Direct Manipulation for Imperative Programs

Qinheping Hu[1], Roopsha Samanta[2], Rishabh Singh[3], and Loris D'Antoni[1(✉)]

[1] University of Wisconsin-Madison, Madison, USA
loris@cs.wisc.edu
[2] Purdue University, West Lafayette, USA
[3] Google, Hanover, USA

Abstract. Direct manipulation is a programming paradigm in which the programmer conveys the intended program behavior by modifying program values at runtime. The programming environment then finds a modification of the original program that yields the manipulated values. In this paper, we propose the first framework for direct manipulation of imperative programs. First, we introduce *direct state manipulation*, which allows programmers to visualize the trace of a buggy program on an input, and modify variable values at a location. Second, we propose a synthesis technique based on program sketching and quantitative objectives to efficiently find the "closest" program to the original one that is consistent with the manipulated values. We formalize the problem and build a tool JDial based on the Sketch synthesizer. We investigate the effectiveness of direct manipulation by using JDial to fix benchmarks from introductory programming assignments. In our evaluation, we observe that direct state manipulations are an effective specification mechanism: even when provided with a single state manipulation, JDial can produce desired program modifications for 66% of our benchmarks while techniques based only on test cases *always* fail.

1 Introduction

Direct manipulation [1–4] is a programming paradigm in which the programmer conveys the intended program behavior by modifying program values at runtime. The programming environment then finds a modification of the original program that yields the manipulated values. This paradigm has been successfully applied to drawing editors [5,6] to provide programming capabilities that allow users to interact directly with the displayed graphics.

In this paper, we propose the first framework for direct manipulation of imperative programs. We start by introducing *direct state manipulation*, a specification mechanism in which users can describe the intended program behavior by directly manipulating intermediate variable values in buggy program traces. We propose a workflow in which the user traverses the step-by-step visualization of the execution of the buggy program on a certain input to identify a location where the values of the program variables do not correspond to the ones

B.-Y. E. Chang (Ed.): SAS 2019, LNCS 11822, pp. 347–367, 2019.
https://doi.org/10.1007/978-3-030-32304-2_17

she expects. At this point, we allow the user to *manipulate* the variable values at the identified location and modify them. We then treat this manipulation as a specification and use it to synthesize a program that, on the same input, can reach the location identified by the user with the new variable values she provided.

We formalize our synthesis problem and present a constraint-based synthesis technique for computing programs consistent with direct state manipulations. Solving this problem requires addressing two key challenges. First, the execution step manipulated by the user in the buggy trace might appear at a different point in the trace of the synthesized program—e.g., when the modified program uses more/fewer loop iterations than the original one. Second, since a single program execution under-specifies the overall program behavior, there can be many possible programs that agree with the manipulated trace. To address the first challenge, given a manipulated location ℓ, we design an encoding that "guesses" in what occurrence of the location ℓ in the trace of the synthesized program the desired variable values are produced. To address the second challenge, we augment our synthesis problem with quantitative objectives [7] to prefer programs that produce execution traces similar to those of the original program—i.e., the goal is to compute a modified program that on the input provided by the user produces an execution trace similar to the one in the original program.

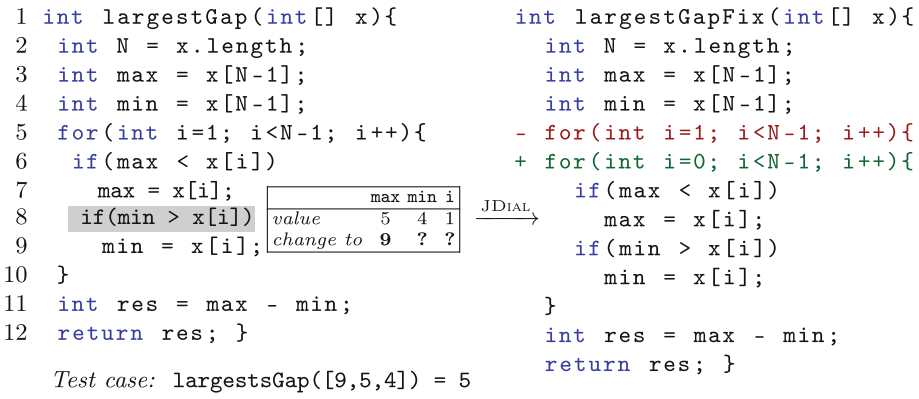
We implemented our synthesis technique in a tool called JDial, which is built on top of the Sketch [8] synthesizer. JDial supports several program transformation models—i.e., descriptions of how the program can be modified—and program distances, and can handle Java programs containing loops, arrays, and recursion. To handle programs containing library functions such as `Math.pow`, JDial introduces a synthesis algorithm that uses concrete program executions to "discover" partial interpretations of external functions and uses such interpretations to synthesize modifications to the whole program. For the common case in which producing a new program only requires modification of a single statement, JDial uses a data flow analysis based on program slicing to summarize and reduce parts of the program for which the corresponding traces will not be affected by the code modification.

We evaluate JDial on a set of representative program repair benchmarks. We observe that direct state manipulations are an effective specification mechanism: even when provided with a single manipulation, JDial can produce desired program modifications for 66% of our benchmarks while techniques based only on test cases *always* fail and produce undesirable programs.

Contributions. We make the following key contributions.

- We introduce the specification mechanism of *direct state manipulation* and a corresponding synthesis problem in which the goal is to find a program that produces the variable values specified by the user at a certain point in the program execution trace and that has minimal distance from the original program according to some metric (Sect. 3).
- We propose a framework based on program sketching for synthesizing programs using direct state manipulations (Sect. 4).

– We instantiate our framework in JDial, a tool that supports direct state manipulations for simple Java programs (Sect. 5).
– We evaluate JDial on 17 representative benchmarks and show JDial computes good program modifications in cases where specifications based on test cases produce undesirable ones (Sect. 6).



a) Direct state manipulation on failing test b) Program computed by JDial

Fig. 1. Examples of synthesis using direct manipulation in JDial.

2 Illustrative Example

In this section, we illustrate our direct manipulation framework using an example student attempt to an introductory programming exercise. In this domain, automatic program repair—i.e., finding program transformations that fix the program—has been used to provide personalized feedback to students [9–11]. We show how direct state manipulations can be used an alternative to test-cases for program repair in this domain.

Consider the example in Fig. 1(a) where a student is trying to write a program `largestGap` for finding the *largest gap* in a non-empty array of integers—i.e., the difference between the maximum and minimum values in the array. In the following, we assume that the student has discovered that the program behavior on test `[9,5,4]` is incorrect and is trying to get a suggestion from the tool on how she could fix the program.

Specification via Test Cases. Several tools support test cases as a way to express the correct behaviour of the program. In this case, the student can specify that on the input `[9,5,4]`, the correct output should be 5. However, even the tool Qlose [7], which can often find correct program modifications using a small number of test cases, will return the following wrong modification to line 11:

```
int res = max - min; ⟶ int res = max - min + 4;
```

For this example, QLOSE requires two additional carefully selected test cases to find the correct program transformation.

