Program Synthesis

SWE 795, Spring 2017

Software Engineering Environments

GEORGE MASON UNIVERSITY

Today

- HW3 is due next week in class!
- Part 1 (Lecture)(~50 mins)
- Break!
- Part 2 (Discussion)(~60 mins)
  - Discussion of readings
- Part 3 (In class activity)(~20 mins)
  - Project work

Main idea

- Developers describe a desired behavior, environment synthesizes code that provides this behavior

- Many applications have been explored
  - Intelligent macro recorders
  - Deobfuscation
  - Autocomplete
  - Bug fixing
  - New algorithm discovery

Generate & Validate approach

- Generate code
- Validate if code satisfies constraints
- If yes, stop

Characterizing generate & validate techniques

- Developer intent: how do developers describe the desired behavior?
- Search space: what programs can possibly be synthesized?
- Search technique: how does the technique enumerate candidate programs within the search space?

Sumit Gulwani. 2010. Dimensions in program synthesis. *Symposium on Principles and practice of declarative programming* (PPDP '10), 13-24.

Expressing developer intent with constraints

- Input / output examples
- Unit tests
- Logical relations between inputs and outputs (specifications)
- User demonstrations
- Keywords describing intent
- Partially complete programs with “holes”

- Key considerations
  - How specific are the constraints?
  - How long does it take to evaluate if candidate program satisfies constraint? (e.g, specification for expression vs. test suite for program)

Sumit Gulwani. 2010. Dimensions in program synthesis. *Symposium on Principles and practice of declarative programming* (PPDP ’10), 13-24.

Specifications: example

$x = 1\,0\,1\,0\,\mathbf{1}\,\mathbf{1}\,0\,0$
$\downarrow$
$y = 1\,0\,1\,0\,\mathbf{0}\,\mathbf{0}\,0\,0$
(a)
	$\exists i, j.\ \{\ 0 \le i, j < n$	`TurnOffRightMostOnes(`x`)`
		`  `i `:= 0;`
		`  while(`$x[i]$`==0` $\wedge$ $i < n$`)`
		`  `i `:= `i`+1;`
		`  while(`$x[i]$`==1` $\wedge$ $i < n$`)`
		`  `$x[i]$ `:= 0;` i `:= `i`+1;`
		`  return` x`;`

$y := x\&(1+(x\|(x-1)))$ (b)	$\wedge\ (\forall k.\ j \le k \le i \implies x[k] = 1)$ $\wedge\ (\forall k.\ 0 \le k < j \implies x[k] = 0)$ $\wedge\ (j \le i\ \vee\ j = n-1)$ $\wedge\ (\forall k.\ i < k < n \implies x[k] = y[k])$ $\wedge\ (\forall k.\ 0 \le k \le i \implies y[k] = 0)\ \}$ (c)	(d)

Figure 2. Consider the problem of masking off the rightmost contiguous sequence of 1's in a given bitvector. (a) describes an example input-output pair (x, y). (b) describes a 4-step program to solve the problem. (c) describes the intent using a logical relation between input bitvector x and output bitvector y, both of which are of size n. (d) describes the intent using an inefficient program.

Sumit Gulwani. 2010. Dimensions in program synthesis. *Symposium on Principles and practice of declarative programming* (PPDP '10), 13-24.

Search space

- Competing goals
  - Expressive: include all programs of interest
  - Restrictive: smaller search space

- Often expressed in terms of what language constructs are or are not allowed
- Examples
  - Expressions only with arithmetic operators
  - Expressions with function invocations & operators
  - Expressions, guarded by one of a specific set of conditionals
  - Loop-free programs with conditionals
  - Expressions with depth a maximum node depth of 4
  - Arbitrary programs

Sumit Gulwani. 2010. Dimensions in program synthesis. *Symposium on Principles and practice of declarative programming* (PPDP '10), 13-24.

