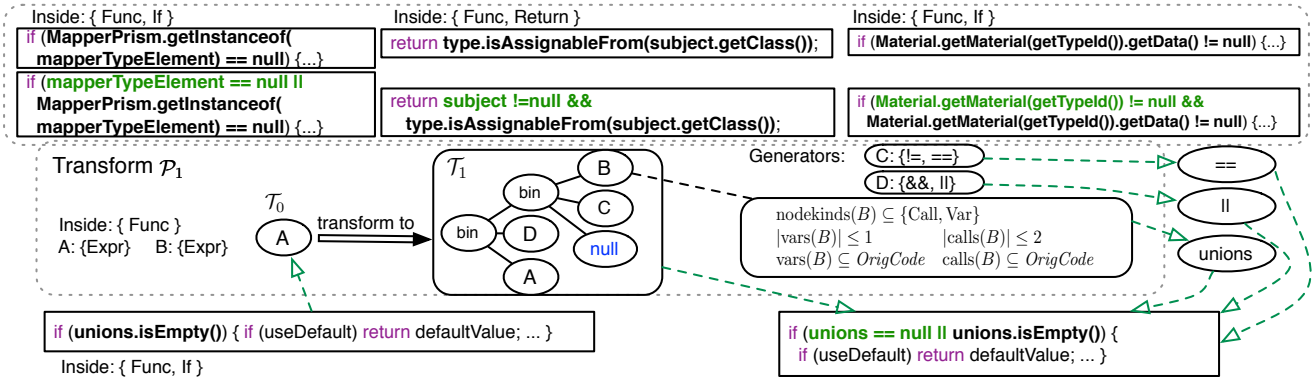


Figure 1. Example inference and application of a Genesis transform. The training patches (original and patched code) are at the top, the inferred transform is in the middle, and the new patch that Genesis generates is at the bottom.

set. For each subset, it applies a *generalization* algorithm to obtain a *transform* that it can apply to generate candidate patches. Figure 1 presents one of the sampled subsets of patches in our example: the first patch disjoins the clause `mapperTypeElement==null` to an if condition, the second patch conjoins the clause `subject!=null` to a return value, and the third patch conjoins the clause `Material.getMaterials(getTypeId())!=null` to an if condition. These patches are from three different applications, specifically mapstruct [20] revision 6d7a4d, modelmapper [25] revision d85131, and Bukkit [4] revision f13115. Genesis generalizes these three patches to obtain the transform $\mathcal{P}_1$ in Figure 1. When applied, $\mathcal{P}_1$ can generate all of the three patches in the selected subset as well as other patches for other applications.

Each transform has an initial template abstract syntax tree (AST) and a transformed template AST. These template ASTs capture the syntactic contexts of the original and patched code, respectively. In our example, the initial template AST $\mathcal{T}_0$ matches a boolean expression A that occurs within a function body (if all of the patches had modified if conditions, the initial pattern would have reflected that more specific context). The transformed template AST replaces the matched boolean expression A with a patch of the form $A \;\; op_1 (B \;\; op_2 \;\; \texttt{null})$, where $op_1 = C \in \{\texttt{!=}, \texttt{==}\}$ and $op_2 = D \in \{\texttt{\&\&}, \texttt{||}\}$, A is the original matched boolean expression, and B is an expression produced by a *generator*. In this example, Genesis infers the generator that generates all expressions that satisfy the constraints in Figure 1, specifically, that B contains method calls and variable accesses, that the number of variables in B is at most 2, that the number of calls in B is at most 1, and that any variables or calls in B must also appear in the original unpatched code.

In general, the generalization algorithm extracts 1) common AST structures shared by the pre-patch ASTs of the selected training patches (as the initial template AST $\mathcal{T}_0$), 2) common AST structures shared by the post-patch ASTs of the selected training patches (as the transformed AST $\mathcal{T}_1$), 3) common AST subtrees that are systematically copied/moved from the pre-patch ASTs to post-patch ASTs (e.g., the template variable A shared by $\mathcal{T}_0$ and $\mathcal{T}_1$), and 4) synthesis constraints of the new code snippets (e.g., the generator constraints for B).

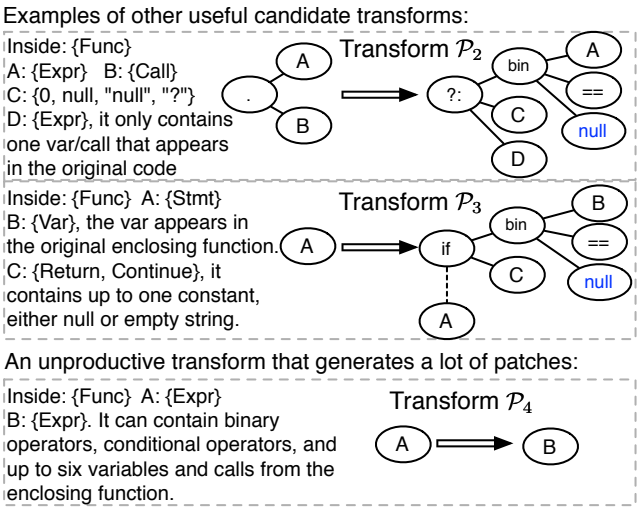
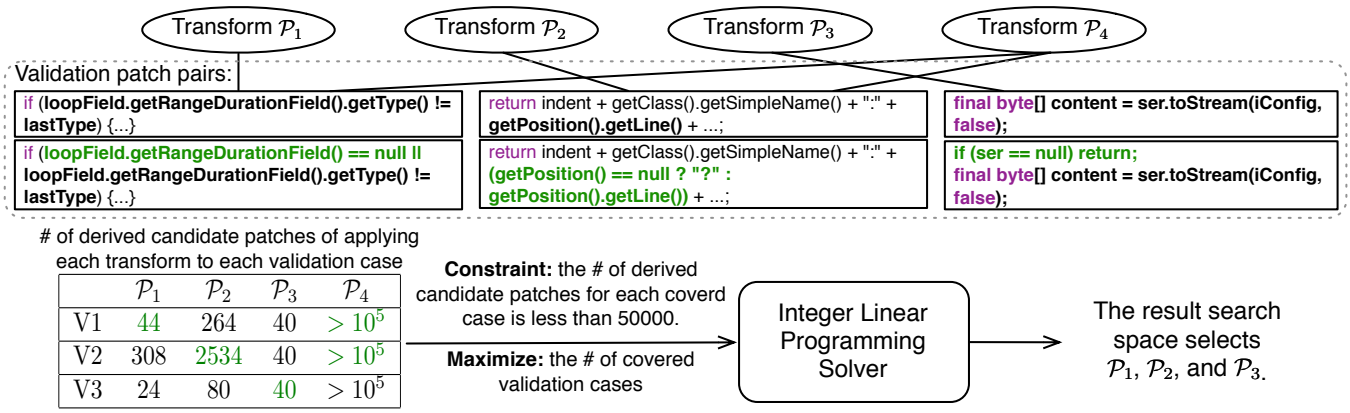


Figure 2. Examples of candidate transforms.

Obtain Candidate Transforms: A Genesis search space contains multiple transforms. To infer the search space, Genesis applies the previous generalization algorithm on many selected subsets to obtain a set of candidate transforms. It turns out that, in practice, essentially all useful transforms can be generated by generalizing relatively small subsets of training patches (six or fewer training patches). For small numbers of training patches, it might be feasible to simply generate all subsets of training patches smaller than a given bound. Because we work with too many training patches for this approach to scale, Genesis uses a sampling algorithm



	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$
V1	44	264	40	$> 10^5$
V2	308	2534	40	$> 10^5$
V3	24	80	40	$> 10^5$

Figure 3. Example of the search space inference. The validation patches (original and patched code) are in the middle.

that incrementally builds up promising subsets (starting with subsets of size two, then working up to subsets of size six). The algorithm uses a fitness function to prune unpromising subsets (Section 3.4).

In our example, $\mathcal{P}_1$ in Figure 1 is one of the obtained candidate transforms. Figure 2 presents examples of other obtained candidate transforms. The transform $\mathcal{P}_2$ in Figure 2 patches method call expressions on null objects. $\mathcal{P}_2$ adds a guard expression of the form A `==` `null` `?` C `:` D that first checks if A is `null` and, if so, evaluates to a constant such as `null` or 0 instead of throwing NPE. Note that $\mathcal{P}_2$ creates a new variable D instead of reusing the original code — the human patches in the training set often slightly refactor the code instead of directly using the original code as the else expression. The transform $\mathcal{P}_3$ in Figure 2 executes a return or continue statement C instead of an original statement A if B `==` `null`. $\mathcal{P}_3$ eliminates null pointer exceptions by returning from the enclosing function or skipping the current loop iteration if subsequent code would throw a null pointer exception.

Note that not all candidate transforms are equally useful. For example, the transform $\mathcal{P}_4$ in Figure 2 replaces an arbitrary expression A with another expression B, where B may contains binary operators, conditional operators, up to six variables from the enclosing function, and up to six method calls from the enclosing function. Due to the exponential number of possible expressions that B can take, $\mathcal{P}_4$ may generate a prohibitively large number of candidate patches when applied to many cases. A search space derived from $\mathcal{P}_4$ would be intractable for patch generation systems to explore. Genesis therefore prunes such candidate transforms with its final search space.

Infer Search Space: Genesis selects a subset of candidate transforms to form its search space. To obtain an effective search space, Genesis must navigate a tradeoff between coverage (how many correct patches it can generate) and tractability (how many candidate patches a patch generation system can effectively explore within a time budget [42]). Increasing the number of selected transforms tends to improve coverage but degrade tractability; decreasing the number of selected transforms tends to have the opposite effect.

Figure 3 presents the search space inference process for the candidate transforms $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_4$ from previous steps (see Figure 1 and Figure 2 for transform details). Genesis uses the patches in the validation set to evaluate the candidate transforms. Three such validation patches in our experiments are shown at the middle of Figure 3. These validation patches are from joda-time [17] revision bcb044, dynjs [6] revision 68df61, and orientdb [22] revision 51706f, respectively.

For each candidate transform and each validation patch, Genesis determines 1) whether the transform can generate the validation patch from the corresponding pre-patch code and 2) the total number of candidate patches that the transform would generate when applied to the pre-patch code. As shown in Figure 3, $\mathcal{P}_1$ can generate the first validation patch; $\mathcal{P}_2$ can generate the second patch; $\mathcal{P}_3$ can generate the third. $\mathcal{P}_4$ can generate both the first and the second validation patches but the numbers of generated candidate patches are large. The matrix at the left bottom of Figure 3 summarizes the evaluation results of the four candidate transforms on the three validation patches. Each number in the matrix denotes the number of candidate patches that a transform generates when applied to the pre-patch code of a validation patch. A green number indicates that a transform can generate a validation patch if applied to the pre-patch code of the patch.

Genesis formulates the tradeoff between the coverage and tractability as an integer linear programming (ILP) problem (Section 3.5). Specifically, given the information from the matrix, the ILP maximizes the number of validation patches that the selected transforms can generate, with the constraint that the total number of generated candidate patches from all

selected transforms for each covered validation case is less than 5×10^4. In our experiments, Genesis solves the ILP problem with an off-the-shelf solver and obtains a search space with 108 selected transforms including $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$.

Generate Patch for New Error: For the NPE error from DataflowJavaSDK [5] revision c06125 (shown at the bottom of Figure 1), Genesis first uses a fault localization technique (Section 4) to produce a ranked list of potential statements to modify. The resulting ranked list includes the if condition shown at the bottom left of Figure 1. Genesis then applies all transforms in the inferred search space including the transform $\mathcal{P}_1$ to the if condition to generate candidate patches.

Figure 1 shows how Genesis applies $\mathcal{P}_1$ to the if condition. Here the patch instantiates B as the variable `unions`, C as == and D as `||` to disjoin the clause `unions == null` to the original if condition. The patch causes the enclosing function `innerGetOnly()` to return a predefined default value when `unions` is `null` (instead of incorrectly throwing a null pointer exception).

Genesis also generates and explores 78301 other candidate patches for the error. Genesis uses the DataflowJavaSDK JUnit [19] test suite (which includes 830 test cases) to filter out these other candidate patches (as well as other candidate patches from other transforms and other patch candidate locations in DataflowJavaSDK). For Genesis to successfully patch the exception, the test suite must contain an input that exposes the exception, i.e., that causes the application to throw the null pointer exception. Genesis finds *validated patches*, i.e., patches that produce the correct output for all test cases, by running the patched application on all of the test cases (including the test case that exposed the exception) and checking the testing results.

Genesis produces a ranked list of validated patches following the space exploration order (see Section 4). In this example, the patch in Figure 1 is the only validated patch. This validated patch is also correct and matches the subsequent human developer patch for this exception. Note that PAR [36], a previous patch generation system based on manual transform templates, is unable to generate this correct patch. PAR does not contain a template to conjoin or disjoin a condition with an additional clause. In fact, PAR templates are unable to generate any correct patch for this error (see Section 5).

3. Inference System

Given a set of training pairs D, each of which corresponds to a program before a change and a program after a change, Genesis infers a set of transforms $\mathbb{P}$ which generates the search space.

Genesis obtains the search space in two steps: 1) it first runs a sampling algorithm to obtain a set of candidate transforms, each of which is generalized from a subset of changes in D and 2) it then selects a subset of the candidate transforms, formulating the trade-off between the coverage and the tractability of the search space as an integer linear programming (ILP) problem. It invokes an off-the-shelf ILP solver [34] to select the final set of transforms.

Sections 3.1 and 3.2 present definitions and notation. Section 3.3 presents definitions for the generalization function which derives candidate transforms from a set of program changes. Section 3.4 presents the sampling algorithm. Section 3.5 presents the search space inference algorithm. We discuss Genesis implementation details for handling Java programs in Section 3.6.

3.1 Preliminaries

The Genesis inference algorithm works with abstract syntax trees (ASTs) of programs. In this section, we model the programming language that Genesis works with as a context free grammar (CFGs) and we model ASTs as the parse trees for the CFG. Note that although the current implementation of Genesis is for Java, it is straightforward to extend the Genesis inference algorithm to other programming languages as well.

