Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2016-010 July 8, 2016

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

Fan Long, Peter Amidon, and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

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 [15, 20–23, 27, 35, 42, 45, 46] 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 [20–23, 35, 42, 45, 46]. 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 a 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 trade-off between coverage and tractability [24]. 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) [24].

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 two classes of errors: null pointer errors (NPE) and out of bounds errors (OOB). The NPE patch training set includes 483 patches from 126 different applications; the OOB patch training set includes 199 patches from 117 different applications. For our benchmark set of 20 null pointer errors and 13 out of bounds errors, Genesis generates correct patches for 12 null pointer errors and 6 out of bounds errors. The time required for Genesis to generate the first correct patch for a given error is often less than a minute. All of the applications are large, real-world applications from github [4] or the MUSE corpus [7] with up to 235K lines of code. These results highlight the effectiveness of the Genesis

inference algorithms in finding productive patch search spaces for errors that occur in practice.

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 infer a search space to generate a correct patch for a real world null-pointer exception (NPE) error (shown at the bottom of Figure 1).

Collect and Split Training Set: Genesis works with a training set of successful human NPE patches to infer a search space for repairing NPE errors. In our example, the training set consists of 483 human patches for NPE errors collected from 126 github repositories. To avoid overfitting, Genesis reserves 121 (25%) human patches from the training set as a validation set (Section 3.4). This leaves 362 human patches remaining in the training set.

Generalizing Patches: The Genesis inference algorithm works with selected subsets of patches from the training 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 selected 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 [6] revision 6d7a4d, modelmapper [8] revision d85131, and Bukkit [2] revision f13115. Given these three patches, the Genesis generalization algorithm produces the transform in Figure 1. When applied, this transform can generate all of the three patches in the selected subset as well as other patches for other applications.

Each transform has a 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 $A\ \ op_1\ \texttt{(}B\ \ op_2\ \texttt{null)}$, 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 is either a call expression or a variable, 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. These constraints generalize the original three patches to appropriately scope the space of patches that the transform generates.

Applying the Transform: Figure 1 shows how Genesis applies the transform to patch a null pointer exception in another application, specifically in DataflowJavaSDK [3] revision c06125. 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 uses fault localization techniques (Section 4) to select this if condition as a patch candidate.

The transform also generates 13 other candidate patches at this if condition. Genesis uses the DataflowJavaSDK JUnit [5] 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 *plausible 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. In the example, the patch in Figure 1 is the only plausible patch. This plausible patch is also correct and matches the subsequent human developer patch for this exception.

Other Transforms: Figure 2 summarizes three other null pointer exception transforms that Genesis infers (out of a total of 14 inferred null pointer transforms). The first transform patches method call expressions on null objects. The transform 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 the transform 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.

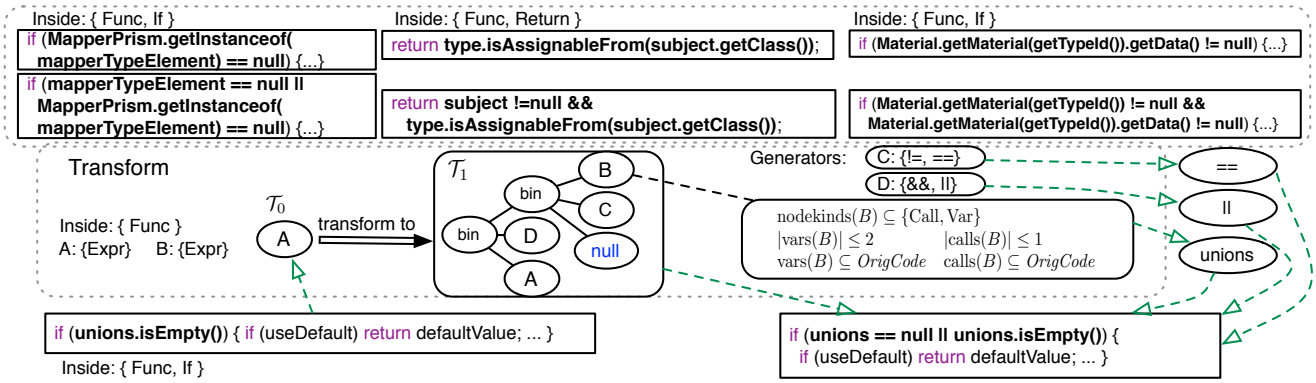

Figure 1. Example inference and application of a Genesis transform. The training patches (original and patched code) are at the top, the inferred transform is in the middle, and the new patch that Genesis generates is at the bottom.