Specification via Direct State Manipulation. Direct state manipulations allow programmers to convey *more information about the behavior of a test case*, rather than only its final output. Our proposal to use direct state manipulations in this domain is inspired by Guo's observation [12] that students find it beneficial to visualize concrete program executions and observe discrepancies between the variable values they observe and those they expect. For example, while debugging the `largestGap` program, the student notices that in the first iteration of the loop, right before executing line 8, variable `max` has value 5 instead of the expected value 9. While visualizing the trace, the student can directly modify the value of `max` as shown in the figure and JDIAL will synthesize the program `largestGapFix` consistent with the manipulation (Fig. 1(b))—i.e., when running `largestGapFix` with input `[9,5,4]`, there is a point in the execution where the variable `max` contains value 9 right before executing line 8. Why does this new specification mechanism lead to the desired program? First, by modifying the program's trace and its value at line 8, the student implicitly states that certain lines do not need modification—e.g., lines 11 and 12. Second, the modification provides information about an intermediate state of the program that a tool cannot access through just an input/output example. Besides the variable `max`, the student can modify the value of `i` from 1 to 0 or the values of both `i` and `max` at the same position and JDIAL will produce the same program.

Remarkably, direct state manipulation can also help debug partial implementations. Consider, for example, an incomplete version of the program `largestGap` in which lines 8–9 are missing because the student has not implemented the logic for min yet. The test case in Fig. 1(a) is essentially useless. On the other hand, the same direct manipulation shown in Fig. 1(a) will yield a good program.

3 Problem Definition

In this section, we define the class of programs we consider, the notion of direct state manipulation, and our synthesis problem.

3.1 Programs and Traces

We consider a simple imperative language in which a program P consists of a function definition $f(i_1, \ldots, i_q) : o$ with input variables $I = \{i_1, \ldots, i_q\}$ and output variable o (`NULL` for void functions), a set of program variables V such that $V \cap I = \emptyset$, and a sequence of labeled statements $\sigma = s_1 \ldots s_n$. A statement is one of the following: `return`, assignment, conditional or loop statement. Each statement in σ is labeled with a unique location identifier from the set $L = \{\ell_0, \ell_1, \ldots, \ell_p, exit\}$. We assume a universe $\mathcal{U}$ of values. We also assume variables are associated with types and assignments are consistent with these types.

Without loss of generality, we assume that executing a `return` statement assigns a value to the output variable and transfers control to a designated

location *exit*. A program configuration η is a pair (ℓ, ν) where $\ell \in L$ is a location and $\nu : I \cup \{o\} \cup V \mapsto \mathcal{U} \cup \{\bot\}$ is a valuation function that assigns values to all variables. The element $\bot$ indicates that a variable has not been assigned a value yet or is out of scope. We write $(\ell, \nu) \rightarrow (\ell', \nu')$ if executing the statement at location ℓ under variable valuation ν transfers control to location ℓ' with variable valuation ν'. The execution trace $\pi_P(\nu_0)$ of the program P on an initial valuation ν_0 is a sequence of configurations $\eta_0, \eta_1, \ldots$, where $\eta_0 = (\ell_0, \nu_0)$ and for each h, we have $\eta_h \rightarrow \eta_{h+1}$. An execution terminates once the location *exit* is reached and we only consider programs that terminate on all inputs. We use $\pi_P(\nu_0)_l = \eta_l$ to denote the configuration at index l and $\pi_P(\nu_0)_{[l,h]}$ to denote the subsequence of configurations between index l and h—e.g., $\pi_P(\nu_0)_{[3,5]} = \eta_3\eta_4\eta_5$.

Consider the program `largestGap` in Fig. 1. The input variable set I is {`x`} and the output variable is `res`. The set of program variables is {`i,max,min`}. Let ν_0 be the initial valuation such that $\nu_0(x) = [9, 5, 4]$ and $\nu_0(w) = \bot$ for every other variable w. The execution of `largestGap` on ν_0 is shown in Fig. 2.

	η_0	η_1	η_2	η_3	η_4	η_5	η_6	η_7	η_8	η_9	η_{10}
loc	2	3	4	5	6	7	8	5	10	11	*exit*
`N`	$\bot$	3	3	3	3	3	3	3	3	3	3
`i`	$\bot$	$\bot$	$\bot$	$\bot$	1	1	1	1	$\bot$	$\bot$	$\bot$
`max`	$\bot$	$\bot$	4	4	4	4	5	5	5	5	5
`min`	$\bot$	$\bot$	$\bot$	4	4	4	4	4	4	4	4
`res`	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	1

Fig. 2. Execution of `largestGap` on ν_0. We omit valuations of the input variable x as $\nu_h(x) = \nu_0(x)$ for all h.

3.2 Synthesis for Direct State Manipulation

We define the notion of direct state manipulation, which allows users to express their intent by modifying variable values in intermediate configurations. We assume a fixed program P. A *direct state manipulation* $\mathcal{M}$ is a tuple (ν_0, k, ν') where ν_0 is an initial valuation, k is an index s.t. $k \leq |\pi_P(\nu_0)|$, and $\nu' : V \cup \{o\} \mapsto \mathcal{U} \cup \{\textbf{?}\}$ is a new partial variable valuation. Intuitively, the manipulation replaces the configuration $\pi_P(\nu_0)_k = (\ell, \nu)$ at location ℓ with the new partial configuration (ℓ, ν'). Notice that a partial configuration cannot change the values of the variables in I and it can assign a special value **?** to certain variables. This value is used to denote that the manipulation "does not care" about the specific values of certain variables. We say that a valuation ν *satisfies* a partial valuation ν', denoted $\nu \vdash \nu'$, iff for every variable $x \in V \cup \{o\}$, if $\nu(x) \neq \textbf{?}$ then $\nu(x) = \nu'(x)$.

Example 1. The direct state manipulation in Fig. 1(a) is formally defined as the pair $(\nu_0, 6, \nu')$ where ν_0 is the same as at the end of Sect. 3.1, $\nu'(max) = 9$ and $\nu'(i) = \nu'(min) = \nu'(o) = \textbf{?}$. This manipulation, which modifies η_6, only sets the value of `max` to 9 at location 8 and leaves all other variables unconstrained.

Given a program P, a direct state manipulation $\mathcal{M} = (\nu_0, k, \nu')$ such that $\pi_P(\nu_0)_k = (\ell, \nu)$, we say that a program P' *satisfies the manipulation* $\mathcal{M}$, if there exists an index j such that $\pi_{P'}(\nu_0)_j = (\ell, \nu_j)$ and $\nu_j \vdash \nu'$—i.e., a program P' satisfies a direct state manipulation if there exists some configuration in the execution trace of P' satisfying the manipulated valuation ν' at location ℓ.