Definition 1 (CFG)**.** *A* context free grammar (CFG) G *is a tuple* $\langle N, \Sigma, R, s\rangle$ *where* N *is the set of non-terminals,* Σ *is the set of terminals,* R *is a set of production rules of the form* $a \rightarrow b_1 b_2 b_3 \ldots b_k$ *where* $a \in N$ *and* $b_i \in N \cup \Sigma$*, and* $s \in N$ *is the starting non-terminal of the grammar. The language of* G *is the set of strings derivable from the start non-terminal:* $\mathcal{L}(G) = \{w \in \Sigma^* \mid s \Rightarrow^* w\}$.

Definition 2 (AST)**.** *An* abstract syntax tree (AST) T *is a tuple* $\langle G, X, r, \xi, \sigma\rangle$ *where* $G = \langle N, \Sigma, R, s\rangle$ *is a CFG,* X *is a finite set of nodes in the tree,* $r \in X$ *is the root node of the tree,* $\xi : X \rightarrow X^*$ *maps each node to the list of its children nodes, and* $\sigma : X \rightarrow (N \cup \Sigma)$ *attaches a non-terminal or terminal label to each node in the tree.*

Definition 3 (AST Traversal and Valid AST)**.** *Given an AST* $T = \langle G, X, r, \xi, \sigma\rangle$ *where* $G = \langle N, \Sigma, R, s\rangle$, $\mathsf{str}(T) = \mathsf{traverse}(r) \in \Sigma^* \cup \{\bot\}$ *is the terminal string obtained via traversing* T *where*

$$\mathsf{traverse}(x) = \begin{cases} \mathsf{traverse}(x_{c_1}) \cdots \mathsf{traverse}(x_{c_k}) & \text{if } \sigma(x) \in N, \xi(x) = \langle x_{c_1} \cdots x_{c_k}\rangle, \text{ and} \\ & \quad \sigma(x) \rightarrow \sigma(x_{c_1}) \cdots \sigma(x_{c_k}) \in R \\ \sigma(x) & \text{if } \sigma(x) \in \Sigma \\ \bot & \text{otherwise.} \end{cases}$$

If the obtained string via traversal belongs to the language of G*, i.e.,* $\mathsf{str}(T) \in \mathcal{L}(G)$*, then the AST is* valid*.*

We next define AST forests and AST slices, which we will use in this section for describing our inference algorithm. An AST forest is similar to an AST except it contains multiple trees and a list of root nodes. An AST slice is a spe-

cial forest inside a large AST which corresponds to a list of adjacent siblings.

Definition 4 (AST Forest)**.** *An AST forest T is a tuple $\langle G, X, L, \xi, \sigma\rangle$ where G is a CFG, X is the set of nodes in the forest, $L = \langle x_1, x_2, \ldots, x_k\rangle$ is the list of root nodes of trees in the forest, ξ maps each node to the list of its children nodes, and σ maps each node in X to a non-terminal or terminal label.*

Definition 5 (AST Slice)**.** *An AST slice S is a pair $\langle T, L\rangle$. $T = \langle G, X, r, \xi, \sigma\rangle$ is an AST; $L = \langle r\rangle$ is a list that contains only the root node or $L = \langle x_{c_i}, \ldots, x_{c_j}\rangle$ is a list of AST sibling nodes in T such that $\exists x' \in X : \xi(x') = \langle x_{c_1}, \ldots, x_{c_i}, \ldots, x_{c_j}, \ldots, x_{c_k}\rangle$ (i.e., L is a sublist of $\xi(x')$).*

Given two ASTs T and T', where T is the AST before the change and T' is the AST after the change, Genesis computes AST difference between T and T' to produce an AST slice pair $\langle S, S'\rangle$ such that S and S' point to the sub-forests in T and T' that subsume the change. For brevity, in this section we assume $D = \{\langle S_1, S_1'\rangle, \langle S_2, S_2'\rangle, \ldots, \langle S_m, S_m'\rangle\}$ is a set of AST slice pairs, i.e., Genesis already converted AST pairs of changes to AST slices.

Notation and Utility Functions: We next introduce notation and utility functions that we are going to use in the rest of this section. For a map M, $\mathrm{dom}(M)$ denotes the domain of M. $M[a \mapsto b]$ denotes the new map which maps a to b and maps other elements in $\mathrm{dom}(M)$ to the same values as M. $\emptyset$ denotes an empty set or an empty map.

$\mathsf{nodes}(\xi, L)$ denotes the set of nodes in a forest, where ξ maps each node to a list of its children and L is the list of the root nodes of the trees in the forest.

$$\mathsf{nodes}(\xi, L) = \bigcup_{i=1}^{k} (\{x_i\} \cup \mathsf{nodes}(\xi, \xi(x_i)))$$
$$\text{where } L = \langle x_1, \ldots, x_k\rangle$$

$\mathsf{nonterm}(L, X, \xi, \sigma, N)$ denotes the set of non-terminals inside a forest, where L is the root nodes in the forest, X is a finite set of nodes, ξ maps each node to a list of children nodes, σ attaches each node to a terminal or non-terminal label, and N is the set of non-terminals:

$$\mathsf{nonterm}(L, X, \xi, \sigma, N) =$$
$$\bigcup_{i=1}^{k} (\{\sigma(x_i) \mid \sigma(x_i) \in N\} \cup \mathsf{nonterm}(\xi(x_i), X, \xi, \sigma, N))$$
$$\text{where } L = \langle x_1, \ldots, x_k\rangle$$

$\mathsf{inside}(S)$ denotes the set of non-terminals of the ancestor nodes of an AST slice S:

$$\mathsf{inside}(S) = \{\sigma(x') \mid \sigma(x') \in N\} \cup \mathsf{inside}(\langle T, \langle x'\rangle\rangle)$$
$$\text{where } S = \langle T, L\rangle, T = \langle G, X, r, \xi, \sigma\rangle, G = \langle N, \Sigma, R, s\rangle$$
$$L = \langle x_1, \ldots, x_k\rangle, \text{ and } \forall i \in [1, k] : x_i \in \xi(x')$$

$$\frac{\mathrm{diff}(A, B) = 0}{A \equiv B} \quad \begin{array}{ll} C = \langle G, X, \xi, \sigma\rangle & L = \langle x_1, x_2, \ldots x_k\rangle \\ C' = \langle G, X', \xi', \sigma'\rangle & L' = \langle x_1', x_2', \ldots, x_{k'}'\rangle \\ G = \langle N, \Sigma, R, s\rangle & \end{array}$$

$$\mathrm{diff}(\langle G, X, r, \xi, \sigma\rangle, \langle G, X', r', \xi', \sigma'\rangle) = \mathrm{d}(\langle \mathrm{C}, \langle \mathrm{r}\rangle\rangle, \langle \mathrm{C}', \langle \mathrm{r}'\rangle\rangle)$$
$$\mathrm{diff}(\langle\langle G, X, r, \xi, \sigma\rangle, L\rangle, \langle\langle G, X', r', \xi', \sigma'\rangle, L'\rangle) =$$
$$\mathrm{diff}(\langle G, X, L, \xi, \sigma\rangle, \langle G, X', L', \xi', \sigma'\rangle) =$$
$$\mathrm{d}(\langle C, L\rangle, \langle C', L'\rangle) =$$
$$\begin{cases} \sum_{i=1}^{k} \mathrm{d}(\langle C, \langle x_i\rangle\rangle, \langle C', \langle x_i'\rangle\rangle) & k = k' > 1 \\ \mathrm{d}(\langle C, \xi(x_1)\rangle, \langle C', \xi'(x_1')\rangle) & k = k' = 1, \sigma(x_1) = \sigma(x_1') \in N \\ 0 & k = k' = 1, \sigma(x_1) = \sigma(x_1') \in \Sigma \\ 1 & k = k' = 1, \sigma(x_1) \neq \sigma(x_1') \in \Sigma \\ 0 & k = k' = 0 \\ \infty & \text{otherwise} \end{cases}$$

Figure 4. Definition of $\mathrm{diff}()$ and "$\equiv$"

$\mathrm{diff}(A, B)$ denotes the number of different terminals in leaf nodes between two ASTs, AST slices, or AST forests. If A and B differs in not just terminals in leaf nodes, $\mathrm{diff}(A, B) = \infty$. $A \equiv B$ denotes that A and B are equivalent, i.e., $\mathrm{diff}(A, B) = 0$. Figure 4 presents the detailed definitions of $\mathrm{diff}()$ and "$\equiv$".

3.2 Template AST Forest, Generator, and Transforms

Template AST Forest: We next introduce the template AST forest, which can represent a set of concrete AST forest or AST slice. The key difference between template AST forest and concrete AST forest is that template AST forest contains template variables, each of which can match against any appropriate AST subtrees or AST sub-forests.

Definition 6 (Template AST Forest)**.** *A template AST forest $\mathcal{T}$ is a tuple $\langle G, V, \gamma, X, L, \xi, \sigma\rangle$, where $G = \langle N, \Sigma, R, s\rangle$ is a CFG, V is a finite set of template variables, $\gamma : V \to \{0, 1\} \times \mathrm{Powerset}(N)$ is a map that assigns each template variable to a bit of zero or one and a set of non-terminals, X is a finite set of nodes in the subtree, $L = \langle x_1, x_2, \ldots, x_k\rangle$, $x_i \in X$ is the list of root nodes of the trees in the forest, $\xi : X \to X^*$ maps each node to the list of its children nodes, and $\sigma : X \to N \cup \Sigma \cup V$ attaches a non-terminal, a terminal, or a template variable to each node in the tree.*

For each template variable $v \in V$, $\gamma(v) = \langle b, W\rangle$ determines the kind of AST subtrees or sub-forests which the variable can match against. If $b = 0$, v can match against only AST subtrees not sub-forests. If $b = 1$, then v can match against both subtrees and sub-forests. Additionally, v can match against an AST subtree or sub-forest only if its root nodes do not correspond to any non-terminal outside W.

Intuitively, each non-terminal in the CFG of a programming language typically corresponds to one kind of syntactic unit in programs at certain granularity. Template AST forests

with template variables enable Genesis to achieve desirable abstraction over concrete AST trees during the inference. They also enable Genesis to abstract away program-specific syntactic details so that Genesis can infer useful transforms from changes across different programs and different applications.

Definition 7 ("$\models$" and "$\models_{\texttt{slice}}$" Operators for Template AST Forest)**.** *Figure 5 presents the formal definition of the operator "$\models$" for a template AST forest $\mathcal{T} = \langle G, V, \gamma, X, L, \xi, \sigma\rangle$. "$\mathcal{T} \models \langle T, M\rangle$" denotes that $\mathcal{T}$ matches the concrete AST forest T with the template variable bindings specified in M, where M is a map that assigns each template variable in V to an AST forest.*

Figure 5 also presents the formal definition of the operator "$\models_{\texttt{slice}}$". Similarly, "$\mathcal{T} \models_{\texttt{slice}} \langle S, M\rangle$" denotes that $\mathcal{T}$ matches the concrete AST slice S with the variable bindings specified in M.

The first rule in Figure 5 corresponds to the simple case of a single terminal node. The second and the third rules correspond to the cases of a single non-terminal node or a list of nodes, respectively. The two rules recursively match the children nodes and each individual node in the list.

The fourth and the fifth rules correspond to the case of a single template variable node in the template AST forest. The fourth rule matches the template variable against a forest, while the fifth rule matches the template variable against a tree. These two rules check that the corresponding forest or tree of the variable in the binding map M is equivalent to the forest or tree that the rules are matching against.

Generators: Many productive patches do not just rearrange existing components and/or logics in the changed slice, but also introduce useful new components and/or logic. We next introduce generators, which enable Genesis to synthesize such patches.

Definition 8 (Generator)**.** *A generator $\mathcal{G}$ is a tuple $\langle G, b, \delta, W\rangle$, where $G = \langle N, \Sigma, R, s\rangle$ is a CFG, $b \in \{0, 1\}$ indicates the behavior of the generator, δ is an integer bound for the number of tree nodes, $W \subseteq N$ is the set of allowed non-terminals during generation.*

Currently, generators in Genesis exhibit two kinds of behaviors. If $b = 0$, the generator generates a sub-forest with less than δ nodes that contains only non-terminals inside the set W. If $b = 1$, such a generator copies an existing sub-forest from the original AST tree with non-terminal labels in W and then replaces up to δ leaf non-terminal nodes in the copied sub-forest.