Some methods of reducing search space

- Expressing programs in less expressive domain specific language
  - e.g,. method invocations & conditionals controlling when they exist; control
- Assembling code from existing code snippets
  - Plastic surgery hypothesis: high redundancy in code, so existing code snippets can often be found (and perhaps slightly adapted)

Search techniques

- Brute force
  - Enumerate all programs in the search space
- Version spaces
  - Maintain list of satisfying boolean functions
  - Order from most general to least general
  - Refine as more constraints are added
- Probabilistic inference
  - Estimate distribution elements in search space from data, use to bias search
  - e.g., toString() is far more frequent than xizo(100032)
- Genetic programming
  - Maintain population of programs, use selection, mutation, crossover to evolve
- SAT solvers
  - Represent constraints as logical formula, generate program that satisfies constraint

Sumit Gulwani. 2010. Dimensions in program synthesis. *Symposium on Principles and practice of declarative programming* (PPDP '10), 13-24.

Techniques we'll examine today

- Genetic programming
- Probabilistic inference
- Keyword constraints
- Execution trace constraints (programming by demonstration)
- Synthesizing transformations

Genetic programming

- One of the oldest approaches, based on genetic algorithms

- Uses analogy with biology
  - DNA —> programs
  - Keep population of programs
  - Select highest scoring programs (e.g., best satisfy constraints) for replication
  - Use crossover & mutation to evolve programs towards better solution

Defect Repair: GenProg

- 1. What is it doing wrong?
  - We take as input a set of negative test cases that characterizes a fault. The input program fails all negative test cases.
- 2. What is it supposed to do?
  - We take as input a set of positive test cases that encode functionality requirements. The input program passes all positive test cases.
- 3. Where should we change it?
  - We favor changing program locations visited when executing the negative test cases and avoid changing program locations visited when executing the positive test cases.
- 4. How should we change it?
  - We insert, delete, and swap program statements and control flow using existing program structure. We favor insertions based on the existing program structure.
- 5. When are we finished?
  - We call the first variant that passes all positive and negative test cases a primary repair. We minimize the differences between it and the original input program to produce a final repair.

Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. 2010. Automatic program repair with evolutionary computation. *Commun. ACM* 53, 5 (May 2010), 109-116.

Example

```
1  void zunebug(int days) {
2    int year = 1980;
3    while (days > 365) {
4      if (isLeapYear (year)){
5        if (days > 366) {
6          days -= 366;
7          year += 1;
8        }
9        else {
10         }
11     }
12     else {
13       days -= 365;
14       year += 1;
15     }
16   }
17   printf("the year is %d\n", year);
18 }
```

Example

```
 5  if (days > 366)   {
 6    days -= 366;
 7    if (days > 366) {   // insert #1
 8      days -= 366;      // insert #1
 9      year += 1;        // insert #1
10    }                   // insert #1
11    year += 1;
12  }
13  else {
14  }
15  days -= 366;          // insert #2
```

```
 5  if (days > 366)   {
 6    // days -= 366;          // delete
 7    // if (days > 366) {     // delete
 8    //   days -= 366;        // delete
 9    //   year += 1;          // delete
10    // }                     // delete
11    year += 1;
12  }
13  else {
14    days -= 366;          // insert
15  }
16  days -= 366;
```

Example

```
 1  void zunebug_repair (int days) {
 2    int year = 1980;
 3    while (days > 365) {
 4      if (isLeapYear (year)) {
 5        if (days > 366) {
 6          // days -= 366;  // deleted
 7          year += 1;
 8        }
 9        else {
10        }
11          days -= 366;        // inserted
12      } else {
13        days -= 365;
14        year += 1;
15      }
16    }
17    printf ("the year is %dn", year);
18  }
```

Promising results?

- GenProg: fixed 55 of 105 considered bugs
- RSRepair: 24 of 105 GenProg bugs
- AE: 54 of 105 considered bugs

- The test suite is a set of novice programming mistakes, likely to contain more obviously atypical erroneous code that is perhaps easier to fix
- But this is still a start?