The synthesis problem is to find a program that satisfies a given manipulation. In what follows, we fix a *transformation model*, which is a function $\mathcal{RM}$ that assigns to a program a corresponding *synthesis space* $\mathcal{P}$. The synthesis space represents a set of programs from which we can draw candidate programs.

Definition 1 (Synthesis for Direct State Manipulation). *Given a program P and a direct state manipulation $\mathcal{M} = (\nu_0, k, \nu')$, the* synthesis for direct state manipulation *problem is to find a program $P' \in \mathcal{RM}(P)$ that satisfies the manipulation $\mathcal{M}$.*

Informally, a direct state manipulation (ν_0, k, ν') at location ℓ is a *reachability specification* requiring that a configuration (ℓ, ν') is *eventually* reached along an execution from the initial valuation ν_0. This specification mechanism is orthogonal to *assertions*, which require a property φ at location ℓ to be an *invariant*—i.e., each time an execution reaches location ℓ, the property φ holds. For instance, in Fig. 1(a), placing the assertion `max = 9` at location 8 would specify that the value of `max` should be 9 at location 8 across all loop iterations in an execution. The astute reader may suggest that for some suitably chosen predicate *condition* over the loop counter, an assertion of the form *condition* $\Rightarrow$ `(max = 9)` at location 8 could encode the direct state manipulation in Fig. 1. However, a direct state manipulation does not explicitly indicate what such a predicate *condition* should be. In particular, a direct state manipulation does not specify what the manipulation-satisfying index j should be.

Handling Test Cases. Definition 1 can be generalized to the problem of synthesising a program P given a direct state manipulation and a set of tests. A test t is a pair (ν^I, ν^O) where ν^I and ν^O are valuations over the input variables I and the output variable o, respectively. Let ν_0^I denote an initial valuation such that $\nu_0^I(w) = \nu^I(w)$ if $w \in I$ and $\perp$ otherwise. Program P satisfies a test t if the value of the output variable o at the end of an execution $\pi_P(\nu_0^I)$ of P on valuation ν_0^I is ν^O—i.e., if $j = |\pi_P(\nu_0^I)| - 1$, $\eta_j = (\ell, \nu)$ and $\nu(o) = \nu^O$. Program P satisfies a set of tests T if it satisfies all the tests $t \in T$. The synthesis problem is then to find a program that satisfies both the direct state manipulation and the tests.

Cost-aware Synthesis. Among the many programs that satisfy a given state manipulation we would like to pick the "best" one. To define what it means for a program to be better than another one, we use the notions of program distances proposed in [7]. We define two types of distances: syntactic and semantic distances. Given a program P, a syntactic distance is a function $f_{syn}^P : \mathcal{P} \to \mathbb{N}$ that maps each program in the synthesis space to a quantity capturing its syntactic similarity to the original program P. We define semantic distances using distance functions over execution traces. Let $dist(\pi, \pi')$ denote a distance function mapping a pair of traces to a non-negative integer. Intuitively, *dist* captures

the similarity between execution traces of P and P' on the same initial valuation ν_0. Given a program P and a direct state manipulation $\mathcal{M} = (\nu_0, k, \nu')$, a semantic distance function $f_{sem}^{P,\mathcal{M}} : \mathcal{P} \to \mathbb{N}$ maps a synthesized program P' to $dist(\pi_P(\nu_0)_{[0,k]}, \pi_{P'}(\nu_0)_{[0,j]})$ capturing the similarity between the manipulated trace $\pi_P(\nu_0)_{[0,k]}$ of P and the corresponding *manipulation-satisfying trace* $\pi_{P'}(\nu_0)_{[0,j]}$ of P' with manipulation-satisfying index j. An aggregation function $\textsc{Aggr} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is used to combine the two distance functions.

Example 2. An example of syntactic distance between two programs P and P' is the number of node edits needed to transform the abstract syntax tree P into the one P'. According to this distance, the change from `i=1` to `i=0` showed in Fig. 1 has syntactic distance 1. An example semantic distance is the sum of the differences in variable valuations in program configurations of the execution traces $\pi_P(\nu_0)_{[0,k]}$ and $\pi_{P'}(\nu_0)_{[0,j]}$ (with j as defined above).

For a program P and direct state manipulation $\mathcal{M}$, we can define the *cost* of a synthesized program P' as $cost(P') = \textsc{Aggr}(f_{syn}^{P}(P'), f_{sem}^{P,\mathcal{M}}(P'))$. The following definition can be generalized to incorporate a set of tests.

Definition 2 (Cost-aware Synthesis for Direct State Manipulation). *Given a program P and a direct state manipulation $\mathcal{M}$, the* Cost-aware synthesis for direct state manipulation *problem is to find a program $P' \in \mathcal{RM}(P)$ that satisfies the manipulation $\mathcal{M}$ and such that, for every $P'' \in \mathcal{RM}(P)$ that satisfies the manipulation $\mathcal{M}$, we have $cost(P') \leq cost(P'')$.*

4 JDial's Architecture

In this section, we describe the architecture of JDial and the sketching-based approach JDial employs to synthesize programs (Fig. 3).

JDial takes as input a buggy program, a direct state manipulation on an input trace, and (optionally) a set of test cases (left of Fig. 3). As described in Sect. 3, the synthesis problem is defined using four components: a transformation model, a syntactic distance function, a semantic distance function, and a cost-aggregation function. In JDial, these components are modular and defined independently from the underlying synthesis engine (grey boxes at the top of Fig. 3). The transformation model is given as a program GetSynthesisSpace that, given a program, returns a sketched version of it—i.e., a program with unknown holes of the form `??`. In Fig. 3, this program simply replaces each constant with a hole. By instantiating the holes in the sketched program with concrete values we obtain a program in the synthesis space. The syntactic distance is given as a program that computes a non-negative integer based on the values of the holes in the sketched program—e.g., how many constants were changed or by how much they were changed. The semantic distance is given as a program that computes a non-negative integer based on the value of two traces—e.g., the Hamming distance.

4.1 Synthesis via Sketching

To solve the synthesis problem, JDial computes a sketched program together with a set of assertions (blue box in Fig. 3). The solution to this sketched program—i.e., values for the holes that satisfy the assertions and minimize the given objective function—is the solution to our synthesis problem.