Definition 9 (Generation Operator "$\Longrightarrow$" for Generators)**.** *Figure 6 presents the formal definition of the operator "$\Longrightarrow$" for a generator $\mathcal{G}$. Given $\mathcal{G}$ and an AST slice $S = \langle T, L\rangle$ as the context, $\langle \mathcal{G}, S\rangle \Longrightarrow T'$ denotes that the generator $\mathcal{G}$ generates the AST forest T'.*

$$\boxed{\begin{array}{ll} G = \langle N, \Sigma, R, s\rangle & \\ \mathcal{T} = \langle G, V, \gamma, X, L, \xi, \sigma\rangle & L = \langle x_1, x_2, \ldots, x_k\rangle \\ T = \langle G, X', L', \xi', \sigma'\rangle & L' = \langle x'_1, x'_2, \ldots, x'_{k'}\rangle \end{array}}$$

$$\frac{k = k' = 1 \qquad \sigma(x_1) = \sigma'(x'_1) \in \Sigma}{\mathcal{T} \models \langle T, M\rangle}$$

$$\frac{\begin{array}{c} k = k' = 1 \qquad \sigma(x_1) = \sigma'(x'_1) \in N \\ \langle G, V, \gamma, X, \xi(x_1), \xi, \sigma\rangle \models \langle \langle G, X', \xi'(x'_1), \xi', \sigma'\rangle, M\rangle \end{array}}{\mathcal{T} \models \langle T, M\rangle}$$

$$\frac{\begin{array}{c} k = k' > 1 \\ \forall i \in \{1, 2, \ldots, k\}\ \left(\langle G, V, \gamma, X, \{x_i\}, \xi, \sigma\rangle \models \langle \langle G, X', \{x'_i\}, \xi', \sigma'\rangle, M\rangle\right) \end{array}}{\mathcal{T} \models \langle T, M\rangle}$$

$$\frac{\begin{array}{c} k = 1 \qquad \sigma(x_1) = v \in V \\ M(v) \equiv T \qquad \gamma(v) = \langle 1, W\rangle \qquad (\cup_{i=1}^{k'} \sigma'(x'_i)) \subseteq (W \cup \Sigma) \end{array}}{\mathcal{T} \models \langle T, M\rangle}$$

$$\frac{\begin{array}{c} k = k' = 1 \qquad \sigma(x_1) = v \in V \\ M(v) \equiv T \qquad \gamma(v) = \langle 0, W\rangle \qquad \sigma'(x'_1) \in (W \cup \Sigma) \end{array}}{\mathcal{T} \models \langle T, M\rangle}$$

$$\frac{\mathcal{T} \models \langle \langle G, X', L', \xi', \sigma'\rangle, M\rangle}{\mathcal{T} \models_{slice} \langle \langle \langle G, X', r', \xi', \sigma'\rangle, L'\rangle, M\rangle}$$

Figure 5. Definition of the operators "$\models$" and "$\models_{\texttt{slice}}$" for the template AST forest $\mathcal{T}$

$$\boxed{\begin{array}{ll} G = \langle N, \Sigma, R, s\rangle & S = \langle T, L\rangle \\ T = \langle G, X, r, \xi, \sigma\rangle & T' = \langle G, X', L', \xi', \sigma'\rangle \end{array}}$$

$$\frac{|\mathsf{nodes}(\xi', L')| \leq \delta \qquad \mathsf{nonterm}(L', X', \xi', \sigma', N) \subseteq W}{\langle \langle G, 0, \delta, W\rangle, S\rangle \Longrightarrow T'}$$

$$\frac{\begin{array}{c} \exists x' \in X\ \left(L'' \text{ is a sublist of } \xi(x')\right) \\ \mathrm{diff}(\langle G, X, L'', \xi, \sigma\rangle, T') \leq \delta \qquad \forall x'' \in L'\ \left(\sigma'(x'') \in W\right) \end{array}}{\langle \langle G, 1, \delta, W\rangle, S\rangle \Longrightarrow T'}$$

Figure 6. Definition of the operator "$\Longrightarrow$" for the Generator $\mathcal{G} = \langle G, b, \delta, W\rangle$

The first rule in Figure 6 handles the case where $b = 0$. The rule checks that the number of nodes in the result forest is within the bound δ and the set of non-terminals in the forest is a subset of W. The second rule handles the case where $b = 1$. The rule checks that the difference result forest and an existing forest in the original AST is within the bound and the root labels are in W.

Note that, theoretically, generators may generate an infinite number of different AST forests for programming languages like Java, because the set of terminals (e.g., identifiers and constants) is infinite. Genesis, in practice, places

$$\frac{\begin{array}{c} S = \langle\langle G, X, r, \xi, \sigma\rangle, L\rangle \qquad A \subseteq \mathsf{inside}(S) \\ \mathcal{T}_0 \models \langle S, M\rangle \qquad B = \{v_1 \mapsto \mathcal{G}_1, v_2 \mapsto \mathcal{G}_2, \ldots, v_m \mapsto \mathcal{G}_m\} \\ \forall_{i=1}^{m} \left(\langle \mathcal{G}_i, S\rangle \Longrightarrow T_i''\right) \\ M' = \{v_1 \mapsto T_1'', v_2 \mapsto T_2'', \ldots v_k \mapsto T_m''\} \\ \mathcal{T}_1 \models \langle T', M \cup M'\rangle \qquad \langle S, T'\rangle \rhd T \qquad \mathsf{str}(T) \in \mathcal{L}(G) \end{array}}{\langle\langle A, \mathcal{T}_0, \mathcal{T}_1, B\rangle, S\rangle \Longrightarrow T}$$

$$\frac{\begin{array}{c} 1 \leq i \leq j \leq k \\ S = \langle\langle G, X, r, \xi, \sigma\rangle, L\rangle \quad L = \langle x_i, \ldots, x_j\rangle \quad \xi(x') = \langle x_1, x_2, \ldots, x_k\rangle \\ T' = \langle G, X', L', \xi', \sigma'\rangle \quad L' = \langle x_1'', x_2'', \ldots, x_{k'}''\rangle \qquad X \cap X' = \emptyset \\ L'' = \langle x_1, \ldots, x_{i-1}, x_1'', x_2'', \ldots, x_{k'}'', x_{j+1}, \ldots, x_k\rangle \end{array}}{\langle S, T'\rangle \rhd \langle G, X \cup X', r, (\xi \cup \xi')[x' \mapsto L''], \sigma \cup \sigma'\rangle}$$

$$\frac{S' = \langle T', L'\rangle \qquad \langle \mathcal{P}, S\rangle \Longrightarrow T'}{\langle \mathcal{P}, S\rangle \Longrightarrow_{\texttt{slice}} S'}$$

Figure 7. Definition of the operators "$\Longrightarrow$" and "$\Longrightarrow_{\texttt{slice}}$" for the transform $\mathcal{P}$

additional Java-specific constraints on generators to make the generated set finite and tractable (See Section 3.6).

Transforms: Finally, we introduce transforms, which generate the search space inferred by Genesis. Given an AST slice, a transform generates new AST trees.

Definition 10 (Transform)**.** *A transform $\mathcal{P}$ is a tuple $\langle A, \mathcal{T}_0, \mathcal{T}_1, B\rangle$. A : Powerset(N) is a set of non-terminals to denote the context where this transform can apply; $\mathcal{T}_0 = \langle G, V_0, \gamma_0, X_0, L_0, \xi_0, \sigma_0\rangle$ is the template AST forest before applying the transform; $\mathcal{T}_1 = \langle G, V_1, \gamma_1, X_1, L_1, \xi_1, \sigma_1\rangle$ is the template AST forest after applying the transform; B maps each template variable v that only appears in $\mathcal{T}_1$ to a generator (i.e., $\forall v \in V_1 \setminus V_0$, $B(v)$ is a generator).*

Definition 11 ("$\Longrightarrow$" and "$\Longrightarrow_{\texttt{slice}}$" Operators for Transforms)**.** *Figure 7 presents the formal definition of the "$\Longrightarrow$" and "$\Longrightarrow_{slice}$" operator for a transform $\mathcal{P}$. "$\langle \mathcal{P}, S\rangle \Longrightarrow T'$" denotes that applying $\mathcal{P}$ to the AST slice S generates the new AST T'. "$\langle \mathcal{P}, S\rangle \Longrightarrow_{slice} S'$" denotes that applying $\mathcal{P}$ to the AST slice S generates the AST of the slice S'.*

Intuitively, in Figure 7 A and $\mathcal{T}_0$ determine the context where the transform $\mathcal{P}$ can apply. $\mathcal{P}$ can apply to an AST slice S only if the ancestors of S have all non-terminal labels in A and $\mathcal{T}_0$ can match against S with a variable binding map M. $\mathcal{T}_1$ and B then determine the transformed AST tree. $\mathcal{T}_1$ specifies the new arrangement of various components and B specifies the generators to generate AST sub-forests to replace free template variables in $\mathcal{T}_1$. Note that $\langle S, T'\rangle \rhd T$ denotes that the obtained AST tree of replacing the AST slice S with the AST forest T' is equivalent to T.

3.3 Transform Generalization

The generalization operation for transforms takes a set of AST slice pairs D as the input and produces a set of transforms, each of which can at least generate the corresponding

$$\begin{array}{l} G = (N, \Sigma, R, s) \qquad D = \langle\langle S_1, S_1'\rangle, \langle S_2, S_2'\rangle, \ldots, \langle S_m, S_m'\rangle\rangle \\ \forall i \in \{1, 2, \ldots, m\}: \\ S_i = \langle T_i, L_i\rangle \qquad T_i = \langle G, X_i, r_i, \xi_i, \sigma_i\rangle \\ S_i' = \langle T_i', L_i'\rangle \qquad T_i' = \langle G, X_i', r_i', \xi_i', \sigma_i'\rangle \\ L_i' = \langle x_{i,1}', x_{i,2}', \ldots, x_{i,k_i'}'\rangle \end{array}$$

$$\psi(D) = \begin{cases} \{\mathbb{A}, \mathbb{B}\} & \forall i \in \{1, \ldots, m\}, \forall j \in \{1, \ldots, k_i'\}, \sigma'(x_{i,j}') \in N \\ \{\mathbb{A}\} & \text{otherwise} \end{cases}$$

where:

$\mathbb{A} = \langle G, 0, \max_{i=1}^{m} |\mathsf{nodes}(\xi_i', L_i')|, \bigcup_{i=1}^{m} \mathsf{nonterm}(S_i')\rangle$

$\mathbb{B} = \langle G, 1, \max_{i=1}^{m} \mathbb{C}_i, \bigcup_{i=1}^{m} \bigcup_{j=1}^{k_i'} \{\sigma'(x_{i,j}')\}\rangle$

$\mathbb{C}_i = \min_{L_i''} \mathrm{diff}(\langle T_i, L_i''\rangle, S_i'), \exists x'' \in X_i,\ L_i''$ is a sublist of $\xi_i(x'')$

Figure 8. Definition of the generator inference operator ψ

$$\begin{array}{l} \Psi(\langle\langle S_1, S_1'\rangle, \langle S_2, S_2'\rangle, \ldots, \langle S_m, S_m'\rangle\rangle) = \\ \{\langle \cap_{i=1}^{m} \mathsf{inside}(S_i), \mathcal{T}_0, \mathcal{T}_1, B\rangle \mid \\ \quad \langle \mathcal{T}_0, M\rangle = \Psi'(\langle S_1, S_2, \ldots, S_m\rangle, \emptyset), \\ \quad \langle \mathcal{T}_1, M'\rangle = \Psi'(\langle S_1', S_2', \ldots, S_m'\rangle, M), \\ \quad B = \{v_i \mapsto \mathcal{G}_i \mid \\ \qquad v_i \in \mathrm{dom}(M') \setminus \mathrm{dom(M)}, \\ \qquad M'(v_i) = \langle b_i, W_i, \langle S_{i,1}'', S_{i,2}'', \ldots, S_{i,m}''\rangle\rangle, \\ \qquad P_i = \{\langle S_1, S_{i,1}''\rangle, \langle S_2, S_{i,2}''\rangle, \ldots, \langle S_m, S_{i,m}''\rangle\}, \\ \qquad \mathcal{G}_i \in \psi(P_i)\}\} \end{array}$$

Figure 9. Definition of the generalization function Ψ

changes of the pairs in D. We first present the generalization operator for generators then we present the generalization operator for transforms.

Definition 12 (Generator Generalization)**.** *Figure 8 presents the definition of generalization function $\psi(D)$. Given a set of of AST slice pairs $D = \{\langle S_1, S_1'\rangle, \langle S_2, S_2'\rangle, \ldots, \langle S_m, S_m'\rangle\}$ from the same CFG grammar G, where S_i is the generation context AST slice and S_i' is the generated result AST slice, $\psi(D) = \{\mathcal{G}_1, \mathcal{G}_2, \ldots \mathcal{G}_k\}$ denotes the set of the generators generalized from D.*

In Figure 8, $\mathbb{A}$ is the formula for a generator that generates from scratch (i.e., $b = 0$) and $\mathbb{B}$ is the formula for a generator that generates via copying from the existing AST tree (i.e., $b = 1$). The formula $\mathbb{A}$ produces the generator by computing the bound of the number of nodes and the set of non-terminals in the supplied slices. The formula $\mathbb{B}$ produces the generator by computing 1) the bound of the minimum diff distance between each supplied slice and an arbitrary existing forest in the AST tree and 2) the set of non-terminals of the root node labels of the supplied slices.