C. L. 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 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, pages 3–13, 2012.

Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In ICSE, pages 254–265, 2014.

W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pages 356–366. IEEE, 2013.

A second look...

- What were these fixes?
  - 104 of 110 considered fixes were *deleting* code selected by fault localization algorithms.
  - This removed relevant functionality.
  - Because of weak tests that checked for errors rather than correct output, appeared to fix defect
- What happens with better tests?
  - Only generates patch for 2 of 105 considered defects (!?!), which were already best possible case
  - Somewhat less promising…

Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. *International Symposium on Software Testing and Analysis* (ISSTA 2015), 24-36.

Synthesis with Prophet

- **Defect Localization:** The Prophet defect localization algorithm analyzes execution traces of the program running on the test cases in the test suite. The result is a ranked list of target program statements to patch (see Section 3.7). Prophet prioritizes statements that are frequently executed on negative inputs (for which the unpatched program produces incorrect results) and infrequently executed on positive inputs (for which the unpatched program produces correct results).
- **Search Space Generation:** Prophet generates a space of candidate patches, each of which modifies one of the statements identified by the defect localization algorithm.
- **Universal Feature Extraction:** For each candidate patch, Prophet extracts features that summarize relevant patch properties. These features include *program value features*, which capture relationships between how variables and constants are used in the original program and how they are used in the patch, and *modification features*, which capture relationships between the kind of program modification that the patch applies and the kinds of statements that appear near the patched statement in the original program. Prophet converts the extracted features into a binary feature vector.
- **Patch Ranking and Validation:** Prophet uses the learned model and the extracted binary feature vectors to compute a probability score for each patch in the search space of candidate patches. Prophet then sorts the candidates according to their scores and validates the patches against the supplied test suite in that order. It returns an ordered sequence of patches that validate (i.e., produce correct outputs for all test cases in the test suite) as the result of the patch generation process.

Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering* (ICSE '16), 702-713.

Prophet mutation operators

- **Condition Refinement:** Given a target if statement to patch, the system transforms the condition of the if statement by conjoining or disjoining an additional condition to the original if condition. The following two patterns implement the transformation:

```
if (C) { ... } => if (C && P) { ... }
if (C) { ... } => if (C || P) { ... }
```

  Here `if (C) { ... }` is the target statement to patch in the original program. `C` is the original condition that appears in the program. `P` is a new condition produced by a condition synthesis algorithm [18, 20].
- **Condition Introduction:** Given a target statement, the system transforms the program so that the statement executes only if a guard condition is true. The following pattern implements the transformation:

```
S => if (P) S
```

  Here `S` is the target statement to patch in the original program and `P` is a new synthesized condition.
- **Conditional Control Flow Introduction:** Before a target statement, the system inserts a new control flow statement (return, break, or goto an existing label) that executes only if a guard condition is true. The following patterns implement the transformation:

```
S => if (P) break; S
S => if (P) continue; S
S => if (P) goto L; S
```

  Here `S` is the target statement to patch in the original program, `P` is a new synthesized condition, and `L` is an existing label in the procedure containing `S`.
- **Insert Initialization:** Before a target statement, the system inserts a memory initialization statement.
- **Value Replacement:** Given a target statement, replace an expression in the statement with another expression.
- **Copy and Replace:** Given a target statement, the system copies an existing statement to the program point before the target statement and then applies a Value Replacement transformation to the copied statement.

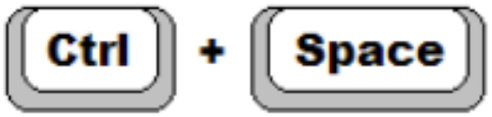
Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering* (ICSE '16), 702-713.