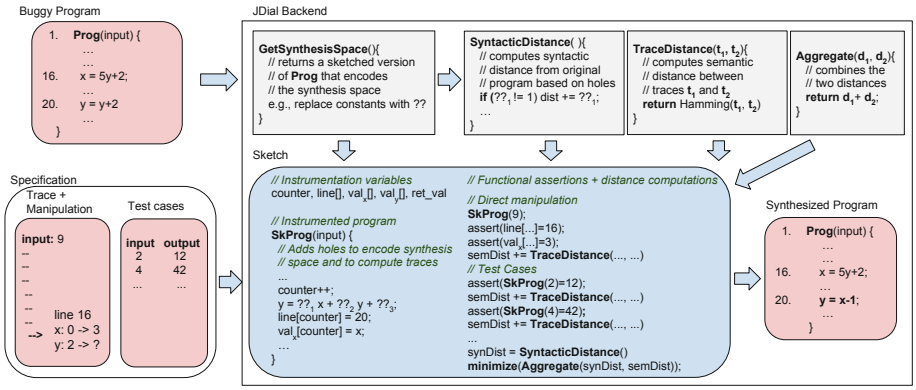


Fig. 3. Architecture of JDial. Grey components can be modified without having to modify the synthesis algorithm. (Color figure online)

Background on Sketching. Program sketching is a technique for specifying a parametric set of programs. This is done by allowing programs to contain holes (denoted by `??`). When one provides a specification—e.g., test cases, assertions, minimization objectives—the sketching problem is to find (typically integer) values of the holes that satisfy the given specification. State-of-the-art sketching tools support complex program constructs, such as arrays, strings, and recursive functions, as well as complex specification mechanisms, such as Boolean assertions and quantitative optimization constraints over the values of the holes [8].

Computing Distances and Guessing Trace Lengths. The GetSynthesisSpace component, given the buggy program, adds holes to generate a sketched program encoding the synthesis space—e.g., `y = ??_1 * x + ??_2 * y + ??_3` in Fig. 3 (blue-box). JDial then generates a function that uses the values placed in the holes to compute the syntactic distance (Fig. 3(top)).

To compute the semantic distance JDial symbolically extracts traces by instrumenting the sketched program with a counter to measure the length of the trace, an array to record the values of each variable in the original program, and an array for the line numbers.[1] After each instruction, the arrays are updated to

[1] We assume that the length of the trace in the synthesized program is at most twice the length of the original trace and we use this assumption to initialize the length of the arrays. This constant is parametric and can be modified.

reflect the current variable values (Fig. 3(blue-box)). JDial then computes the semantic distance using the traces extracted from such arrays.

The key difficulty in encoding our synthesis problem is that there can be many ways to "align" the location manipulated by the user with a location in the sketched program—e.g., the execution of one synthesized program might reach the desired manipulated value the second time the manipulated location is visited, while another candidate program might reach the desired value the tenth time the manipulated location is visited. JDial must be able to consider all these possibilities.

Example 3. Consider the manipulation described in Fig. 1(a). The execution of the synthesized program `largestGapFix` presented in Fig. 1(b) on input $[9, 5, 4]$ hits the manipulated location with `max=9` the first time line 8 is traversed. However, another correct program which changes the loop in line 5 to `i=N-2; i>=0; i--`, hits the manipulated location with `max=9` the second time line 8 is traversed.

Our key idea is to introduce an existential variable—i.e., a hole—in our sketched program to guess at what visit time the manipulated line is reached with the variable values provided by the user. Concretely, we define a global variable `int visit_time=??` to guess the number of visits of the manipulated line and modify the sketched program right before the sketched version of the manipulated line to interrupt the trace at the correct time (see Fig. 4). Thus, every time the manipulated line is reached, `visit_time` is decremented and, when the counter hits zero, the execution has reached the guessed number of visit times.

```
...
if(visit_time==0){
   // record state
   return; }
else
   visit_time--;
//Sketch of
//manipulated line
...
```

Fig. 4. Instrumentation to guess visiting times.

Finally, JDial adds assertions to guarantee the sketch solution satisfies the manipulation and a minimization objective to ensure the returned solution is optimal with respect to the given distances (see right of blue box in Fig. 3).

4.2 Correctness of the Synthesis Procedure

We now state the correctness of our encoding. Given a program P and a direct state manipulation $\mathcal{M}$, we call $\textsc{sket}(P, \mathcal{M})$ the sketched program computed by JDial. Recall the definition of cost-aware synthesis for direct state manipulation in Definition 2. Theorem 1 states that JDial correctly encodes the problem of cost-aware synthesis for direct state manipulation. We say an algorithm for this problem is *sound* if it only produces solutions in $\mathcal{RM}(P)$ that have minimal cost and satisfy $\mathcal{M}$, and *complete* if it produces a solution whenever one exists. Moreover, a program sketching solver is sound and complete if it can correctly solve all program sketches.

Theorem 1. JDial *is sound and complete for the problem of cost-aware synthesis for direct state manipulation iff the program sketching solver it uses is sound and complete.*

5 Implementation and Optimizations

JDial is composed of a frontend, which allows to visualize program traces and manipulate intermediate states, and a backend, which synthesizes the transformed programs. JDial can handle Java programs over integers, characters, Booleans, and arrays over these basic types. In its default mode, JDial only tries to modify statements in the function in which the manipulated line appears. In this section, we describe the concrete transformation model and distance functions JDial uses as well as several optimizations employed by JDial.

$$
\begin{array}{rcl}
\textsc{r}(\texttt{e}_1\ \texttt{bop}\ \texttt{e}_2) & \rightarrow & \textsc{r}(\texttt{e}_1)\ \texttt{bop}\ \textsc{r}(\texttt{e}_2)\ \texttt{+}\ (\sum_{\texttt{v}\in V} ??_b\texttt{v+??}) \\
\textsc{r}(\texttt{x = e}) & \rightarrow & \texttt{x} = \textsc{r}(\texttt{e})\ \texttt{+}\ (\sum_{\texttt{v}\in V} ??_b\texttt{v+??}) \\
\textsc{r}(\texttt{return e}) & \rightarrow & \texttt{return}\ \textsc{r}(\texttt{e})\ \texttt{+}\ (\sum_{\texttt{v}\in V} ??_b\texttt{v+??}) \\
\textsc{r}(\texttt{c*e}) & \rightarrow & \texttt{c *}\ (\textsc{r}(\texttt{e}) + (\sum_{\texttt{v}\in V} ??_b\texttt{v}+??)) \\
\textsc{r}(\texttt{x[e]}) & \rightarrow & ??_b\texttt{x[}??_b\texttt{*e+??]}
\end{array}
\qquad
\begin{array}{rcl}
\textsc{r}(\texttt{c}) & \rightarrow & \texttt{c} \\
\textsc{r}(\texttt{x}) & \rightarrow & ??_b\texttt{x} \\
\textsc{r}(\texttt{e}_1\texttt{+e}_2) & \rightarrow & \textsc{r}(\texttt{e}_1) + \textsc{r}(\texttt{e}_2) \\
\textsc{r}(\texttt{f(e)}) & \rightarrow & ??_b\texttt{f(e)}
\end{array}
$$

Fig. 5. JDial's transformation model.

5.1 Transformation Model and Syntactic Distance

JDial supports complex transformation models—e.g., it can allow statements to be added to the program. However, overly expressive transformation models will often lead to undesired programs that overfit to the given manipulation. In fact, existing tools for automatically fixing introductory programming assignments typically employ several transformation models, each tailored to a particular programming assignment [9,13].