Definition 13 (Transform Generalization)**.** *Figure 9 presents the definition of $\Psi(D)$. Given a set of pairs of AST slices $D = \{\langle S_1, S_1'\rangle, \langle S_2, S_2'\rangle, \ldots, \langle S_m, S_m'\rangle\}$ where S_i is the*

$\mathbb{S} = \langle S_1, S_2, \ldots, S_m \rangle$ $G = (N, \Sigma, R, s)$ x' is a fresh node v' is a fresh template variable
$\forall i \in \{1, 2, \ldots, m\}:$ $S_i = \langle T_i, L_i \rangle$ $L_i = \langle x_{i,1}, x_{i,2}, \ldots, x_{i,k_i} \rangle$ $T_i = \langle G, X_i, r_i, \xi_i, \sigma_i \rangle$ $c_i = \sigma_i(x_{i,1})$

$\Psi'(\mathbb{S}, M) =$	**Conditions for k and c**	**Other Conditions**
$\langle\langle G, \emptyset, \emptyset, \langle\rangle, \emptyset, \emptyset\rangle, M\rangle$	$\forall i \in \{1, \ldots, m\}\, k_i = 0$	
$\langle\langle G, \emptyset, \emptyset,$ $\{x'\}, \langle x'\rangle,$ $\{x' \mapsto \emptyset\}, \{x' \mapsto d\}\rangle, M\rangle$	$\forall i \in \{1, \ldots, m\}$ $k_i = 1$ $c_i = d$ $d \in \Sigma$	
$\langle\langle G, V, \gamma,$ $X' \cup \{x'\}, \langle x'\rangle,$ $\xi'[x' \mapsto L'], \sigma'[x' \mapsto d]\rangle, M'\rangle$	$\forall i \in \{1, \ldots, m\}$ $k_i = 1$ $c_i = d$ $d \in N$	$\mathbb{S}' = \langle\langle T_1, \xi_1(x_{1,1})\rangle, \langle T_2, \xi_2(x_{2,1})\rangle, \ldots, \langle T_m, \xi_m(x_{m,1})\rangle\rangle$ $\Psi'(\mathbb{S}', M) = \langle \mathcal{T}, M'\rangle$ $\mathcal{T} = \langle G, V, \gamma, X', L', \xi', \sigma'\rangle$
$\langle\langle G, \{v\}, \{v \mapsto \langle 0, W\rangle\},$ $\{x'\}, \langle x'\rangle,$ $\{x' \mapsto \emptyset\}, \{x' \mapsto v\}\rangle, M\rangle$	$\exists i, i' \in \{1, \ldots, m\}$ $c_i \neq c_{i'}$	$M(v) = \langle 0, W, \langle S'_1, S'_2, \ldots, S'_m\rangle\rangle$ $\forall i \in \{1, \ldots, m\}\, (S_i \equiv S'_i)$
$\langle\langle G, \{v'\}, \{v' \mapsto \langle 0, W\rangle\},$ $\{x'\}, \langle x'\rangle,$ $\{x' \mapsto \emptyset\}, \{x' \mapsto v'\}\rangle, M'\rangle$	$\forall i \in \{1, \ldots, m\}$ $k_i = 1$ $\exists i', i'' \in \{1, \ldots, m\}$ $(c_{i'} \neq c_{i''})$	$\forall v \in \mathrm{dom}(M)$ $M(v) = \langle 0, W', \langle S'_1, S'_2, \ldots, S'_m\rangle\rangle \quad \exists i \in \{1, 2, \ldots, m\}\, (S_i \not\equiv S'_i)$ $W = N \cap (\cup_{i=1}^{m} \{\sigma_i(x_{i,1})\})$ $M' = M[v' \mapsto \langle 0, W, \mathbb{S}\rangle]$
$\langle\langle G, \cup_{j=1}^{k} V_j, \cup_{j=1}^{k} \gamma_j,$ $\cup_{j=1}^{k} X_j, \langle r_1, r_2, \ldots, r_k\rangle,$ $\cup_{j=1}^{k} \xi_j, \cup_{j=1}^{k} \sigma_j\rangle, M'_k\rangle$	$\forall i \in \{1, \ldots, m\}$ $k_i = k'$ $k' > 1$	$M'_0 = M$ $\forall j \in \{1, \ldots, k\}$ $\mathbb{S}'_j = \langle\langle T_1, \langle x_{1,j}\rangle\rangle, \langle T_2, \langle x_{2,j}\rangle\rangle, \ldots, \langle T_m, \langle x_{m,j}\rangle\rangle\rangle$ $\Psi'(\mathbb{S}'_j, M'_{j-1}) = \langle\langle G, V_j, \gamma_j, X_j, \langle r_j\rangle, \xi_j, \sigma_j\rangle, M'_j\rangle$
$\langle\langle G, \{v\}, \{v \mapsto \langle 1, W\rangle\},$ $\{x'\}, \langle x'\rangle,$ $\{x' \mapsto \emptyset\}, \{x' \mapsto v\}\rangle, M\rangle$	$\exists i', i'' \in \{1, \ldots, m\}$ $k_{i'} \neq k_{i''}$	$M(v) = \langle 1, W, \langle S'_1, S'_2, \ldots, S'_m\rangle\rangle$ $\forall i \in \{1, 2, \ldots, m\}\, (S_i \equiv S'_i)$
$\langle\langle G, \{v'\}, \{v' \mapsto \langle 1, W\rangle\},$ $\{x'\}, \langle x'\rangle,$ $\{x' \mapsto \emptyset\}, \{x' \mapsto v'\}\rangle, M'\rangle$	$\exists i', i'' \in \{1, \ldots, m\}$ $k_{i'} \neq k_{i''}$	$\forall v \in \mathrm{dom}(M)$ $M(v) = \langle 1, W', \langle S'_1, S'_2, \ldots, S'_m\rangle\rangle \quad \exists i \in \{1, 2, \ldots, m\}\, (S_i \not\equiv S'_i)$ $W = N \cap (\cup_{i=1}^{m} \cup_{j=1}^{k_i} \{\sigma_i(x_{i,j})\})$ $M' = M[v' \mapsto \langle 1, W, \mathbb{S}\rangle]$

Figure 10. Definition of Ψ'

AST slice before a change and S'_i is the AST slice after a change, $\Psi(D)$ denotes the set of transforms generalized from D.

The formula for Ψ in Figure 9 invokes Ψ' twice to compute the template AST forest before the change $\mathcal{T}_0$ and the template AST forest after the change $\mathcal{T}_1$. It then computes B by invoking ψ to obtain the generalized generators for AST sub-slices that match against each free template variable in $\mathcal{T}_1$.

Note that Figure 10 presents the definition of Ψ'. Intuitively, Ψ' is the generalization function for template AST forests. The function $\Psi'(\mathbb{S}, M) = \langle \mathcal{T}, M'\rangle$ takes a list of AST slices $\mathbb{S}$ and an initial variable binding map M and produces a generalized template AST forest $\mathcal{T}$ and an updated variable binding map M'.

The first two rows in Figure 10 correspond to the formulas for the cases of empty slices and slices with a single terminal, respectively. The two formulas simply create an empty template AST forest or a template AST forest with a single non-terminal node. The third row corresponds to the formula for the case of a single non-terminal. The formula recursively invokes Ψ' on the list of children nodes of each slice and creates a new node with the non-terminal label in the result template AST forest as the root node.

The fourth and fifth rows correspond to the formulas for the cases where each slice is a single tree and the root nodes of the slice trees do not match. The fourth formula handles the case where there is an existing template variable in M that can match the slice trees. The formula creates a template AST forest with the matching variable. The fifth formula handles the case where there is no existing template variable in M that can match the slice trees. The formula creates a template AST forest with a new template variable and updates the variable binding map to include this new variable accordingly.

The sixth row corresponds to the formula for the case where each slice is a forest with the same number of trees. The formula recursively invokes Ψ' on each individual tree and combines the obtained template AST forests. The seventh row corresponds to the formula for the case in which each slice is a forest, the forests do not match, and there is an existing template variable in M to match these forests. The formula therefore creates a template AST forest with the matching variable. The eighth row corresponds to the formula for the case where the forests do not match and there is no template variable in M to match these forests. The formula creates a template AST forest with a new template variable and updates the variable binding map accordingly.

Input : a training set of pairs of AST slices D and a validation set of pairs of AST slices E
Output: a set of transforms $\mathbb{P}'$

1 $\mathbb{W} \leftarrow \{\{\langle S, S'\rangle, \langle S'', S'''\rangle\} \mid \langle S, S'\rangle \in D, \langle S'', S'''\rangle \in D, \langle S, S'\rangle \neq \langle S'', S'''\rangle\}$
2 **for** $i = 1$ **to** 5 **do**
3 $\quad f \leftarrow \{\mathbb{S} \mapsto \text{fitness}(\mathbb{W}, \mathbb{S}, D, E) \mid \mathbb{S} \in \mathbb{W}\}$
4 $\quad \mathbb{W}' \leftarrow \{\mathbb{S} \mid \mathbb{S} \in \mathbb{W}, f(\mathbb{S}) > 0\}$
5 $\quad$ Sort elements in $\mathbb{W}'$ based on f
6 $\quad$ Select top α elements in $\mathbb{W}'$ with largest f value as a new set $\mathbb{W}''$
7 $\quad \mathbb{W} \leftarrow \mathbb{W}''$
8 $\quad$ **if** $i \neq 5$ **then**
9 $\quad\quad$ **for** $\mathbb{S}$ **in** $\mathbb{W}''$ **do**
10 $\quad\quad\quad$ **for** $\langle S, S'\rangle$ **in** D **do**
11 $\quad\quad\quad\quad \mathbb{W} \leftarrow \mathbb{W} \cup \{\mathbb{S} \cup \langle S, S'\rangle\}$
12 $\mathbb{P}' \leftarrow \cup_{\mathbb{S} \in \mathbb{W}} \Psi(\mathbb{S})$
13 **return** $\mathbb{P}'$

Figure 11. Sampling algorithm $\text{sample}(D, E)$

Theorem 14 (Soundness of Generalization)**.** *For any set of AST slice pairs D, $\forall \mathcal{P} \in \Psi(D)$, $\forall \langle S, S'\rangle \in D$, $\langle \mathcal{P}, S\rangle \Longrightarrow_{\texttt{slice}} S'$.*

The generalization function Ψ is sound so that the transform $\mathcal{P}$ is able to generate the corresponding change for each pair in D, from which it is generalized. Intuitively, assume a transform space that denotes all possible program changes and the program changes in the training database D are points in the transform space. Then the generalization function Ψ produces a set of potentially useful transforms, each of which covers all of the points for D in the space.

3.4 Sampling Algorithm

Given a training database D, we could obtain an exponential number of transforms with the generalization function Ψ described in Section 3.3, i.e., we can invoke Ψ on any subset of D to obtain a different set of transforms.

Not all of the generalized transforms are useful. The goal of the sampling algorithm is to use the described generalization function to systematically obtain a set of productive candidate transforms for the inference algorithm to consider.

Figure 11 presents the pseudo-code of our sampling algorithm. As a standard approach in other learning and inference algorithms to avoid overfitting, Genesis splits the training database into a training set D and a validation set E. Genesis invokes the generalization functions only on pairs in the training set D to obtain candidate transforms. Genesis uses the validation set E to evaluate generalized transforms only.

$\mathbb{W}$ in Figure 11 is a work set that contains the candidate subset of D that the sampling algorithm is considering to use to obtain generalized transforms. The algorithm runs five iterations. At each iteration, the algorithm first computes a fitness score for each candidate subset, keep the top α candidate subsets (we empirically set α to 500 and 1000 in our experiments, see Section 5.1), and eliminate the rest from $\mathbb{W}$ (see lines 3-7). The algorithm then attempts to update $\mathbb{W}$ by augmenting each subset in $\mathbb{W}$ with one additional pair in D (see lines 8-10).

Input : a power set of pairs of AST slices $\mathbb{W}$, a set $\mathbb{S} \in \mathbb{W}$, a training set of AST slice pairs D, and a validation set of AST slice pairs E
Output: the fitness score for $\mathbb{S}$

1 Initialize C to map each pair in $D \cup E$ to 0
2 **for** $\mathbb{S}'$ **in** $\mathbb{W}$ **do**
3 $\quad A \leftarrow \emptyset$
4 $\quad$ **for** $\mathcal{P}$ **in** $\Psi(\mathbb{S}')$ **do**
5 $\quad\quad B \leftarrow \{\langle S, S'\rangle \mid \langle S, S'\rangle \in (D \cup E), \langle \mathcal{P}, S\rangle \Longrightarrow_{\texttt{slice}} S', |\{\texttt{str}(T) \mid \langle \mathcal{P}, S\rangle \Longrightarrow T\}| < \beta\}$
6 $\quad\quad A \leftarrow A \cup B$
7 $\quad$ **for** $\langle S, S'\rangle$ **in** A **do**
8 $\quad\quad C \leftarrow C[\langle S, S'\rangle \mapsto C(\langle S, S'\rangle) + 1]$
9 $f \leftarrow 0$
10 **for** $\mathcal{P}$ **in** $\Psi(\mathbb{S})$ **do**
11 $\quad B \leftarrow \{\langle S, S'\rangle \mid \langle S, S'\rangle \in (D \cup E), \langle \mathcal{P}, S\rangle \Longrightarrow_{\texttt{slice}} S'\}$
12 $\quad f' \leftarrow 0$
13 $\quad$ **for** $\langle S, S'\rangle$ **in** B **do**
14 $\quad\quad c \leftarrow |\{\texttt{str}(T) \mid \langle \mathcal{P}, S\rangle \Longrightarrow T\}|$
15 $\quad\quad d \leftarrow \beta / C(\langle S, S'\rangle)$
16 $\quad\quad$ **if** $\langle S, S'\rangle$ **in** D **then**
17 $\quad\quad\quad d \leftarrow d \times \theta$
18 $\quad\quad f' \leftarrow f' + \max\{0, d - c\}$
19 $\quad f \leftarrow \max\{f, f'\}$
20 **return** f

Figure 12. Pseudo-code of $\text{fitness}(\mathbb{W}, \mathbb{S}, D, E)$

Note that it is possible to run more iterations to obtain better candidate patch sets. In practice, we find that the work set $\mathbb{W}$ always converges after five iterations in our experiments. We do not observe any useful transforms that can only be generalized from more than five training AST slice pairs in our experiments.

Figure 12 presents the pseudo-code for the fitness function Genesis uses in its sampling algorithm. The function first computes for each training and validation pair, the number of candidate subsets that produce a transform that covers the pair (see lines 1-8). In this function, a transform covers a AST slice pair if the transform can generate the corresponding change for the slice and the size of the search space derived from this transform is less than a threshold β (see line 5). We empirically set β to 5×10^4 for all experiments we performed.

The algorithm then computes the score for a transform $\mathcal{P}$ as follows. For each validation pair $\langle S, S'\rangle$ in E that $\mathcal{P}$ covers, it gets a bonus score $\max\{\beta / C(\langle S, S'\rangle) - c, 0\}$, where $C(\langle S, S'\rangle)$ is the number of candidate subsets in $\mathbb{W}$ that cover the pair and c is the size of the number of candidate

changes of applying $\mathcal{P}$ to S. The intuition here is to obtain a more diverse set of candidate transforms, i.e., a transform that covers a pair which is also covered many other transforms should receive much lower score than a transform that covers a pair which is not covered by any other transform.

For each training pair $\langle S, S' \rangle$ in D, the bonus score is $\max\{\beta / C(\langle S, S' \rangle) \times \theta - c, 0\}$ instead. We empirically set $\theta = 0.1$ in our experiments. The intuition here is that the pairs in the training data should provide much lower scores than the pairs in validation set to avoid overfitting.

There are three optimizations in the Genesis implementation for the above sampling algorithm. Firstly, when Genesis computes $\mathbb{W}$ at lines 10-11, Genesis filters out many unproductive subsets that may yield intractable search space. For a new subset A, if another subset $B \subseteq A$ was already discarded because the space sizes of obtained transforms from B are more than β, then Genesis discards A immediately without adding it to $\mathbb{W}$ at line 11.