Prophet results

App	LoC	Tests	Defects/ Changes	Plausible					Correct				
				Prophet	SPR	Kali	GenProg	AE	Prophet	SPR	Kali	GenProg	AE
libtiff	77k	78	8/16	5/0	5/0	5/0	3/0	5/0	2,2/0	1,1/0	0/0	0/0	0/0
lighttpd	62k	295	7/2	3/1	3/1	4/1	4/1	3/1	0,0/0	0,0/0	0/0	0/0	0/0
php	1046k	8471	31/13	17/1	16/1	8/0	5/0	7/0	13,10/0	10,9/0	2/0	1/0	2/0
gmp	145k	146	2/0	2/0	2/0	1/0	1/0	1/0	1,1/0	1,1/0	0/0	0/0	0/0
gzip	491k	12	4/1	2/0	2/0	1/0	1/0	2/0	1,1/0	1,0/0	0/0	0/0	0/0
python	407k	35	9/2	5/1	5/1	1/1	0/1	2/1	0,0/0	0,0/0	0/1	0/1	0/1
wireshark	2814k	63	6/1	4/0	4/0	4/0	1/0	4/0	0,0/0	0,0/0	0/0	0/0	0/0
fbc	97k	773	2/1	1/0	1/0	1/0	1/0	1/0	1,1/0	1,0/0	0/0	0/0	0/0
Total			69/36	39/3	38/3	25/2	16/2	25/2	18,15/0	16,11/0	2/1	1/1	2/1

Keyword constraints

```
public List<String> getLines(BufferedReader in) throws Exception {
    List<String> lines = new Vector<String>();
    while (in.ready()) {
        add line
    }
    return lines;
}
```




```
public List<String> getLines(BufferedReader in) throws Exception {
    List<String> lines = new Vector<String>();
    while (in.ready()) {
        lines.add(in.readLine());
    }
    return lines;
}
```

- Explore space of expressions, scoring by match of identifiers in expression to provided keywords
- Use in scope variables as leafs in exploration

Greg Little and Robert C. Miller. 2007. Keyword programming in java. *International conference on Automated software engineering* (ASE '07), 84-93.

User study

task	desired expression
1	message.replaceAll(space, comma)
2	new Integer(input)
3	list.remove(list.length() - 1)
4	fruits.contains(food)
5	vowels.indexOf(c)
6	numberNames.put(key, value)
7	Math.abs(x)
8	tokens.add(st.nextToken())
9	message.charAt(i)
10	System.out.println(f.getName())
11	buf.append(s)
12	lines.add(in.readLine())
13	log.println(message)
14	input.toLowerCase()
15	new BufferedReader(new FileReader(filename))