Transformation Model. Since in our application domain we do not know a priori what program the programmer is trying to write, JDial's default transformation model only allows to rewrite constants in linear arithmetic expressions. Figure 5 illustrates JDial's default transformation model and Fig. 6 illustrates an example of how the transformation model generates a Sketch from a program.

First, any variable in any expression is multiplied by a hole $??_b$ that only takes values from the set $\{-1, 0, 1\}$. These holes can be used to remove variables and negate their coefficients. Second, the term $\sum_{\texttt{v}\in V} ??_b\texttt{v+??}$, where V is the set of variables, is added to each expression appearing in an assignment or in a Boolean comparison. These terms can be used to add new variables, further increase/decrease the coefficients of variables appearing in the expression, or add new constants—e.g., turn `x<0` into `x>y`. This transformation model permits modifications of multiple expressions and it subsumes the default error model of the AutoGrader tool, which, despite its simplicity, was shown to be able to automatically fix 30%–60% of edX student submissions depending on the problem type [9].

Syntactic Distance. JDial's syntactic distance computes the difference between the synthesized hole values and the original ones. For example, in the expression

$??_b$`x < 0 +` $\sum_{v \in V}$ $??_b$`v+??` (corresponding to original expression `x<0`), the original value of the first hole $??_b$ is 1, while the original value of all the other holes is 0. The syntactic distance is the sum of the absolute difference between each hole's synthesized value and the original one. Intuitively, this distance penalizes modifications that introduce new variables and modify constants by large amounts.

5.2 Semantic Distance over Traces

When computing the distance from the original program traces, JDial ignores the variables that have been manipulated because they are likely to contain "incorrect" values that are not necessary to preserve. JDial first computes the restricted traces where the values of the manipulated variables are omitted and then uses a modified version of the Hamming distance to compute their distance. In the following definitions, we assume Boolean tests return 1 when true and 0 when false. Given two configurations $\eta = (l, \nu)$ and $\eta' = (l', \nu')$ over a set of variables V, the distance between the two configurations is defined as $H(\eta, \eta') = (l \neq l') + \sum_{w \in V} \nu(w) \neq \nu'(w)$. Finally, JDial computes the distance between two traces $\pi = \eta_1 \cdots \eta_s$ and $\pi' = \eta'_1 \cdots \eta'_t$, where $m = min(s, t)$ and $M = max(s, t)$ as the quantity $H(\eta_1, \eta'_1) + \cdots + H(\eta_m, \eta'_m) + M - m$.

```
int triple(int x){                int SkTriple(int x){
  int y = 3 * x;                    int y=3*(??b x+(??b x+??))+(??b *x+??);
  if(x == 10)          R            if(??b x == 10 + (??b x+??b y+??))
     y = 30;          ⟶              y = 30 + (??b x+??b y+??);
  return y; }                       return ??b y + (??b x+??b y+??); }
```

Fig. 6. A sketched program obtained from applying the transformation model to a program. Holes of the form `??` can be instantiated with arbitrary integers. Holes of the form $\sum_{v \in V}$ $??_b$`v+??` can only be instantiated with values in $\{-1, 0, 1\}$.

Example 4. Consider again the example described in Fig. 1. The restricted trace of the synthesized program up to the manipulation-satisfying index has distance 3 from the original trace since it only changes the value of the variable `i` in the last three steps and it has the same length as the original program trace.

JDial contains other implementations of trace distances—e.g., longest common subsequences. Since the distance presented above yields good results and performance in practice, we use it as default and in our experiments. JDial aggregates the syntactic and semantic distances by taking their sum.

5.3 Handling External Functions

JDial employs a new Counterexample-Guided Inductive Synthesis (CeGIS) scheme to handle programs that contain external functions for which semantics

might be unknown or expensive to encode directly in SKETCH. Given the input program with an external function `ext` and the manipulated trace, JDIAL creates a sketched program that assigns a partial interpretation to the external function using the set of concrete values obtained from the input trace execution—i.e., for every call of the function observed in the input trace. JDIAL then computes a solution for the sketched program using this partial definition of `ext`. If synthesizing the program requires knowing the interpretation of `ext` on inputs that have not been observed yet, JDIAL lets SKETCH "guess" an interpretation for the function `ext` on such inputs. JDIAL can then execute the function `ext` and check whether the guesses were correct. If they are not correct, JDIAL modifies the new sketched program to incorporate the partial interpretation to the external function `ext` on the newly discovered inputs. The process continues until JDIAL finds a program that respects the semantics of `ext`.

```
// Test case
// sumPow(3)=15
int sumPow(int x){
  int sum = 1;
  for(int i=1;i<x;i++){
    sum+=Math.pow(2,i);
  }
  return sum;
}
```

(a)

```
// Partial interpretation
// of Mathpow
int Mathpow(a, b){
  if(a==2 && b==1)
    return 2;
  if(a==2 && b==2)
    return 4;
  if(a==?? && b==??)
    return ??; }
```

(b)

1. JDIAL guesses `Mathpow(2,3)=14`
2. Is `Math.pow(2,3)=14`?
3. No, modify `Mathpow`

(c)

```
// Refined interpretation
// of Mathpow
int Mathpow(a, b){
  if(a==2 && b==1)
    return 2;
  if(a==2 && b==2)
    return 4;
  if(a==2 && b==3)
    return 8;
  if(a==?? && b==??)
    return ??; }
```

(d)

Fig. 7. Given an example with an incorrect for-condition and an input test (a), JDIAL uses the execution of `sumPow` on the test to learn an initial partial interpretation of the function `Math.pow` (b). JDIAL then produces a proposed program and guesses the interpretation of `Math.pow` to be such that `Math.pow(2,3)=14` (c). After executes `Math.pow` in Java, JDIAL discovers that `Math.pow(2,3)=8`, and refines the interpretation of `Math.pow` for the next round of synthesis (d).

Example 5. Consider the program `sumPow` in Fig. 7(a), which should compute the sum of powers of 2 up to `x`, but instead it only computes the sum up to `x-1`. By running the program on the given input 3, JDIAL can obtain the output of the

`Math.pow` function on input values 1 and 2, and constructs a SKETCH function that describes a partial interpretation of `Math.pow` as shown in Fig. 7(b). To synthesize the program, JDIAL needs to change the condition of the for loop, but this transformation requires knowing the output of the function `Math.pow` on arguments (2,3) and our partial interpretation of `Math.pow` does not contain this information. JDIAL synthesizes a transformation for the function `sumPow` and, while doing so, it assigns an interpretation to the inputs for which the behaviour of the function `Math.pow` is unknown (Fig. 7(c)). JDIAL then uses the concrete execution of the function `Math.pow` to check whether the synthesized interpretation is incorrect, and in this case it modifies the partial interpretation of `Math.pow` in the sketched program.

```
1 // input: [3,2,7]
2 int[] subLargestGap(int[] a){
3   int N=a.length;
4   int min=max=a[0];
5   for(int i=0; i<N; i++){
6     if(max<a[i]) max=a[i];
7     if(min>a[i]) min=a[i];}
8   int largestgap = max-min;
9  R:for(int i=1; i<N; i++){
10    a[i]=a[i]-largestgap;
11 M:}
12  return a; }
```

```
1 void sliced(int[] a){
2   int N=3;
3   int largestgap=5;
4   for(int i=1; i<N; i++)
5     a[i]=a[i]-largestgap;
6   return a; }
```

Fig. 8. Program `subLargestGap` and its `sliced` version when the manipulation happens at line 11 and only line 9 can be modified.