Secondly, for the computation of the search space size of the transform (i.e., the number of candidate changes derived from the transform) at line 5 and line 14 in Figure 12, Genesis computes an estimated size instead of generating each AST tree one by one. This estimation assumes that all generators in the transform generate binary trees. This optimization trades the accuracy of the fitness score computation for performance. Thirdly, Genesis computes C in Figure 12 at the start of each iteration of the sampling algorithm to avoid redundant computation during each invocation of the $\mathrm{fitness}()$ function.

3.5 Search Space Inference Algorithm

ILP Formulation: Given a candidate set of transforms $\mathbb{P}'$, the goal is to select a subset $\mathbb{P}$ from $\mathbb{P}'$ to form the result search space. We formulate the trade-off of the search space design between the coverage and the tractability as an integer linear programming (ILP) problem.

Figure 13 presents the ILP formulation of the transform selection problem. $C_{i,j}$ corresponds to the space size derived from the j-th transform when applying to the i-th AST slice pair in the validation set. $G_{i,j}$ indicates whether the space derived from the j-th transform contains the corresponding change for the i-th AST slice pair.

The variable x_i indicates whether the result search space covers the i-th AST slice pair and the variable y_i indicates whether the ILP solution selects the i-th transform. The ILP optimization goal is to maximize the sum of x, i.e., the total number of covered AST pairs in the validation set.

The first group of constraints is for tractability. The i-th constraint specifies that if the derived final search space size (i.e. $\Sigma_{j=1}^{k} C_{i,j} y_j$), when applied to the i-th AST pair in E, should be less than β if the space covers the i-th AST pair (i.e. $x_i = 1$) or less than ζ if the space does not cover the i-th AST pair (i.e. $x_i = 0$). In Genesis, $\beta = 5 \times 10^4$ and $\zeta = 10^8$. The second group of constraints is for coverage. The i-th constraint specifies that if the final search space covers the i-th AST slice pair in E (i.e. $x_i = 1$), then at least one of the selected transforms should cover the i-th pair.

$$
\begin{aligned}
&\mathbb{P}' = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k\} \\
&E = \{\langle S_1, S_1' \rangle, \ldots, \langle S_n, S_n' \rangle\} \\
&C_{i,j} = |\{\mathsf{str}(T) \mid \langle \mathcal{P}_j, S_i \rangle \Longrightarrow T\}| \\
&G_{i,j} = \begin{cases} 1 & \langle \mathcal{P}_j, S_i \rangle \Longrightarrow_{\texttt{slice}} S_i' \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Variables: x_i, y_i
Maximize: $\sum_{i=1}^{n} x_i$
Satisfy:
$\forall i \in \{1, \ldots, n\} : \zeta - (\zeta - \beta) \cdot x_i - \sum_{j=1}^{k} C_{i,j} y_j \geq 0$
$\forall i \in \{1, \ldots, n\} : \sum_{j=1}^{k} G_{i,j} y_j - x_i \geq 0$
$\forall i \in \{1, \ldots, n\} : x_i \in \{0, 1\}$
$\forall i \in \{1, \ldots, k\} : y_i \in \{0, 1\}$

Result Transform Set: $\mathbb{P} = \{\mathcal{P}_i \mid y_i = 1\}$

Figure 13. Integer linear programming formulas for selecting transforms given a set of candidate transforms $\mathbb{P}'$ and a validation set of AST slice pairs E

Input : a set of training AST slice pairs D
Output: a set of transforms that generate a search space
1 Remove a subset of D from D to form the validation set E
2 $\mathbb{P}' \leftarrow \mathrm{sample}(D, E)$
3 Solve ILP in Figure 13 to obtain $\mathbb{P}$
4 **return** $\mathbb{P}$

Figure 14. Search space inference algorithm

The second group of constraints is for coverage. The i-th constraint specifies that if the final search space covers the i-th AST slice pair in E (i.e. $x_i = 1$), then at least one of the selected transforms should cover the i-th pair.

Genesis also implements an alternative ILP formulation which considers also the patches in the training set and maximizes the number of covered training patches when two different solutions have the same number of covered validation patches. We empirically find that this alternative formulation tends to produce better search spaces when the validation set is small. See Section 5.

Inference Algorithm: Figure 14 presents the high-level pseudo-code of the Genesis inference algorithm. Starting from a training set of AST slice pairs D, Genesis first removes 25% of the AST slice pairs from D to form the validation set E. It then runs the sampling algorithm to produce a set of candidate transforms $\mathbb{P}'$. It finally solves the above ILP with Gurobi [34], an off-the-shelf solver, to obtain the set of transforms $\mathbb{P}$ that forms the result search space.

3.6 Implementation

We have implemented the Genesis inference algorithm for Java programs. We use the spoon library [51] to parse Java programs to obtain Java ASTs. We next discuss several ex-

tensions of the above inference algorithm for handling Java programs.

Semantic Checking: Genesis performs type checking in its implementation of the generation operators for generators and transforms. Genesis will discard any AST tree or AST forest that cannot pass Java type checking. Genesis also performs semantic checking to detect common semantic problems like undefined variables, uninitialized variables, etc..

Identifiers and Constants: The CFG for Java has an infinite set of terminals, because there are an infinite amount of possible variables, fields, functions, and constants. For the kind of generators that enumerate all possible AST forests (i.e., $b = 0$), Genesis does not generate changes that import new packages and or changes that introduce new local variables (even if a change introduces new local variables, it is typically possible to find a semantic equivalent change that does not). Therefore Genesis only considers a finite set of possible variables, fields, and functions.

Genesis also extends enumeration-based generators so that each generator has an additional set to track the constant values that the generator can generate. For the generation operator of a generator, Genesis will only consider finite constant values that are 0, 1, `null`, `false`, or any value that is inside the tracked set of the generator.

Many string constants in Java programs are text messages (e.g., the message in throw statements). These constants may cause a sparsity problem when Genesis computes the set of the allowed constants during the generalization process. To avoid such sparsity problems, Genesis detects such string constants and convert them to a special constant string — the specific string values are typically not relevant to the overall correctness of the programs.

Identifier Scope: Genesis exploits the structure of Java programs to obtain more accurate generators. Each enumeration-based generator (i.e., $b = 0$) tracks separate bounds for the number of variables and functions it uses inside the original slice, from the enclosing function, from the enclosing file, and from all imported files. For example, a generator may specify that it will only use up to two variables from the enclosing function in the generated AST forests. Similarly, each copy-based generator (i.e., $b = 0$) has additional flags to determine whether it copies code from the original code, the enclosing function, or the entire enclosing source file.

Code Style Normalization: Genesis has a code style normalization component to rewrite programs in the training set while preserving semantic equivalence. The code style normalization enables Genesis to find more common structures among ASTs of training patches and improves the quality of the inferred transforms and search spaces.

Input : the original program p, the validation test suite V, and the set of the transformation patterns of the search space $\mathbb{P}$
Output: the list of generated patches

```
1 𝕊 ← localization(p, V)
2 G ← emptylist
3 for S in 𝕊 do
4     for 𝒫 in ℙ do
5         for p′ in {str(T) | ⟨𝒫, S⟩ ⟹ T} do
6             if validate(p′, V) then
7                 Append p′ to G
8 return G
```

Figure 15. Patch generation algorithm

4. Patch Generation

Figure 15 presents the Genesis patch generation algorithm. The Genesis error localization algorithm (line 1) produces a ranked list of suspicious locations (as AST snippets) in the original program p. Genesis applies each transform in $\mathbb{P}$ to each suspicious AST snippet $S \in \mathbb{S}$ to obtain candidate patches p' (lines 3-5). It validates each candidate patch against the test cases and appends it to the returned patch list if it passes all test cases (lines 6-7). Our current implementation supports any Java application that operates with the Apache maven project management system [2] and JUnit [19] testing framework.

Error Localization: Genesis is designed to work with arbitrary error localization algorithms. Our current implementation starts with stack traces generated from test cases that trigger the null pointer or out of bounds access error. It extracts the top ten stack trace entries and discards any entries that are not from source code files in the project (as opposed to external libraries or JUnit). For each entry it finds the corresponding line of code in a project source code file and collects that line as well as the 50 lines before and after that line of code.

For each of the collected lines of code, Genesis first computes a suspiciousness score between 0 and 0.5. The line of code given by the first stack trace entry has a suspiciousness score of 0.5, with the score linearly decreasing to zero as the sum of the distance to the closest line of code from the stack trace and the rank of that line within the stack trace increases. Genesis prioritizes lines containing if, try, while, or for statements by adding 0.5 to their suspiciousness scores. The final scores are in the range of 0 to 1. Genesis prioritizes lines of code for patch generation according their final scores.

Space Exploration Order: Genesis applies its search space in order to each of the suspicious statements in the ranked list returned by the error localization algorithm. For each

transform, Genesis computes a cost score which equals to the average number of candidate patches the transform need to generate to cover a validation case. For each suspicious statement, Genesis prioritizes candidate patches that are generated by those transforms with lower cost scores.

5. Experimental Results

We next present experimental results of Genesis.

5.1 Methodology

Collect Training Patches and Errors: We developed a script that crawled the top 1000 github Java projects (ranked by number of stars), a list of 50968 github repositories from the MUSE corpus [21], and the github issue database. The script crawls the repositories and issues and it collects a project revision if 1) the project uses the apache maven management system [2], 2) we can use maven 3.3 to automatically compile both the current revision and the parent revision of the current revision (in the github revision tree) in our experimental environment, 3) we can use the spoon library [51] to parse the source code of both of the two revisions into AST trees, 4) the issue discussion or the commit message of the current revision contains certain keywords to indicate that the revision corresponds to a patch for NPE, OOB, or CCE errors, and 5) the revision changes only one source file (because revisions that change more than one source file often correspond to composite changes and not just patches for NPE, OOB, or CCE errors).

For NPE errors, the scripts search for keywords "null deref", "null pointer", "null exception", and "npe". For OOB errors, the scripts search for keywords "out of bounds", "bound check", "bound fix", and "oob". For CCE errors, the scripts search for keywords "classcast exception", "cast check", and "cast fix". We manually inspected the retrieved revisions to discard revisions that do not correspond to fixing NPE, OOB, CCE errors. Note that we discard many repositories and revisions because we are unable to automatically compile them with maven, i.e., they do not support maven or they have special dependencies that cannot be automatically resolved by the maven system.

We focus on NPE, OOB, and CCE errors because we want to compare the patch generation results of Genesis with PAR, a previous patch generation system. PAR [36] contains manual transform templates for NPE, OOB, and CCE errors. In fact, a review of PAR system finds out that most reported patches in the PAR paper are generated by its manual templates for NPE and OOB errors [49].

In total we collected 1012 human patches from 372 different applications. These patches include 503 null pointer error (NPE) patches, 212 out of bound error (OOB) patches, and 303 class cast error (CCE) patches. Note that there are six patches whose revision commit logs contain keywords for two kinds of errors and we count them as patches for both of the two kinds.

Benchmark Errors: We then went over each of the collected patches with another script we developed. The script collects a revision if 1) the revision has a JUnit [19] test suite in the repository that Genesis can run automatically, 2) the JUnit test suite contains at least one test case that can expose and reproduce the error in our experimental environment, 3) the JUnit test suite contains at least 50 test cases in total, and 4) the test suite does not cause non-deterministic behaviors. This script collects in total 49 reproducible errors from the collected 1012 patches. It includes 20 NPE errors, 13 OOB errors, and 16 CCE errors. These are the *benchmark errors*.

Partition into Training and Validation: We partition the collected 1012 patches into training, validation, and benchmark patches as follow. We first removed the 49 benchmark errors, leaving 963 remaining patches. We randomly partition the remaining patches into 722 training patches and 241 validation patches (the learning algorithm uses validation patches to evaluate candidate transforms, see Section 3.5).

We also performed experiments to run learning algorithm on training patches of NPE, OOB, and CCE errors separately to to infer targeted search spaces for each class of errors. For such experiments, we first removed all of the benchmark errors from the collected 503 NPE patches, 212 OOB patches, and 303 CCE patches, respectively. We then randomly partitioned the 483, 199, and 287 remaining patches into 362 NPE, 149 OOB, and 215 CCE training patches and 121 NPE, 50 OOB, and 72 CCE validation patches, respectively.

Search Space Inference: We run the inference algorithm on the partitioned training and validation patches for each class of errors to infer targeted search spaces. To evaluate the proposed optimization for working with small validation sets in Section 3.5, for each kind of error we run the ILP solver twice to produce two targeted search spaces. Genesis produces the first search space by solving an ILP formulation that considers the coverage of only the validation patches. Genesis produces the second space by solving an ILP formulation that considers both the validation and training patches (see Section 3.5). We set the work list size α of the sampling algorithm (see Section 3.4) to 1000 for the targeted inference runs.

We also run the Genesis search space inference algorithm on the 722 training patches and 241 validation patches to infer a single composite search space for all NPE, OOB, and CCE errors. For this run, Genesis uses the ILP formulation that considers both the training and validation patches. We run the inference algorithm with 36 threads in parallel on an Amazon EC2 c4.8xlarge instance with Intel Xeon E5-2666 processors, 36 vCPU, and 60GB memory. We set the work list size α of the sampling algorithm (see Section 3.4) to 500 for the composite inference run.

PAR Template Implementation: PAR [36] deploys a set of transform templates to fix bugs in Java programs, with the templates manually derived by humans examining multiple real-world patches. The description of PAR templates is publicly available at the PAR website [23], but despite our repeated requests to the PAR authors, we are unable to obtain the original implementation of PAR.