The second transform in Figure 2 adds an if statement that executes an original statement A only if B `!= null`. This transform eliminates null pointer exceptions by simply skipping statements when they would otherwise throw null pointer exceptions. The third transform executes a return or continue statement C instead of an original statement A if B `== null`. This transform 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. All of these transforms capture null pointer exception patch patterns that appear in our training set of human null pointer exception patches.

Search Space: The Genesis candidate patch search space is determined by the set of inferred transforms that Genesis selects to generate candidate patches (operating in tandem with the error localization algorithm, see Section 4). To obtain an effective search space, Genesis must navigate a tradeoff between coverage (how many correct patches it can generate) and tractability (how long it takes to generate the search space, how long it takes to run the patched versions against the test suite, and how many plausible but incorrect patches it generates [24]). 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.

Genesis includes a search space selection algorithm that uses integer linear programming to navigate this tradeoff (Section 3.5). The search space is determined by a collection of selected transforms generalized from subsets of training patches. The integer linear program selects a collection of transforms that 1) maximizes the number of validation patches in the collected validation set that are inside the search space subject to 2) the number of candidate patches that all selected transforms generate when applied to the original code for any of such validation patches is less than a chosen bound. The linear program therefore maximizes coverage while excluding transforms that generate unacceptably large search spaces.

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, then use the integer linear program to select the search space. Because we work with too many training patches for this approach to scale, Genesis first uses a sampling algorithm that incrementally builds up promising subsets (starting with subsets of size two, then working up to subsets of size six). It uses a fitness function to prune unpromising

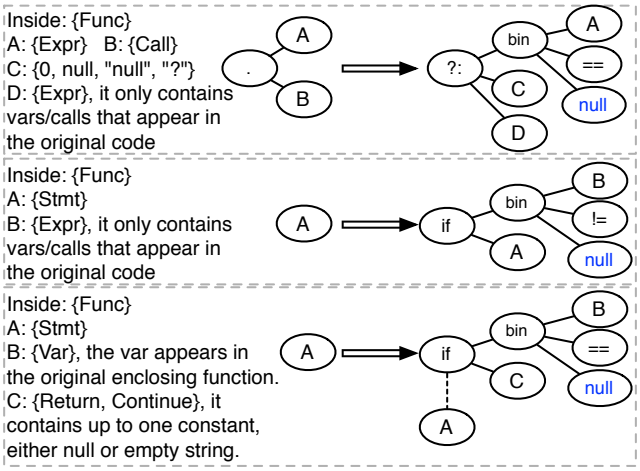

Figure 2. Three additional inferred transforms.

subsets (Section 3.4). Once it has obtained a collection of promising subsets of training patches, it applies the integer linear program to obtain the final selected transforms.

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 [16] 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 Java implementation details 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* $a \rightarrow b_1 b_2 b_3 \dots b_k$ *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*

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

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 special 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,* $L = \langle x_1, x_2, \dots, x_k\rangle$ *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* $L = \langle x_{c_i}, \dots, x_{c_j}\rangle$ *is a list of AST sibling nodes in* T *such that* $\exists x' \in X : \xi(x') = \langle x_{c_1}, \dots, x_{c_i}, \dots, x_{c_j}, \dots, x_{c_k}\rangle$ *(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 $D = \{\langle S_1, S_1'\rangle, \langle S_2, S_2'\rangle, \dots, \langle S_m, S_m'\rangle\}$ 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.

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

$\mathsf{nonterm}(L, X, \xi, \sigma, N)$ 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:

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

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

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

$\mathrm{diff}(A, B)$ 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 3 presents the detailed definitions of $\mathrm{diff}()$ and "$\equiv$".