5.4 Additional Features and Optimizations

Specified Transformation Range. Since the programmer might want to prevent JDIAL from modifying certain program statements, JDIAL's frontend allows the programmer to specify what statements the tool is allowed to modify.

Single Statement Transformations. Since most synthesized programs only require transforming a single statement, JDIAL supports this restricted transformation model and it uses an optimized solver that, for each line of code, builds a separate sketched program that is only allowed to modify that line. The separate sketched programs are solved in parallel and JDIAL outputs the program of least cost. For each sketch that can only modify a certain line of code, JDIAL uses program slicing [14] to summarize parts of the program that will not be affected by the line modification. Concretely, let $\ell_{\mathcal{M}}$ be the location at which the manipulation is performed and ℓ_R be the location JDIAL is allowed to modify. By computing a *backward* slice of the manipulated location $\ell_{\mathcal{M}}$, we obtain the statements that can affect the values of the manipulated variables. Similarly, only statements

that are reachable from location ℓ_R in the control-flow graph of the program are affected by modifications to line ℓ_R. Finally, the intersection of the two sets gives us the statements where variable values may vary as a result of a transformation. All other statements are irrelevant and can be removed or summarized.

Example 6. Consider the program `subLargestGap` in Fig. 8 that returns a new array obtained by subtracting the largest gap of the input array from all its elements. This program contains a mistake in the second for loop. Assume a student is trying to fix it by manipulating the variable `a[0]` at location 11 on input `[3,2,7]` and that the transformation model only allows modification to location 9. The backward slice of location 11 contains all the statements in the program except the return statement and the lines 9 to 12 are the only lines reachable from location 9. Using this information, JDial summarizes all other statements' values. For example, the whole computation of the variable `largestGap` is replaced by the constant assignment `largestGap=5`.

Table 1. Effectiveness and performance of JDial. ✗ denotes out of memory.

	Problem	**LOC**	**Vars**	**|Trace|**	**Time** [sec]	**Time single line** [sec]	
						JDial$_1$	JDial$_1^\circ$
Qlose [7]	`largestGap-1.1`	7	4	11	3.8	1.6	1.0
	`largestGap-1.2`	7	4	10	2.2	0.8	0.6
	`largestGap-2`	7	4	15	4.2	1.1	0.5
	`largestGap-3.1`	7	4	10	1.8	1.1	0.5
	`largestGap-3.2`	7	4	10	2.8	1.0	0.6
	`tcas`	10	4	7	0.8	0.4	0.4
	`max3`	5	3	3	0.5	0.3	0.3
	`iterPower-1`	5	3	14	0.4	0.6	0.4
	`iterPower-2`	5	3	14	0.7	0.4	0.3
	`ePoly-1`	6	4	12	4.6	3.7	1.3
	`ePoly-2`	6	4	12	2.5	1.7	0.9
	`multIA`	4	4	9	1.3	0.8	1.1
New	`ePoly-3`	7	4	13	2.9	2.8	2.5
	`max4`	7	4	4	0.3	0.2	0.3
	`bubbleSort`	7	5	12	3.1	1.3	0.6
	`subLargestGap`	13	6	35	✗	✗	0.7
	`maxMin`	13	6	37	✗	✗	0.9

6 Evaluation

We evaluate the effectiveness of JDial through the following questions.

Q1 Can JDial yield desirable programs more often than test-based techniques?
Q2 Is the optimized version of JDial presented in Sect. 5.4 effective?
Q3 How sensitive is JDial w.r.t. the location of the manipulation?
Q4 Can JDial handle programs that contain external functions?

We perform our evaluation on 17 Java programs: 12 from Qlose [7] and 5 new programs. All benchmarks and the corresponding Sketch files are available at this url: https://tinyurl.com/yd6bp3dx. The five variants of the `largestGap` problem presented in Sect. 2 are taken from the CodeHunt platform [15], The `tcas-semfix` program is a toy traffic collision avoidance system from [16]. The `max`, `iterPower`, `ePoly`, and `multIA` problems are taken from the Introduction to Python Programming course taught on edX [17]. Two of the new programs we consider are variations of Qlose benchmarks. The other three `bubbleSort`, `subLargestGap`, and `maxMin` are larger programs that contain multiple loops, which are more complex than the benchmarks considered in [7].

Table 1 shows detailed metrics for each benchmark and the average runtime of JDial when performing synthesis on five randomly generated failing inputs. All experiments were performed on an Intel Core i7 4.00 GHz CPU with 32GB/RAM.

6.1 Comparison to Test-Based Tools

We compare JDial against the tool Qlose to see if synthesis via direct manipulation can find meaningful programs more often than synthesis via test cases. We compare against Qlose because it is the only test-based tool that uses semantic distances and it produces "good" programs when using a small number of test cases more often than tools that only use syntactic distances [7].

For each benchmark, we randomly generate 5 input tests that result in incorrect outputs. For each failing test, we run Qlose using the test as a specification

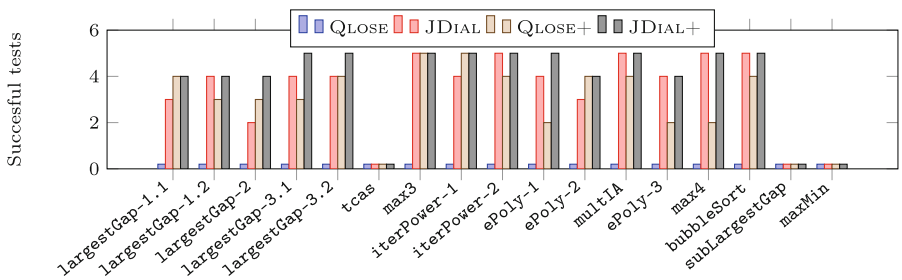


Fig. 9. Correct transformations out of 5 randomly generated tests for JDial vs Qlose. Additional test provided in JDial+ and Qlose+.

and run JDial by manually constructing a manipulation: we identify the first location in the execution trace where a variable has the wrong value and modify it to the correct one. Figure 9 illustrates the results of this comparison (JDial and Qlose bars). JDial generates the desired transformations in 66% (56/85) of the cases while Qlose never produces a correct program. When given only one test case, Qlose always modifies the return statement of the program.