To this end, we implemented PAR templates for NPE, OOB, and CCE errors under our own patch generation framework. For NPE error, we implemented the PAR template "Null Checker", which inserts if statements that either 1) skip a statement with a null dereference or 2) returns a default value from a function before a null dereference. For OOB errors, we implemented five PAR templates "Range Checker", "Collection Size Checker", "Lower Bound Setter", "Upper Bound Setter", and "Off-by-one Mutator". "Range Checker" and "Collection Size Checker" insert if statements that either 1) skip a statement with index out-of-bound access for array or collection or 2) returns a default value from a function before an out-of-bound access. "Lower Bound Setter" and "Upper Bound Setter" change the value of an out-of-bound index variable back to its lower bound or upper bound value. "Off-by-one Mutator" increases or decreases the value of an index variable by one. For CCE errors, we implemented three templates "Class Cast Checker", "Caster Mutator", and "Castee Mutator". "Class Cast Checker" insert if statement with *instanceof* checks that either 1) skip a statement triggering a class cast exception or 2) returns a default value from a function before a class cast exception. "Caster Mutator" changes the cast type that an expression converts to in a cast statement. "Castee Mutator" replaces the castee expression in a cast statement with a variable in similar name.

Note that the PAR paper [36] and the PAR website [23] are the only references that we find for PAR templates. Some PAR template descriptions are ambiguous (e.g., a PAR template inserts an if guarded return statement to return a "default" value from the current function, but there is no clear definition of "default" in the PAR paper [36] and the PAR website [23]). We implemented such ambiguous PAR templates, with our best efforts, in a way that will enable the templates to generate correct patches for as many benchmark errors as possible. Also note that there are eight PAR templates we do not consider in our experiments, because the eight templates are not for NPE, OOB, and CCE errors. Our manual analysis indicate that these templates can generate no additional correct patches for our benchmark errors and considering those templates only harms the performance of PAR templates.

Patch Generation: We run Genesis with each of the inferred targeted search spaces on the corresponding kind of benchmark errors (the NPE search spaces on 20 NPE errors, the OOB search spaces on 13 OOB errors, and the CCE search spaces on 16 CCE errors, respectively). We run the Genesis patch generation system with the inferred composite search space on all 49 benchmark errors. For comparison, we further run the patch generation system with the implemented PAR templates on all benchmark errors. We performed all of the patch generation experiments on Amazon EC2 m4.xlarge instances with Intel Xeon E5-2676 processors, 4 vCPU, and 16 GB memory. We set a time limit of five hours, i.e., we terminate a patch generation process if it does not finish the exploration of the search space in five hours.

For each of the benchmark errors, we manually analyze the root cause of the error and the corresponding developer patch in the repository. For each patch generation run, Genesis or PAR produces a ranked list of validated patches and we analyze each validated patch one by one in order until we identify the first correct patch (i.e., the patch correctly fixes the error for all possible inputs) in the produced ranked list if any.

Note that the corresponding developer patches for several benchmark errors emit text error/log messages if certain conditions are true. Genesis and PAR validated patches do not attempt to generate the text error messages. In our experiments, we count such a Genesis and PAR patch correct if the patch semantically differs with the corresponding developer patch only in such text error/log messages.

In principle, determining patch correctness can be difficult. We emphasize that this is not the case for the patches and errors in our experiments. The correct behavior for all of the errors is clear, as is patch correctness and incorrectness. In fact, all of those generated patches that our analysis identifies as correct patches are in fact semantically equivalent to the corresponding developer patches or differ only in error/log messages.

5.2 Inference Results

Table 1 presents the Genesis inference results for the composite search space and the six targeted search spaces for NPE, OOB, and CCE errors. "VO" denotes the targeted search spaces that Genesis generates with an ILP formulation that considers validation patches only. "TV" denotes the targeted search spaces that Genesis generates with an ILP formulation that considers both training and validation patches.

The second row presents the number of transforms produced by the Genesis sampling algorithm (Section 3.4). The third and fourth rows presents the number of training and validation patches these post-sampling transforms can generated if applied to the pre-patch ASTs of the training and validation patches. The fifth row presents the number of selected transforms in each of the inferred search spaces. The

	NPE (VO)	NPE (TV)	OOB (VO)	OOB (TV)	CCE (VO)	CCE (TV)	Composite
Transforms after Sampling	1354		978		769		577
Covered Training after Sampling	248 of 362		101 of 149		87 of 215		443 of 722
Covered Validation after Sampling	58 of 121		14 of 50		14 of 72		87 of 241
Final Inferred Transforms	13	52	9	35	10	28	108
Covered Training in Space	144 of 362	209 of 362	31 of 149	88 of 149	34 of 215	78 of 215	362 of 722
Covered Validation in Space	57 of 121	57 of 121	13 of 50	11 of 50	12 of 72	12 of 72	80 of 241
Total Inference Time	1464m		1181m		861m		2886m

Table 1. Transform and search space inference results

Type	Errors	Genesis						Genesis (Composite)			PAR	
		Correct		Top 3		Top 10		Correct	Top 3	Top 10	Correct	Top 3
		VO	TV	VO	TV	VO	TV					
NPE	20	**13**	11	**12**	8	**12**	11	11	10	11	7	7
OOB	13	5	**6**	3	**4**	5	**5**	6	5	5	4	4
CCE	16	3	**5**	2	**3**	3	**4**	4	2	4	0	0
Total	49	**24**(21/22)		**19**(17/15)		**21**(20/20)		21	17	20	11	11

Table 2. The summary of patch generation results.

sixth and seventh rows present the number of training and validation patches each search space can generate if applied to the pre-patch ASTs of the patches. The ninth row presents the running time of inferring each search space. In our experiments, the inference takes less than 50 hours. The time is dominated by the sampling algorithm and the generalization operation over training patches. Solving the ILPs takes less one minute for all four search spaces.

The results highlight the ability of our ILP formulation to navigate the search trade-off between the coverage and tractability. Over 85% of validation patches that are covered by the candidate transforms are also covered by the result search space produced by the Genesis ILP solver. The Genesis ILP formulation enables Genesis to select a productive subset of transforms among candidate transforms to cover these validation patches while satisfying the tractability constraints. The results show that the Genesis inference algorithm achieves higher coverage of validation patches on NPE errors than it achieves on OOB and CCE errors. One possible explanation is that human patches for OOB and CCE errors tend to be more complicated and diverse than patches for NPE errors. It is therefore more challenging to extract and infer transforms to summarize human OOB and CCE patches.

5.3 Patch Generation Results

Table 2 summarizes the patch generation results for Genesis and PAR. See Appendix A for the detailed results on each benchmark error. The first column "Type" presents the type of error (NPE, OOB, or CCE). The second column "Number of Errors" presents the total number of errors of each type. The third to eighth columns "Genesis" present the Genesis results with targeted search spaces (applying a targeted NPE search space to NPE errors, a targeted OOB space to OOB errors, and a targeted CCE search space to CCE errors). The "VO" column presents the results of the search spaces that Genesis generates with an ILP formulation that considers validation patches only. The "TV" column presents the results of the search spaces that Genesis generates with an ILP formulation that considers both training and validation patches. Each entry presents the number of benchmark errors for which the corresponding system generates at least one correct patch. Each entry in the "Top 3" columns presents the number of benchmark errors for which a correct patch is ranked top three in the generated patches. Each entry in the "Top 10" columns presents the number of errors for which a correct patch is ranked top ten in the generated patches. The ninth to eleventh columns "Genesis (Composite)" present the Genesis results of applying the inferred composite search space to all benchmark errors. The twelfth and thirteenth columns "PAR" present the patch generation results of applying PAR templates.

With the "NPE (VO)", "OOB (TV)", and "CCE (TV)" search spaces, Genesis generates correct patches for 24 out of 49 benchmark errors, with 19 in the top 3 and 21 in the top 10 generated patches. Compared to the standard ILP formulation that only considers validation patches, the formulation that considers both training and validation patches enables Genesis to generate correct patches for two more OOB errors, two more CCE errors, but two fewer NPE errors. One explanation is that for NPE errors we have a relatively large number of validation patches. Considering only validation patches for the ILP formulation helps reduce the overfitting of the inferred search space. On the other hand, for OOB and

CCE errors we have relatively small number of validation patches. Considering both training and validation patches enables Genesis to cover those cases that are present in the training patches only.

Our results highlight the effectiveness of the Genesis inferred search space in comparison with the PAR manually defined templates. Genesis generates correct patches for six more NPE errors, two more OOB errors, and five more CCE errors than PAR. Also Genesis generates correct patches ranked as top three among generated patches for five more NPE errors and three more CCE errors than PAR.

Our results also highlight the ability of Genesis to infer a single composite search space from training patches for multiple kinds of errors together. Genesis with the composite search space is able to generate correct patches for 21 out of the 49 benchmark errors. One interesting phenomenon is that for the 21 errors the composite search space generates correct patches ranked as top three and top ten for more errors than the targeted search spaces do. This is because when Genesis infers the single composite search space for all three kinds of errors, Genesis prunes away relatively less efficient transforms that are selected in the targeted search spaces to maintain tractability. Pruning these transforms enables Genesis to generate fewer validated but incorrect patches.

NPE Patch Categories: Correct patches for nine NPE cases insert if statements with a null pointer check. Then if the check succeeds, the patches either 1) return a synthesized value or void (for five out of the nine cases), 2) throw a new synthesized exception (for two cases), or 3) skip a statement with a null pointer error (for two cases). Correct patches for two NPE cases modify an existing boolean expression by conjoining or disjoining a synthesized expression. The correct patch for one case inserts a call statement that is copied from an existing statement in the enclosing function. The correct patch for the remaining one NPE case deletes an assignment statement that assigns a null value to a variable.

OOB Patch Categories: Correct patches for five OOB cases insert if statements with a comparison that checks for an out of bounds access. Then if the check succeeds, the patches either return a synthesized value or void (for three out of five cases) or break from the enclosing loop (for two cases). The correct patch for the remaining one OOB case replaces a constant zero with a local variable in an conditional expression.

CCE Patch Categories: The correct patch for one CCE case changes the class type of a local variable declaration to the super class of its original type. The correct patch for one CCE case modifies the target class type of an `instanceof` condition to tighten the condition. Correct patches for two CCE cases insert a try-catch statement to catch and ignore class cast errors. The correct patch for the remaining CCE case wraps an expression with a new method call to appropriately convert the expression value.

Error Localization Oracle: To isolate the effect of error localization, we also run Genesis and PAR with an oracle that identifies, for each error, the correct line of code to patch. With the oracle, Genesis ("OOB (TV)") generates correct patches for two more OOB errors (RoaringBitmap [24] revision 29c6d5 and maven-shared [3] revision 77937e) and ("CCE (TV)") one more CCE error (fastjson [8] revision c88687). PAR is unable to generate any additional correct patch with the oracle.

5.4 Discussion

The inferred Genesis search spaces generate correct patches for significantly more benchmark errors than manually defined PAR templates. The reason is that there are many design choices and parameters for a transform (template) and it is difficult for human to tune them manually.

For example, consider an inferred transform for NPE errors $A \rightarrow$ `if (B!=null) {A}`, which adds an if statement to guard an existing statement A. There are many design questions for this transform: what is allowed in the checked expression B? Should we just allow local variables and constants or do we allow complicated expressions? If we allow complicated expressions, how we are going to bound the expressions? If A is already an if statement, should we consider to change its condition instead of warpping it with a new if statement? All these questions correspond to design choices and parameters in the transform. Suboptimal choices often cause unproductive search spaces that do not contain enough useful patches or that are too large to explore. It is difficult for manually defined patch templates to capture these design choice complexities.

Genesis captures these complexities with its novel definition of code transforms, where different design choices correspond to structures in template ASTs and parameters in generators. The sampling algorithm first automatically samples candidate transforms with productive parameters from the training set. The ILP formulation then automatically selects a subset of the candidate transforms that deliver productive trade-off between coverage and the tractability. On the other hand, PAR templates are designed to only capture most common patterns of human patches. The resulting PAR search space is relatively narrow and misses many important useful patterns.

Here is how this issue plays out in our experiments. For NPE errors, PAR manual templates consider only two kinds of transforms "add an if statement to guard an existing statement" and "add an if guarded return statement" with null check boolean conditions. For the 13 NPE errors for which Genesis generates correct patches, two NPE errors (spring-data-rest [26] revision aa28ae and error-prone [7] re-

vision 370933) are outside PAR's space because the correct patches add "if (...) throw ..." statements; three NPE errors (DataflowJavaSDK [5] revision c06125, javaslang [15] revision faf9ac, and Activiti [1] revision 3d624a) are outside PAR's space because the correct patches change condition expressions in a non-trivial way that is not equivalent to add an if-guard; one NPE error (HikariCP [12] revision ce4ff9) is outside PAR's space because PAR templates do not generate patches that directly remove statements.

For OOB errors, PAR manual templates consider "add an if statement to guard an existing statement" and "add an if guarded return statement" with range check conditions. The templates also consider "increases or decreases a variable by one" and "add an if guarded assignment statement to enforce index lower and upper bounds of a variable". For six OOB errors for which Genesis generates correct patches, two OOB cases (jgit [16] revision 929862 and jPOS [18] revision df400a) are outside PAR's space, because correct patches change conditions in a way different from the standard range checks.

For CCE errors, PAR manual templates consider "add an if statement to gaurd an existing statement" and "add an if guarded return statement" with `instanceof` type checks. The PAR templates also include "change the casting type of a cast operator". The templates do not generate correct patches for any CCE erorrs. For the five errors for which Genesis generates correct patches, the correct patches for two of the five errors (jade4j [14] revision 114e88 and HdrHistogram [11] revision 030aac) change existing expressions in a way that is not equivalent to adding a type check guard. The correct patches for two of the five errors (htmlelements [13] revision bf3f27 and hamcrest-bean [10] revision 84586d) insert try-catch statement to catch and ignore class cast exceptions (the developers introduced these try-catch statements in the patched revision and these statements are still present in the latest revision of these repositories). The correct patch for the remaining error (jade4j [14] revision dd4739) modifies the declared type of a local variable to avoid class cast exceptions.