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

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

Figure 3. Definition 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,* $\gamma : V \rightarrow \{0, 1\} \times \mathrm{Powerset}(N)$ *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 4 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 4 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 4 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 5 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,*

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

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

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

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

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

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

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

Figure 4. Definition of the operators "$\models$" and "$\models_{\texttt{slice}}$" for the template AST forest $\mathcal{T}$

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

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

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

Figure 5. Definition of the operator "$\Longrightarrow$" for the Generator $\mathcal{G} = \langle G, b, \delta, W \rangle$

$\langle \mathcal{G}, S \rangle \Longrightarrow T'$ denotes that the generator $\mathcal{G}$ generates the AST forest T'.

The first rule in Figure 5 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 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.

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

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

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

Figure 6. Definition of the operators "$\Longrightarrow$" and "$\Longrightarrow_{\texttt{slice}}$" for the transform $\mathcal{P}$

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 6 presents the formal definition of the "$\Longrightarrow$" and "$\Longrightarrow_{\texttt{slice}}$" 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'. "$\langle \mathcal{P}, S\rangle \Longrightarrow_{\texttt{slice}} S'$" denotes that applying $\mathcal{P}$ to the AST slice S generates the AST of the slice S'.*

Intuitively, in Figure 6 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 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 7 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 7, $\mathbb{A}$ 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., $b = 1$).

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

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

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 7. Definition of the generator inference operator ψ

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

Figure 8. Definition of the generalization function Ψ

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 8 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 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 8 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 9 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 9 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

$\mathbb{S} = \langle S_1, S_2, \ldots, S_m \rangle$ $G = (N, \Sigma, R, s)$ 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 9. Definition of Ψ'

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 tress. 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.

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.