We perform another study where, for each previous experiment, we provide JDial and Qlose with an additional (failing or passing) test—i.e., we provide Qlose with two tests and JDial with one test and one manipulation. Figure 9 illustrates the results of this comparison (cf. JDial+ and Qlose+ bars). JDial generates the intended transformations for 75% (64/85) of the cases while Qlose produces the intended transformations on 58% (49/85) of the cases. While Qlose performs better than when given a single test, for every input on which Qlose produces the correct transformation, JDial also does so. Remarkably, when given a single manipulation and nothing more, JDial produces correct transformations more often than Qlose, even when the latter is provided with 2 tests. To answer **Q1**, **JDial produces meaningful programs more often than techniques that only use tests**.

Before concluding, we explain why both tools performed poorly on some benchmarks. For the `tcas` program, the desired fix modifies an expression by adding a large constant that can only be synthesized from a very specific test case. Additionally, `subLargestGap` and `maxMin` benchmarks are too large. For the instances for which JDial produces the incorrect program, we evaluate whether JDial produces correct transformations if it is allowed further "attempts". Whenever an undesired transformation is generated at a location ℓ, we disallow JDial to transform location ℓ again or reject the proposed transformation and ask for a different one. This approach correctly synthesizes an additional 6 failing benchmarks with an average of 2.2 user interactions.

6.2 Optimizations for Single-Line Transformations

We repeat the previous experiment using the single-line transformation model described in Sect. 5.4. We refer to the version of JDial with this restricted transformation model as JDial_1 and its optimized version as JDial_1°. All our benchmarks can be fixed using a single-line transformation so both JDial_1 and JDial_1° find the same transformation. The last two columns of Table 1 show the running times. JDial_1 is generally faster than the version of JDial that uses the more complex transformation model. However, the optimized version JDial_1° is on average 1.37x faster than JDial_1. Moreover, for `subLargestGap` and `maxMin`, JDial_1° finds transformations in < 1 second while JDial_1 times out. The improvement is due to the slicing-based data-flow analysis, which, can reduce the number of lines in the sketched program from 25 to 8.

To answer **Q2**, **the optimization from** Sect. 5.4 **is beneficial** for single-line transformations. This transformation model is very practical and our results hint that our slicing technique can make JDial scale to larger programs.

6.3 Sensitivity of Manipulated Location

One of the key aspects of JDial is that the user has to find a "good" location to perform the desired transformation. In this experiment, we evaluate how sensitive JDial is with respect to the location choice. We consider the experiment we performed against test-based tools and for each test case on which JDial and JDial+ successfully found a correct transformation, we then perform the following analysis: if the manipulation was performed at step i in the program trace, we measure after how many steps the generated transformation is "lost"—i.e., we compute the smallest k for which performing the manipulation at position $i + k$ would yield a wrong transformation.

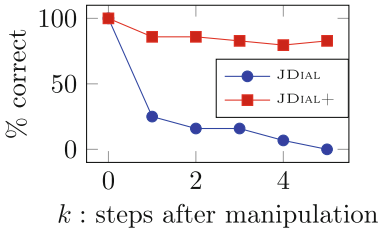


Fig. 10. Correct transformation if manipulating k steps after first point of error.

Figure 10 shows the results. In 80% of the cases, if JDial is provided only with a manipulation and the manipulation is performed one step later, JDial returns an incorrect transformation. However, when provided with one additional test case JDial returns the correct transformation in 80% of the cases, even when the manipulation is performed 5 steps after the ideal location. Even in these extreme conditions, JDial returns correct transformations more often than Qlose does when provided with two test cases. To answer **Q3**, **JDial is sensitive with respect to the manipulation location only if no additional tests are provided**, but it is still more precise than Qlose.

6.4 Ability to Handle External Functions

We evaluate if JDial can handle programs with external functions. `ePoly-1` and `ePoly-2`, contain the function `Math.pow` and JDial is able to produce a transformation for them using between 2 and 5 iterations (average 4.2), of the CeGIS algorithm presented in Sect. 5.3.

To better evaluate the algorithm, we design two more families of benchmarks. The first family of programs tries to compute $\sum_{i=0}^{n} \texttt{Math.pow}(2, i)$ for values of n between 2 and 8. The bug in this benchmark is the one shown in Fig. 7. For inputs 2 and 3, JDial can find the correct transformation that is compliant with the external function after 2 CeGIS iterations, while for inputs 4 through 8, JDial requires 3 iterations. The second family of programs computes the maximum value in an array using the `Math.max` function for different incorrect initializations of the variable `max`. In this case, the size of the initial constant affects the number of required CeGIS iterations. While incorrectly initializing `max` to 2 only requires a couple of iterations to produce the correct transformation, if we incorrectly initialize `max` to 100, computing the transformation requires guessing many new interpretations of the function `Math.max` that did not appear in the original trace, resulting in more than 90 iterations.

To answer **Q4**, **JDial can handle programs that contain external functions**, but in certain pathological cases it requires many CeGIS iterations.

7 Related Work

Direct Manipulation. Direct manipulation has been used in drawing editors [1–4]. The most relevant work in this space is Sketch-n-Sketch [5,6], which uses program synthesis to apply direct manipulation to scalable vector graphics (SVG)—i.e., constants in the program can be modified conforming to the direct manipulations. Sketch-n-Sketch and JDial tackle different domains. Unlike JDial, Sketch-n-Sketch can only rename constants defined at the top of the program and cannot handle complex updates involving changes in the program structure—e.g., replacing $x = y$ with $x = y - z$. Finally, Sketch-n-Sketch uses heuristics to select the "right" fix, while JDial does so using program distances.

In Wolverine [18], the user can modify a graphical abstract representation of a data structure such as a linked list and the tool will attempt to find a program modification consistent with the modification. Similar to Sketch-n-Sketch, Wolverine's technique is specific to certain families of data structure transformations and relies on the graphical abstraction used for the manipulation.

CodeHint [19] synthesizes simple Java expressions—e.g., library calls—at user-set breakpoints using partial specifications—e.g., variable types. It uses information from the execution to construct expressions of a user-provided type. CodeHint is different from JDial in two main aspects: (*i*) CodeHint helps programmers auto-complete function calls given some expected type at a given location, whereas JDial transforms the original program using a global analysis. (*ii*) CodeHint performs brute-force search while JDial uses constraint-based search with optimization objectives.

Personalized Education. There are many tools for teaching programming that help with grading (see [20] for a survey), personalized feedback [7,9,13,21,22], and visualization [12]. Several works have dealt with transforming synthesis tools into feedback generators [11]. Here, we discuss tools relevant to our work.

AutoGrader [9] and Qlose [7] repair incorrect student solutions to introductory programming assignments. These systems require a reference implementation or a comprehensive set of test cases while JDial also allows students to discover potential transformations using direct manipulations. JDial extends Qlose's technique to compute minimal program transformations. In particular, JDial encodes the problem of finding good stop points for aligning partial program traces, which is a new problem arising from our specification mechanism.