6. Related Work

Generate And Validate Systems: Generate and validate patch generation systems apply a set of transforms to generate a search space of candidate patches that are then evaluated against a set of inputs to filter out patches that produce incorrect outputs for the test inputs. Prophet [40] and SPR [41] apply a set of predefined parameterized transformation schemas to generate candidate patches. Prophet processes a corpus of successful human patches to learn a model of correct code to rank plausible patches; SPR uses a set of hand-code heuristics for this purpose. GenProg [39, 65], AE [64], and RSRepair [54] use a variety of search algorithms (genetic programming, stochastic search, random search) in combination with transforms that delete, insert, and swap existing program statements. Kali [55] applies a single transform that simply deletes code. All of these systems were evaluated on the same benchmark set [39]. For the 69 defects in this set (the set also contains 36 functionality changes), Prophet, SPR, Kali, GenProg, RSRepair, and AE generate correct patches for 15, 11, 2, 1, 2, and 2 defects, respectively. We attribute the relatively poor performance of Kali, GenProg, RSRepair, and AE to the fact that their search spaces do not appear to contain correct patches for the remaining defects in the set [41, 55]. Like Prophet, history-driven program repair [38] uses information from previous human patches to rank candidate patches generated by human-specified transforms.

PAR [36] deploys a set of patterns to fix bugs in Java programs, with the patterns manually derived by humans examining multiple real-world patches. The PAR null pointer checker pattern inserts if statements that either 1) skip a statement with a null dereference or 2) returns a default value before a null dereference. The PAR range checker pattern inserts bounds checks. Genesis automatically infers a larger and richer set of transforms that generate all of the patches generated by the PAR manually-derived patterns and more. In particular, the generated Genesis patch in Section 2 is outside the PAR search space.

Genesis differs from all of these systems in that it does not work with a fixed set of human-specified transforms. It instead automatically processes patches from repositories to automatically infer a set of transforms that together define its patch search space.

Constraint Solving Systems: Prophet [40], SPR [41], Qlose [29], NOPOL [33], SemFix [50], and Angelix [45] all use constraint solving to generate new values for potentially faulty expressions (often faulty conditions). ClearView [53] enforces learned invariants to eliminate security vulnerabilities. Angelic Debugging [28] finds new values for potentially incorrect subexpressions that allow the program to produce correct outputs for test inputs.

PHPQuickFix and PHPRepair use string constraint-solving techniques to automatically repair PHP programs that generate HTML [61]. By formulating the problem as a string constraint problem, PHPRepair obtains sound, complete, and minimal repairs to ensure the patched php program passes a validation test suite. Specification-based data structure repair [30, 31, 35, 66] takes a data structure consistency specification and an inconsistent data structure, then synthesizes a repair that produces a modified data structure that satisfies the consistency specification.

Genesis differs from all of these systems in that it works with automatically inferred transforms, with generators playing the role of constraint solvers to generate ex-

pressions that enable parameterized transforms to produce correct patches.

Repair with Formal Specifications: It is possible to leverage formal specifications to generate patches that produce a patched program that satisfies the specification [37, 52, 60]. One difference is that Genesis works with large real world applications where formal specifications are typically not available.

Probabilistic Model for Programs: There is a rich set of work on applying probabilistic model and machine learning learning techniques for programs, specifically, for identifying correct repairs [40], code refactoring [57], and code completion [27, 56, 58]. These techniques learn a probabilistic model from a training set of patches or programs and then use the learned model to identify the best repair or token for a defective or partial program. In contrast, instead of learning individual concrete patches, Genesis has the high-order goal of inferring transforms that can be applied to a new bug to generate a set of candidate patches. Genesis does not use probabilistic models. It instead obtains candidate transforms with a novel generalization algorithm and formulates the transform selection problem as an integer linear programming.

Repair Model Mining: Martinez and Monperrus manually analyze previous human patches to mine repair models for program repair systems. They manually define a set of transforms and then classify the patches into the defined transforms based on the kind of modification operations of the patches [44]. In contrast, Genesis does not work with any predefined transform and automatically infers the set of transforms from a set of human patches.

Systematic Edit: SYDIT [47] and Lase [48] extract edit scripts from one (SYDIT) or more (Lase) example edits. The script is a sequential list of modification operations that insert statements or update existing statements. SYDIT and Lase then generate changes to other code snippets in the same application with the goal of automating repetitive edits. RASE [46] uses Lase edit scripts to refactor code clones. FixMeUp [63] works with access control templates that implement policies for sensitive operations. Using these templates, FixMeUp finds unprotected sensitive operations and inserts appropriate checks. An analysis of the application can extract an application-specific template [62], which FixMeUp can then apply across the same application. Genesis differs in that it processes multiple patches from multiple applications to derive generalized application-independent transforms that it can apply to fix bugs in yet other applications. The Genesis transforms include generators so that transforms can generate multiple candidate patches (as opposed to a single edit as in SYDIT, Lase, and FixMeUp).

Dynamic Recovery: Failure-oblivious computing [59] discards out of bounds writes and manufactures values for out of bounds reads. RCV [43] returns zero as the result of null pointer dereferences and divide by zero errors. APPEND [32] detects attempted null pointer dereferences and applies recovery actions such as creating a default object to replace the null pointer. In all cases the goal is to enable successful (but not necessarily correct) continued execution.

Genesis, in contrast, learns transforms and applies these transforms to derive a patched program without the null pointer or out of bounds access error — the goal is to obtain a correct patch, not simply continued execution via runtime recovery. The automatically inferred Genesis templates provide a broader and more sophisticated set of techniques for dealing with null pointer dereferences and out of bounds accesses.

7. Conclusion

Previous generate and validate patch generation systems work with a fixed set of transforms defined by their human developers. By automatically inferring transforms from sets of successful human patches, Genesis makes it possible to leverage the combined expertise and patch generation strategies of developers worldwide to automatically patch bugs in new applications.

References

[1] Activiti. http://activiti.org/.

[2] Apache maven. https://maven.apache.org/.

[3] Apache maven project - shared components. http://maven.apache.org/shared/.

[4] Bukkit. https://bukkit.org.

[5] Dataflow java sdk. https://github.com/GoogleCloudPlatform/DataflowJavaSDK.

[6] dyn.js. http://dynjs.org/.

[7] Error Prone. http://errorprone.info/.

[8] fastjson. https://github.com/alibaba/fastjson.

[9] GitHub. https://github.com/.

[10] Hamcrest Bean. https://github.com/eXparity/hamcrest-bean.

[11] Hdrhistogram. https://github.com/HdrHistogram/HdrHistogram.

[12] HikariCP. https://brettwooldridge.github.io/HikariCP/.

[13] Html elements framework. https://github.com/yandex-qatools/htmlelements.

[14] jade4j. https://github.com/neuland/jade4j.

[15] Javaslang. http://www.javaslang.io/.

[16] JGit - Eclipse. https://eclipse.org/jgit/.

[17] Joda-Time. http://www.joda.org/joda-time/.

[18] jPOS. http://www.jpos.org/.

[19] Junit. http://junit.org/.

[20] Mapstruct - java bean mappings, the easy way! http://mapstruct.org/.

[21] Mining and understanding software enclaves (muse) program. https://wiki.museprogram.org.

[22] OrientDB. http://orientdb.com/orientdb/.

[23] Pattern-based automatic program repair (par). https://sites.google.com/site/autofixhkust/home.

[24] RoaringBitmap. https://github.com/RoaringBitmap/RoaringBitmap.

[25] Simple, intelligent, object mapping. http://modelmapper.org/.

[26] Spring data REST. http://projects.spring.io/spring-data-rest/.

[27] P. Bielik, V. Vechev, and M. Vechev. Phog: Prababilistic model for code. In *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

[28] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11', pages 121–130, New York, NY, USA, 2011. ACM.

[29] L. D'Antoni, R. Samanta, and R. Singh. Qlose: Program repair with quantitative objectives. In *Computer-Aided Verification (CAV)*, 2016.

[30] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244, 2006.

[31] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12):931–951, 2006.

[32] K. Dobolyi and W. Weimer. Changing java's semantics for handling null pointer exceptions. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 0:47–56, 2008.

[33] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan. Automatic repair of real bugs: An experience report on the defects4j dataset. *CoRR*, abs/1505.07002, 2015.

[34] I. Gurobi Optimization. Gurobi optimizer reference manual, 2015.

[35] S. Khurshid, I. García, and Y. L. Suen. Repairing structurally complex data. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, pages 123–138, 2005.

[36] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13', pages 802–811. IEEE Press, 2013.

[37] E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *Computer-Aided Verification (CAV)*, 2015.

[38] X. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016*, pages 213–224, 2016.

[39] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 3–13. IEEE Press, 2012.

[40] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 298–312.

[41] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[42] F. Long and M. C. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 702–713, 2016.

[43] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14', pages 227–238, New York, NY, USA, 2014. ACM.

[44] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[45] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 691–701, 2016.

[46] N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15', pages 392–402, Piscataway, NJ, USA, 2015. IEEE Press.

[47] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: Generating program transformations from an example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11', pages 329–342, New York, NY, USA, 2011. ACM.

[48] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 502–511, 2013.

[49] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the

problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 234–242, New York, NY, USA, 2014. ACM.

[50] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13', pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[51] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, page na, 2015.

[52] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Trans. Softw. Eng.*, 40(5):427–449, May 2014.

[53] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102. ACM, 2009.

[54] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, New York, NY, USA, 2014. ACM.

[55] Z. Qi, F. Long, S. Achour, and M. Rinard. An anlysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2015*, 2015.

[56] V. Raychev, P. Bielik, M. Vechev, and A. Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16', pages 761–774, New York, NY, USA, 2016. ACM.

[57] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15', pages 111–124, New York, NY, USA, 2015. ACM.

[58] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14', pages 419–428, New York, NY, USA, 2014. ACM.

[59] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[60] R. Samanta, O. Olivo, and E. A. Emerson. Cost-aware automatic program repair. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, pages 268–284, 2014.

[61] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 277–287, 2012.

[62] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 1069–1084, 2011.

[63] S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[64] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE'13*, pages 356–366, 2013.

[65] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09', pages 364–374. IEEE Computer Society, 2009.

[66] R. N. Zaeem, M. Z. Malik, and S. Khurshid. Repair abstractions for more efficient data structure repair. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, pages 235–250, 2013.

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
caelum-stella	2ec5459	1m	9545	9545	17m	6	<1m	1	51
caelum-stella	2d2dd9c	1m	6597	6597	24m	22	<1m	1	10
caelum-stella	e73113f	1m	6642	6642	24m	22	<1m	1	4
HikariCP	ce4ff92	3m	20294	20294	215m	39	167m	11	14633
nutz	80e85d0	2m	202795	93904	>5h	0	-	-	-
spring-data-rest	aa28aeb	7m	8439	8439	275m	7	56m	2	1501
checkstyle	8381754	2m	114736	76495	>5h	4	2m	1	359
checkstyle	536bc20	3m	126115	77864	>5h	6	<1m	1	7
checkstyle	aaf606e	2m	119541	85059	>5h	0	-	-	-
checkstyle	aa829d4	1m	0	0	<1m	0	-	-	-
jongo	f46f658	1m	64373	32199	>5h	3	-	-	-
DataflowJavaSDK	c06125d	3m	12010	12010	100m	1	4m	1	429
webmagic	ff2f588	<1m	26270	26270	99m	1	-	-	-
javapoet	70b38e5	<1m	22806	22806	72m	0	-	-	-
closure-compiler	9828574	3m	162227	12394	>5h	5	2m	1	794
truth	99b314e	<1m	8509	8509	15m	0	-	-	-
error-prone	3709338	2m	95533	781	>5h	3	<1m	1	72
javaslang	faf9ac2	<1m	191829	153888	>5h	6	<1m	3	407
Activiti	3d624a5	4m	67528	895	>5h	99	1m	2	74
spring-hateoas	48749e7	<1m	6965	6965	14m	8	<1m	1	62

Table 3. Experimental results of the Genesis "NPE (VO)" search space on NPE defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
Bukkit	a91c4c6	<1m	120682	120682	130m	6	<1m	1	39
RoaringBitmap	29c6d59	4m	246699	34787	>5h	0	-	-	-
commons-lang	52b46e7	2m	56393	56393	147m	0	-	-	-
HdrHistogram	db18018	<1m	105685	105685	300m	515	-	-	-
spring-hateoas	29b4334	<1m	67856	67856	124m	0	-	-	-
wicket	b708e2b	6m	88938	68158	>5h	28	136m	8	31573
coveralls-maven-plugin	20490f6	<1m	5533	5533	10m	0	-	-	-
named-regexp	82bdfeb	<1m	0	0	<1m	0	-	-	-
jgit	929862f	2m	59053	59053	152m	0	-	-	-
jPOS	df400ac	3m	129158	129158	264m	10	19m	1	11172
httpcore	dd00a9e	2m	112849	76295	>5h	78	<1m	1	93
vectorz	2291d0d	<1m	86488	86488	117m	21	27m	9	24927
maven-shared	77937e1	2m	0	0	<1m	0	-	-	-

Table 4. Experimental results of the Genesis "OOB (VO)" search space on OOB defects

A. Patch Generation Results per Benchmark Error

Tables 3, 4, and 5 present the results of Genesis patch generation, when run with targeted search spaces constructed from the validation-patch only ILP formulation, on our benchmark set of 20 NPE, 13 OOB, and 16 CCE errors, respectively. Tables 6, 7, and 8 similarly present the results of Genesis when run using targeted search spaces constructed from the ILP formulation which uses both the training and the validation set (see Section 3.5). Tables 9, 10, and 11 show the results when using the composite search space containing patches for all types of defects. Finally, Tables 12, 13, 14, contain the results of running our formulation of the PAR templates on our benchmark sets.