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  W ← {{⟨S, S'⟩, ⟨S'', S'''⟩} | ⟨S, S'⟩ ∈ D, ⟨S'', S'''⟩ ∈
   D, ⟨S, S'⟩ ≠ ⟨S'', S'''⟩}
2  for i = 1 to 5 do
3      f ← {S ↦ fitness(W, S, D, E) | S ∈ W}
4      W' ← {S | S ∈ W, f(S) > 0}
5      Sort elements in W' based on f
6      Select top α elements in W' with largest f value as a new set
       W''
7      W ← W''
8      if i ≠ 5 then
9          for S in W'' do
10             for ⟨S, S'⟩ in D do
11                 W ← W ∪ {S ∪ ⟨S, S'⟩}
12 P' ← ∪_{S∈W} Ψ(S)
13 return P'
```

Figure 10. Sampling algorithm $\mathrm{sample}(D, E)$

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

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

Figure 11. Pseudo-code of $\mathrm{fitness}(\mathbb{W}, \mathbb{S}, D, E)$

to systematically obtain a set of productive candidate transforms for the inference algorithm to consider.

Figure 10 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 10 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 $\alpha = 500$ in our experiments), 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).

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 11 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 other transforms should receive much less 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 less scores than the pairs in validation set to avoid overfitting.

There are three optimizations in the Genesis implementation for the above sampling algorithm. Firstly, Genesis filters many unproductive subsets that may yield intractable search space when introducing new training in $\mathbb{W}$ at lines 10-11. 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, the computation of the search space size of the transform at line 5 and line 14 in Figure 11. 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 11 at the start of each iteration of the sampling algorithm to avoid redundant computation during each invocation of the $\mathrm{fitness}()$ function.

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

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

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 12 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.

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 $\mathbb{P}' \leftarrow \text{sample}(D, E)$
3 Solve ILP in Figure 12 to obtain $\mathbb{P}$
4 **return** $\mathbb{P}$

Figure 13. Search space inference algorithm

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 $\zeta = \infty$.

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.

Inference Algorithm: Figure 13 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 [16], an off-the-shelf solver, to obtain the set of transforms $\mathbb{P}$ that forms the result search space.

3.6 Java Implementation

We have implemented the Genesis inference algorithm for Java programs. We use the spoon library [32] to parse Java programs to obtain Java ASTs. We next discuss several extensions 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,

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 $\mathbb{S} \leftarrow \text{localization}(p, V)$
2 $G \leftarrow$ emptylist
3 **for** S **in** $\mathbb{S}$ **do**
4 **for** $\mathcal{P}$ **in** $\mathbb{P}$ **do**
5 **for** p' **in** $\{\mathsf{str}(T) \mid \langle \mathcal{P}, S \rangle \Longrightarrow T\}$ **do**
6 **if** $\text{validate}(p', V)$ **then**
7 Append p' to G
8 **return** G

Figure 14. Patch generation algorithm

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 semantical 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.

4. Patch Generation

Figure 14 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 [1] and JUnit [5] testing framework.

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.

5. Experimental Results

We next present experimental results of Genesis.

5.1 Methodology

Collect NPE and OOB Patches and Errors: We developed a script that crawled the top 1000 github Java projects (ranked by number of stars) to collect 503 null pointer error patches from 126 different applications. The script also crawled the top 1000 github Java projects and a list of 50968 github repositories from the MUSE corpus [7] to collect 212 out of bounds error patches from 117 different applications. The script that crawls the repositories and collects a project revision if 1) the project uses the apache maven management system [1], 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 [32] to parse the source code of both of the two revisions into AST trees, 4) the commit message of the current revision contains certain keywords to indicate that the revision corresponds to a patch for NPE errors or OOB 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 or OOB 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”. We manually inspected the retrieved revisions to discard revisions that do not correspond to actual NPE or OOB 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 3.3 or they have special dependencies that cannot be automatically resolved by the maven system.

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 [5] 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 20 NPE errors and 13 OOB errors from the total 503 NPE and 212 OOB errors, respectively. These are the *benchmark errors*.

Partition into Training, Validation, and Benchmark Patches: We partition the collected 503 NPE and 212 OOB patches into training, validation, and benchmark patches as follows. We first removed the 20 NPE and 13 OOB benchmark errors, leaving 483 and 199 remaining patches. We partitioned these remaining patches into 362 NPE and 149 OOB training patches and 121 NPE and 50 OOB validation patches (the inference algorithm uses these patches to avoid overfitting, see Section 3.5).