Program Repair. Program repair is the problem of automatically fixing bugs in large pieces of code. This topic has been studied extensively and researchers have proposed techniques based on constraint-solving [16,23], abstractions [24], and genetic algorithms [25]. These tools are mostly interested in fixing particular types of bugs—e.g., null-pointer exceptions. JDial uses constraint solving, but it would be interesting to investigate if other techniques work in our domain.

There are program repair approaches that find repaired programs that are syntactically close [23,26] or semantically close [27] to the original program. It was demonstrated in [7] that transformations generated using a combination of syntactic and semantic program distances are, in general, more *desirable* although more expensive to compute. Hence, JDial chooses this last approach and only compares against Qlose [7] since other tools rely on high-quality test suites.

Existing tools use test cases [7,16,28], logic specifications [29], or reference programs [9]. Direct manipulation "augments" a test case by allowing the user to specify intermediate information about the run of the program on a certain input. Moreover, direct manipulations can be used to debug incomplete implementations. Finally, it is important to note that direct manipulation is not directly expressible using assertions or test cases: while an assertion at a certain location is valid if *every* time the location is traversed the predicate in the assertion is true, a direct manipulation at a certain location only requires that *at some point* in the trace the variables evaluate to the manipulated values at that location.

Several tools use fault localization to find likely locations to modify [30–32]. The work on angelic debugging [33] is particularly relevant, where possible faulty expressions in a program are inferred by replacing them with an alternate concrete value (oracle) that makes all the tests pass. However, the burden on repairing the program with the correct expression still lies with the programmer.

The CeGIS refinement of external functions in Sect. 5.3 is related to the notion of Sketch models [34], which allow one to specify certain properties (such as associativity, idempotence, etc.) to provide richer interpretations to uninterpreted functions. In contrast, JDial iteratively builds a model of the auxiliary function directly in the synthesis process.

Acknowledgment. This work was supported by NSF under grants CNS-1763871, CCF-1704117 and CCF-1846327; and by the UW-Madison OVRGE with funding from WARF.

References

1. Victor, B.: Drawing dynamic visualizations (2013). http://worrydream.com/
2. Schachman, T.: Apparatus (2015). http://aprt.us/
3. Hottelier, T., Bodik, R., Ryokai, K.: Programming by manipulation for layout. In: UIST, pp. 231–241 (2014)
4. Shneiderman, B.: Direct manipulation: a step beyond programming languages. ACM SIGSOC Bullstin, vol. 13, no. 2–3, p. 143 (1982)
5. Chugh, R., Hempel, B., Spradlin, M., Albers, J.: Programmatic and direct manipulation, together at last. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 341–354. ACM (2016)
6. Hempel, B., Chugh, R.: Semi-automated SVG programming via direct manipulation. In: Proceedings of the 29th Annual Symposium on User Interface Software and Technology, pp. 379–390. ACM (2016)

7. D'Antoni, L., Samanta, R., Singh, R.: Qlose: program repair with quantitative objectives. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 383–401. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_21
8. Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transfer **15**(5–6), 475–495 (2013)
9. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: ACM SIGPLAN Notices, vol. 48, no. 6, pp. 15–26 (2013)
10. Yi, J., Ahmed, U.Z., Karkare, A., Tan, S.H., Roychoudhury, A.: A feasibility study of using automated program repair for introductory programming assignments. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2017, pp. 740–751. ACM (2017)
11. Suzuki, R., et al.: Tracediff: debugging unexpected codebehavior using synthesized code corrections. In: VL/HCC 2017 (2017)
12. Guo, P.J.: Online python tutor: embeddable web-based program visualization for CS education. In: Proceeding of the 44th ACM Technical Symposium on Computer Science Education, pp. 579–584. ACM (2013)
13. Rolim, R., et al.: Learning syntactic program transformations from examples. In: Proceedings of the 39th International Conference on Software Engineering, ser. ICSE 2017, pp. 404–415. IEEE Press, Piscataway (2017)
14. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449. IEEE Press 1981
15. Tillmann, N., De Halleux, J., Xie, T., Bishop, J.: Code hunt: gamifying teaching and learning of computer science at scale. In: Proceedings of the First ACM Conference on Learning@ Scale Conference, pp. 221–222. ACM (2014)
16. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 772–781. IEEE Press (2013)
17. edX: Introduction to computer science and programming using python (2017). https://www.edx.org/course/introduction-computer-science-mitx-6-00-1x-10
18. Verma, S., Roy, S.: Synergistic debug-repair of heap manipulations. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2017, pp. 163–173. ACM (2017)
19. Galenson, J., Reames, P., Bodik, R., Hartmann, B., Sen, K.: Codehint: dynamic and interactive synthesis of code snippets. In: Proceedings of the 36th International Conference on Software Engineering, pp. 653–663. ACM (2014)
20. Striewe, M., Goedicke, M.: A review of static analysis approaches for programming exercises. In: Kalz, M., Ras, E. (eds.) CAA 2014. CCIS, vol. 439, pp. 100–113. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08657-6_10
21. Gulwani, S., Radiček, I., Zuleger, F.: Automated clustering and program repair for introductory programming assignments. arXiv preprint arXiv:1603.03165 (2016)
22. Kim, D., et al.: Apex: automatic programming assignment error explanation. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ser. OOPSLA 2016, pp. 311–327. ACM (2016)
23. Mechtaev, S., Yi, J., Roychoudhury, A.: Directfix: Looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, pp. 448–458. IEEE Press (2015)
24. Logozzo, F., Ball, T.: Modular and verified automatic program repair. In: ACM SIGPLAN Notices, vol. 47, no. 10, pp. 133–146. ACM (2012)

25. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In: International Conference on Software Engineering (ICSE), pp. 3–13. IEEE Press (2012)
26. Samanta, R., Olivo, O., Emerson, E.A.: Cost-aware automatic program repair. In: Müller-Olm, M., Seidl, H. (eds.) SAS 2014. LNCS, vol. 8723, pp. 268–284. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10936-7_17
27. Von Essen, C., Jobstmann, B.: Program repair without regret. Formal Meth. Syst. Des. **47**(1), 26–50 (2015)
28. Le, X.-B.D., Chu, D.-H., Lo, D., Le Goues, C., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2017, pp. 593–604. ACM (2017)
29. Koukoutos, M., Kneuss, E., Kuncak, V.: An update on deductive synthesis and repair in the leon tool. arXiv preprint arXiv:1611.07625 (2016)
30. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: ACM SIGPLAN Notices, vol. 38, no. 1, pp. 97–105. ACM (2003)
31. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: ACM SIGPLAN Notices vol. 46. no. (6), pp. 437–446 (2011)
32. Könighofer, R., Bloem, R.: Automated error localization and correction for imperative programs. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 91–100. IEEE (2011)
33. Chandra, S., Torlak, E., Barman, S., Bodik, R.: Angelic debugging. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 121–130. IEEE (2011)
34. Singh, R., Singh, R., Xu, Z., Krosnick, R., Solar-Lezama, A.: Modular synthesis of sketches using models. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 395–414. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_22