Each table contains one line for each error of its defect type. The "Init. Time" column presents the amount of time required to initialize the search for that error. The "Search Space Size" column presents the size of the search space for that error, the "Explored Space Size" column presents the

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
jade4j	dd47397	<1m	2442	2442	16m	2	3m	1	442
jade4j	114e886	<1m	4383	4383	26m	0	-	-	-
HdrHistogram	030aac1	<1m	588	588	1m	0	-	-	-
pdfbox	93c0b69	1m	2318	2318	4m	0	-	-	-
tree-root	fef0f36	<1m	558	558	<1m	0	-	-	-
spoon	48d3126	8m	0	0	<1m	0	-	-	-
pebble	942aa6e	2m	2081	2081	19m	0	-	-	-
fastjson	c886874	1m	9228	9228	17m	0	-	-	-
htmlelements	bf3f275	2m	1659	1659	9m	12	3m	7	284
spring-cloud-connectors	56c6eca	<1m	4328	4328	11m	0	-	-	-
joinmo	a5ee885	<1m	22935	22935	71m	5	-	-	-
buildergenerator	d9d73b3	<1m	566	566	1m	0	-	-	-
mybatis-3	809c35d	7m	6024	6024	39m	0	-	-	-
antlr4	9e7b131	3m	43	43	<1m	0	-	-	-
hamcrest-bean	84586d9	<1m	21705	21705	46m	6	<1m	1	37
raml-java-parser	49aab8f	<1m	370	370	2m	0	-	-	-

Table 5. Experimental results of the Genesis "CCE (VO)" search space on CCE Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
caelum-stella	2ec5459	1m	46240	46240	74m	21	<1m	1	40
caelum-stella	2d2dd9c	1m	24089	24089	44m	18	<1m	1	83
caelum-stella	e73113f	1m	24445	24445	46m	18	<1m	1	81
HikariCP	ce4ff92	3m	253513	66146	>5h	9	-	-	-
nutz	80e85d0	2m	559462	130027	>5h	0	-	-	-
spring-data-rest	aa28aeb	7m	53974	6651	>5h	0	-	-	-
checkstyle	8381754	2m	1130045	109538	>5h	17	12m	4	1719
checkstyle	536bc20	2m	1252145	106140	>5h	58	<1m	1	6
checkstyle	aaf606e	2m	1191987	118726	>5h	0	-	-	-
checkstyle	aa829d4	1m	0	0	<1m	0	-	-	-
jongo	f46f658	1m	240259	33510	>5h	2	-	-	-
DataflowJavaSDK	c06125d	3m	63391	58135	>5h	30	6m	1	3072
webmagic	ff2f588	1m	1166381	92138	>5h	0	-	-	-
javapoet	70b38e5	<1m	328306	182898	>5h	0	-	-	-
closure-compiler	9828574	3m	$> 10^6$	150055	>5h	2	3m	1	4
truth	99b314e	<1m	57401	57401	90m	0	-	-	-
error-prone	3709338	2m	536502	14681	>5h	7	3m	1	448
javaslang	faf9ac2	<1m	$> 10^6$	177886	>5h	146	1m	10	1465
Activiti	3d624a5	4m	866860	1719	>5h	87	5m	4	68
spring-hateoas	48749e7	<1m	18820	18820	45m	47	<1m	1	17

Table 6. Experimental results of the Genesis "NPE (TV)" search space on NPE defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
Bukkit	a91c4c6	<1m	291493	163821	>5h	3	<1m	1	39
RoaringBitmap	29c6d59	4m	553322	62426	>5h	0	-	-	-
commons-lang	52b46e7	2m	74203	5226	>5h	0	-	-	-
HdrHistogram	db18018	<1m	193578	181161	>5h	551	-	-	-
spring-hateoas	29b4334	<1m	66284	66284	50m	1	-	-	-
wicket	b708e2b	6m	119439	108523	>5h	34	112m	8	39486
coveralls-maven-plugin	20490f6	<1m	5252	5252	4m	0	-	-	-
named-regexp	82bdfeb	<1m	0	0	<1m	0	-	-	-
jgit	929862f	2m	144180	144180	215m	8	90m	3	57712
jPOS	df400ac	2m	179730	179730	213m	10	7m	1	12132
httpcore	dd00a9e	2m	265367	92815	>5h	151	<1m	1	93
vectorz	2291d0d	<1m	241445	241445	263m	49	77m	40	78156
maven-shared	77937e1	2m	0	0	<1m	0	-	-	-

Table 7. Experimental results of the Genesis "OOB (TV)" search space on OOB Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
jade4j	dd47397	<1m	64985	64985	223m	3	87m	1	26997
jade4j	114e886	<1m	115062	72499	>5h	1	229m	1	54148
HdrHistogram	030aac1	<1m	20361	20361	27m	69	4m	33	717
pdfbox	93c0b69	1m	66136	66136	147m	0	-	-	-
tree-root	fef0f36	<1m	22090	22090	17m	0	-	-	-
spoon	48d3126	9m	0	0	<1m	0	-	-	-
pebble	942aa6e	2m	183615	142738	>5h	0	-	-	-
fastjson	c886874	1m	1062660	129234	>5h	0	-	-	-
htmlelements	bf3f275	2m	52673	52673	128m	12	13m	7	7553
spring-cloud-connectors	56c6eca	<1m	68488	68488	78m	0	-	-	-
joinmo	a5ee885	<1m	1322370	169888	>5h	4	-	-	-
buildergenerator	d9d73b3	<1m	17533	17533	23m	0	-	-	-
mybatis-3	809c35d	6m	275791	89482	>5h	0	-	-	-
antlr4	9e7b131	3m	1860	1860	24m	1	-	-	-
hamcrest-bean	84586d9	<1m	678946	123335	>5h	3	62m	1	29143
raml-java-parser	49aab8f	<1m	23770	23770	81m	0	-	-	-

Table 8. Experimental results of the Genesis "CCE (TV)" search space on CCE Defects

size of the search space that the algorithm explores within the five hour timeout, the "Search Time" column presents the amount of time spent exploring the space, and "Validated Patches" presents the number of candidate patches that validate (produce correct outputs for all test cases). The last three columns present statistics for the first generated correct patch, specifically how long it takes to generate the patch ("Generation Time"), the rank of the first correct patch in the sequence of validated patches ("Validated Rank"), and the rank of the correct patch in the sequence of candidate patches ("Space Rank").

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
caelum-stella	2ec5459	2m	62433	62433	76m	25	<1m	1	147
caelum-stella	2d2dd9c	1m	33198	33198	47m	28	<1m	1	553
caelum-stella	e73113f	<1m	33592	33592	48m	28	<1m	1	528
HikariCP	ce4ff92	3m	163998	79985	>5h	33	-	-	-
nutz	80e85d0	2m	675603	245793	>5h	0	-	-	-
spring-data-rest	aa28aeb	7m	153943	18773	>5h	0	-	-	-
checkstyle	8381754	3m	592851	110058	>5h	29	12m	3	3261
checkstyle	536bc20	2m	839914	119964	>5h	49	<1m	1	26
checkstyle	aaf606e	2m	681420	117819	>5h	0	-	-	-
checkstyle	aa829d4	1m	0	0	<1m	0	-	-	-
jongo	f46f658	1m	325561	41504	>5h	0	-	-	-
DataflowJavaSDK	c06125d	3m	86731	78301	>5h	1	10m	1	4653
webmagic	ff2f588	1m	184724	115693	>5h	0	-	-	-
javapoet	70b38e5	<1m	280469	136400	>5h	0	-	-	-
closure-compiler	9828574	4m	$> 10^6$	31940	>5h	5	16m	1	14
truth	99b314e	<1m	84076	84076	56m	0	-	-	-
error-prone	3709338	2m	665832	3350	>5h	9	2m	1	473
javaslang	faf9ac2	<1m	$> 10^6$	392392	>5h	18	2m	2	12242
Activiti	3d624a5	4m	462310	2142	>5h	62	3m	4	31
spring-hateoas	48749e7	<1m	25633	25633	38m	43	<1m	1	268

Table 9. Experimental results of the Genesis composite search space on NPE Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
Bukkit	a91c4c6	<1m	430352	118319	>5h	4	2m	1	728
RoaringBitmap	29c6d59	4m	537740	70293	>5h	0	-	-	-
commons-lang	52b46e7	2m	136402	8347	>5h	0	-	-	-
HdrHistogram	db18018	<1m	344483	134113	>5h	140	-	-	-
spring-hateoas	29b4334	<1m	37105	37105	41m	0	-	-	-
wicket	b708e2b	7m	233586	102339	>5h	29	160m	12	46506
coveralls-maven-plugin	20490f6	<1m	7298	7298	6m	0	-	-	-
named-regexp	82bdfeb	<1m	0	0	<1m	0	-	-	-
jgit	929862f	3m	140077	140077	193m	3	84m	1	54732
jPOS	df400ac	3m	222560	222560	299m	17	26m	1	18022
httpcore	dd00a9e	2m	300612	20984	>5h	166	1m	1	427
vectorz	2291d0d	<1m	184636	184636	268m	32	<1m	1	2
maven-shared	77937e1	2m	0	0	<1m	0	-	-	-

Table 10. Experimental results of the Genesis composite search space on OOB Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
jade4j	dd47397	<1m	239966	120217	>5h	1	169m	1	65325
jade4j	114e886	<1m	437323	87192	>5h	0	-	-	-
HdrHistogram	030aac1	<1m	49997	49997	166m	74	2m	7	2662
pdfbox	93c0b69	1m	208714	142352	>5h	0	-	-	-
tree-root	fef0f36	<1m	43785	43785	39m	0	-	-	-
spoon	48d3126	9m	0	0	<1m	0	-	-	-
pebble	942aa6e	2m	255076	103554	>5h	0	-	-	-
fastjson	c886874	1m	$> 10^6$	258758	>5h	0	-	-	-
htmlelements	bf3f275	1m	164348	144123	>5h	12	36m	7	16270
spring-cloud-connectors	56c6eca	1m	151568	151568	244m	5	-	-	-
joinmo	a5ee885	1m	689426	158816	>5h	4	-	-	-
buildergenerator	d9d73b3	<1m	58520	58520	84m	0	-	-	-
mybatis-3	809c35d	8m	1184696	136625	>5h	0	-	-	-
antlr4	9e7b131	3m	5917	5917	48m	1	-	-	-
hamcrest-bean	84586d9	<1m	1081802	148401	>5h	4	30m	2	12029
raml-java-parser	49aab8f	<1m	87054	87054	162m	0	-	-	-

Table 11. Experimental results of the Genesis composite search space on CCE Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
caelum-stella	2ec5459	1m	876	876	2m	4	<1m	1	5
caelum-stella	2d2dd9c	<1m	606	606	3m	9	<1m	1	18
caelum-stella	e73113f	<1m	614	614	3m	9	<1m	1	16
HikariCP	ce4ff92	3m	2021	2021	12m	1	-	-	-
nutz	80e85d0	2m	11937	11937	48m	9	-	-	-
spring-data-rest	aa28aeb	7m	1349	1349	59m	0	-	-	-
checkstyle	8381754	3m	8126	8126	33m	5	<1m	1	41
checkstyle	536bc20	2m	8551	8551	38m	10	<1m	1	6
checkstyle	aaf606e	2m	7862	7862	29m	0	-	-	-
checkstyle	aa829d4	<1m	0	0	<1m	0	-	-	-
jongo	f46f658	<1m	6395	6395	86m	3	-	-	-
DataflowJavaSDK	c06125d	3m	1519	1519	7m	0	-	-	-
webmagic	ff2f588	1m	4624	4624	15m	0	-	-	-
javapoet	70b38e5	<1m	3343	3343	17m	0	-	-	-
closure-compiler	9828574	3m	3809	3809	25m	2	1m	1	8
truth	99b314e	<1m	1128	1128	1m	0	-	-	-
error-prone	3709338	2m	15905	117	>5h	0	-	-	-
javaslang	faf9ac2	<1m	45225	45225	47m	0	-	-	-
Activiti	3d624a5	4m	6113	6113	286m	92	-	-	-
spring-hateoas	48749e7	<1m	357	357	1m	6	<1m	1	6

Table 12. Experimental results of PAR templates on NPE Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
Bukkit	a91c4c6	<1m	543	543	1m	2	<1m	1	9
RoaringBitmap	29c6d59	4m	2054	2054	24m	0	-	-	-
commons-lang	52b46e7	2m	1460	1460	3m	0	-	-	-
HdrHistogram	db18018	<1m	853	853	2m	0	-	-	-
spring-hateoas	29b4334	<1m	640	640	<1m	0	-	-	-
wicket	b708e2b	8m	2917	2917	18m	3	3m	1	550
coveralls-maven-plugin	20490f6	<1m	266	266	<1m	0	-	-	-
named-regexp	82bdfeb	<1m	0	0	<1m	0	-	-	-
jgit	929862f	3m	1234	1234	4m	0	-	-	-
jPOS	df400ac	3m	3171	3171	6m	0	-	-	-
httpcore	dd00a9e	2m	1588	1588	9m	4	2m	1	15
vectorz	2291d0d	<1m	2314	2314	5m	2	<1m	1	107
maven-shared	77937e1	2m	0	0	<1m	0	-	-	-

Table 13. Experimental results of PAR templates on OOB Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
jade4j	dd47397	<1m	5371	5371	25m	0	-	-	-
jade4j	114e886	<1m	5313	5313	27m	0	-	-	-
HdrHistogram	030aac1	<1m	936	936	3m	7	-	-	-
pdfbox	93c0b69	1m	1375	1375	5m	0	-	-	-
tree-root	fef0f36	<1m	1722	1722	1m	0	-	-	-
spoon	48d3126	9m	0	0	<1m	0	-	-	-
pebble	942aa6e	2m	5104	5104	178m	0	-	-	-
fastjson	c886874	1m	35079	35079	131m	0	-	-	-
htmlelements	bf3f275	1m	3609	3609	9m	0	-	-	-
spring-cloud-connectors	56c6eca	<1m	3258	3258	5m	0	-	-	-
joinmo	a5ee885	<1m	15045	15045	33m	0	-	-	-
buildergenerator	d9d73b3	<1m	750	750	2m	0	-	-	-
mybatis-3	809c35d	7m	28248	28248	119m	0	-	-	-
antlr4	9e7b131	3m	446	446	2m	0	-	-	-
hamcrest-bean	84586d9	<1m	25912	25912	69m	0	-	-	-
raml-java-parser	49aab8f	<1m	1003	1003	3m	0	-	-	-

Table 14. Experimental results of PAR templates on CCE Defects