Search Space Inference: We run the Genesis search space inference algorithm on the NPE training and validation patches to infer the NPE transforms and on the OOB training and validation patches to infer the OOB transforms. For the OOB errors, we use the alternative ILP formulation in Section 3.5 because the number of validation OOB patches is small. 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.

Patch Generation for NPE and OOB Errors: We then run the Genesis patch generation system with the inferred NPE search space on the 20 testing NPE errors. We also run Genesis with the inferred OOB search space on the 13 testing OOB errors. We run the patch generation process 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 Genesis if it does not finish the exploration of the search space in five hours.

For each of the benchmark NPE and OOB errors, we manually analyze the root cause of the error, the corresponding developer patch in the repository, and all Genesis generated validated patches that pass the test cases. For each validated patch, we identify whether the patch is correct patch or not (i.e., the patch correctly fixes the error for all possible inputs).

Note that the corresponding developer patches for several NPE and OOB errors throws new exceptions with text error messages if certain conditions are true. Genesis validated patches do not attempt to generate the text error messages but instead leave empty string placeholders for the messages. In our experiments, we count such a Genesis patch correct if the patch semantically differs with the corresponding developer patch only in such text error messages.

	NPE	OOB
Transforms after Sampling	985	461
Covered Training Patches after Sampling	220 of 362	85 of 149
Covered Validation Patches after Sampling	58 of 121	14 of 50
Final Inferred Transforms	14	23
Covered Training Patches in Search Space	130 of 362	71 of 149
Covered Validation Patches in Search Space	51 of 121	10 of 50
Total Inference Time	199m	657m

Table 1. Transform and search space inference results

5.2 Inference Results

Table 1 presents search space inference results. The Genesis sampling algorithm (Section 3.4) produces 985 NPE transforms and 461 OOB transforms. These transforms cover 220 of the NPE training patches, 58 of the NPE validation patches, 85 of the OOB training patches, and 14 of the OOB validation patches. The Genesis search space inference (Section 3.5) algorithm selects 14 of the 985 NPE transforms and 23 of the 461 OOB transforms. These selected transforms cover 130 of the NPE training patches, 51 of the NPE validation patches, 71 of the OOB training patches, and 10 of the OOB validation patches. The search space inference times are reasonable at 199 minutes for NPE inference and 657 minutes for OOB inference. The time is dominated by the sampling algorithm. Solving the ILPs takes less one minute for both NPE and OOB transforms. We attribute the larger OOB inference time to the fact that the inferred OOB transforms have more sophisticated generators that tend to generate larger, more complex search spaces.

5.3 Patch Generation Results

Table 2 presents the Genesis patch generation results for the benchmark 20 NPE and 13 OOB errors. There is a line in the table for each benchmark error. The Init. Time column presents the amount of time required to initialize the search for that error. The Search

Repository	Revision	Type	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	Correct Patches	First Correct Patch		
									Generation Time	Validated Rank	Space Rank
caelum-stella	2ec5459	NPE	<1m	2682	2682	6m	4	1	<1m	1	12
caelum-stella	2d2dd9c	NPE	<1m	1287	1287	8m	18	18	<1m	1	5
caelum-stella	e73113f	NPE	<1m	1320	1320	8m	18	18	<1m	1	3
HikariCP	ce4ff92	NPE	3m	7262	7262	74m	26	2	47m	12	5265
nutz	80e85d0	NPE	1m	21266	21266	120m	14	0	-	-	-
spring-data-rest	aa28aeb	NPE	5m	5687	5576	>5h	8	6	38m	2	785
checkstyle	8381754	NPE	2m	23261	23261	227m	4	4	<1m	1	134
checkstyle	536bc20	NPE	2m	25984	25984	237m	8	8	<1m	1	4
checkstyle	aaf606e	NPE	2m	25752	25752	225m	0	0	-	-	-
checkstyle	aa829d4	NPE	<1m	0	0	<1m	0	0	-	-	-
jongo	f46f658	NPE	<1m	10893	10893	175m	3	0	-	-	-
Dataflow JavaSDK	c06125d	NPE	3m	4039	4039	43m	1	1	2m	1	124
webmagic	ff2f588	NPE	<1m	8398	8398	45m	0	0	-	-	-
javapoet	70b38e5	NPE	<1m	8107	8107	47m	0	0	-	-	-
closure-compiler	9828574	NPE	3m	116853	58520	>5h	4	4	6m	1	1039
truth	99b314e	NPE	<1m	1328	1328	3m	0	0	-	-	-
error-prone	3709338	NPE	2m	40238	40238	294m	5	1	73m	4	8854
javaslang	faf9ac2	NPE	<1m	54224	54224	63m	5	1	<1m	3	122
Activiti	3d624a5	NPE	2m	20057	1632	>5h	108	2	7m	4	137
spring-hateoas	48749e7	NPE	<1m	1064	1064	3m	6	6	<1m	1	13
Bukkit	a91c4c6	OOB	<1m	657814	657814	181m	7	5	<1m	1	21
RoaringBitmap	29c6d59	OOB	4m	407858	45604	>5h	0	0	-	-	-
commons-lang	52b46e7	OOB	1m	49561	3038	>5h	0	0	-	-	-
HdrHistogram	db18018	OOB	<1m	97613	97613	243m	71	0	-	-	-
spring-hateoas	29b4334	OOB	<1m	16506	16506	17m	0	0	-	-	-
wicket	b708e2b	OOB	5m	93738	93738	274m	22	6	79m	7	21695
coveralls-maven-plugin	20490f6	OOB	<1m	1567	1567	3m	0	0	-	-	-
named-regexp	82bdfeb	OOB	<1m	0	0	<1m	0	0	-	-	-
jgit	929862f	OOB	2m	60921	60921	121m	6	6	55m	1	27517
jPOS	df400ac	OOB	2m	88610	88610	176m	9	2	8m	1	5842
httpcore	dd00a9e	OOB	2m	123402	70176	>5h	104	9	148m	60	16455
vectorz	2291d0d	OOB	<1m	108273	108273	181m	25	9	60m	11	37749
maven-shared	77937e1	OOB	2m	0	0	<1m	0	0	-	-	-

Table 2. Experimental results for automatic patch generation.

Space Size column presents the size of the Genesis search space for that error, the Explored Space Size column presents the size of the search space that the algorithm explores within the five hour timeout (for most errors Genesis is able to explore the full search space), Validated Patches presents the number of candidate patches that validate (produce correct outputs for all test cases), and Correct Patches presents the number of validated patches that are correct. We note that some errors have multiple correct patches. These are different patches that are semantically equivalent in the context in which they appear.

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 validate patches, and the rank of the correct patch in the sequence of candidate patches. The generated NPE patches used a total of 10 of the 14 inferred NPE transforms; the generated OOB patches used a total of 11 of the 23 inferred OOB transforms.

To isolate the effect of error localization, we also run Genesis with an oracle that identifies, for each error, the correct line of code to patch. For NPE errors the oracle results are close to unchanged (for HikariCP the Validated Rank changes from 12 to 1; for javaslang the Validate Rank changes from 3 to 5), which highlights the effectiveness of our error localization algorithm for this class of errors. The Validate Rank for javaslang is slightly worse with oracle error localization because of traversal order variations in the underlying set and map data structures in the patch generation algorithm.

For OOB errors oracle error localization enables Genesis to generate 2 more correct patches, improves the Validation Rank of two patches (httpcore and vectorz) from 60 and 10 to 1 and changes the Validation Rank of wicket from 7 to 8 (again because of traversal order variations in the underlying set and map data structures). The remaining unpatched errors all fall outside the inferred search spaces. We believe that an enhanced training set would enable Genesis to learn transforms that bring most if not all of these errors within the Genesis search space.

NPE Patch Categories: All of the generated correct patches are available in the supplemental material. Broadly speaking, the generated correct NPE patches fall within several categories. Some patches introduce a null pointer check, then if the check succeeds, either 1) return a synthesized value or void, 2) throw a new synthesized exception, or 3) skip a statement with a null pointer error. Other patches modify an existing boolean expression, either by conjoining or disjoining a synthesized expression or by replacing a clause in the expression with a pointer equality check.

OOB Patch Categories: The generated correct OOB patches also fall within several categories. Some patches introduce a comparison that checks for an out of bounds access followed by a return or break statement if the comparison succeeds. Other patches modify an existing boolean expression, typically by conjoining or disjoining a synthesized expression or by changing a binary comparison operator (such as changing < to <=).

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 [22] and SPR [23] 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 [21, 46], AE [45], and RSRepair [35] 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 [36] applies a single transform that simply deletes code. All of these systems were evaluated on the same benchmark set [21]. 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 [23, 36]. Like Prophet, history-driven program repair [20] uses information from previous human patches to rank candidate patches generated by human-specified transforms.

PAR [18] 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 [22], SPR [23], Qlose [11], NOPOL [15], SemFix [31], and Angelix [27] all use constraint solving to generate new values for potentially faulty expressions (often faulty conditions). ClearView [34] enforces learned invariants to eliminate security vulnerabilities. Angelic Debugging [10] 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 [42]. 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 [12, 13, 17, 47] 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 expressions 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 [19, 33, 41]. 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 [22], code refactoring [38], and code completion [9, 37, 39]. 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 [26]. 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 [29] and Lase [30] 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 [28] uses Lase edit scripts to refactor code clones. FixMeUp [44] 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 [43], 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 [40] discards out of bounds writes and manufactures values for out of bounds reads. RCV [25] returns zero as the result of null pointer dereferences and divide by zero errors. APPEND [14] 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 run-time 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. Results from our implemented Genesis system highlight its ability to infer productive search spaces and deploy those search spaces to effectively patch null pointer and out of bounds errors in real-world Java programs.

References

[1] Apache maven. `https://maven.apache.org/`.

[2] Bukkit. `https://bukkit.org`.

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

[4] GitHub. `https://github.com/`.

[5] Junit. `http://junit.org/`.

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

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

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

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

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.