Table 3: Missing Expressions for Tasks

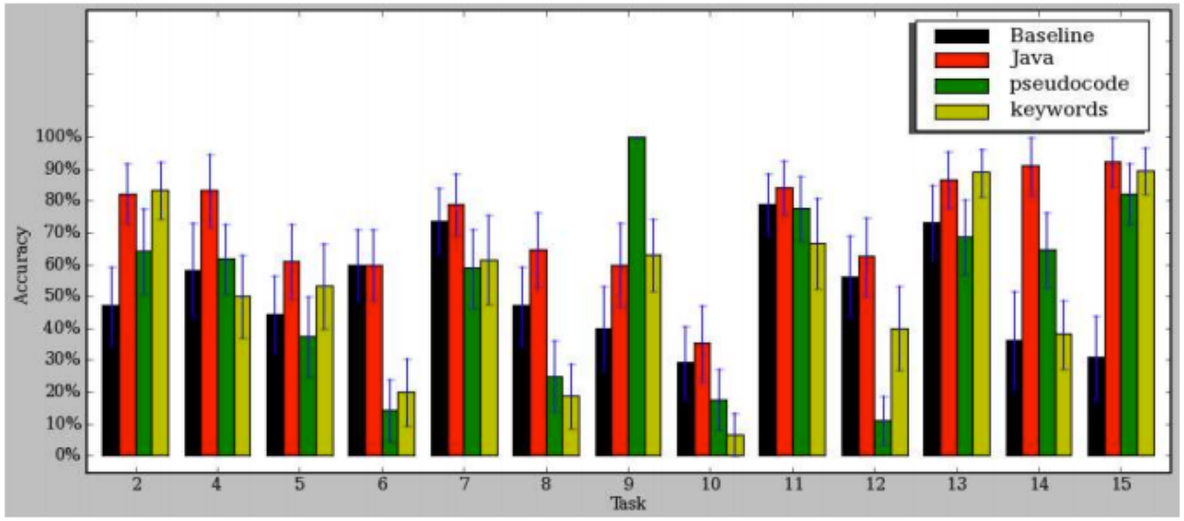


Figure 10: Accuracy of the algorithm for each task, and for each instruction type, along with standard error. The "Baseline" refers to Java responses treated as Java, without running them through the algorithm.

Greg Little and Robert C. Miller. 2007. Keyword programming in java. *International conference on Automated software engineering* (ASE '07), 84-93.

Programming by demonstration

- Program is a set of operations with effects recorded by the user
  - e.g., click a button, enter String in textbox
- User expresses constraints by recording multiple traces
- Goal is to generate program that has same output on demonstrated examples but also work on other similar situations
- Example
  - User selects the first entry from Google search result, pastes that into a form field on another website
  - User demonstrates doing this once (or twice)
  - Want a program that will work for all search results returned by Google

Programming with constraints

- What happens if the specification is underspecified (ambiguity) or there are multiple conflicting specifications (over specification)

- Key idea: communicate ambiguity to user to offer choices and prevent conflicts when users to create them

Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by manipulation for layout. In *Proceedings of the 27th annual ACM symposium on User interface software and technology* (UIST '14), 231-241.

Demo

https://www.youtube.com/watch?v=EDS82S9QMaM

Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by manipulation for layout. In *Proceedings of the 27th annual ACM symposium on User interface software and technology* (UIST '14), 231-241.

Approach

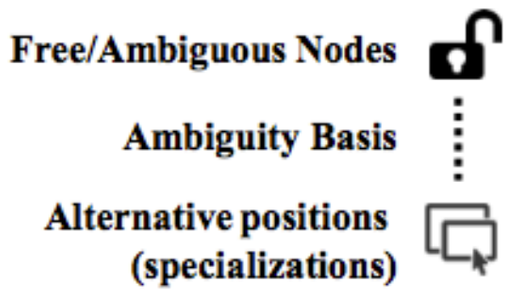
Starting Configuration

User identifies undesirably positioned elements (here, inner nodes and leaves are vertically aligned with the root). Next, he drags the incorrectly positioned element(s), as if breaking the layout constraints that hold the element(s) in the wrong position. This is a *What-is-Wrong* (WiW) manipulation.

WiW Manipulation

PBMM uses the manipulation to relax (generalize) the layout constraints so that elements dragged in the manipulation become unconstrained and are thus free to move. PBMM also computes the alternative sets of constraints that can be enabled to make the layout constraints unambiguous (specializations).

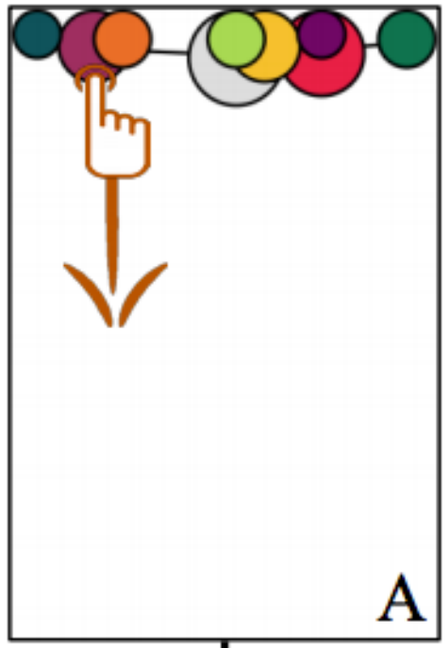
Generalization



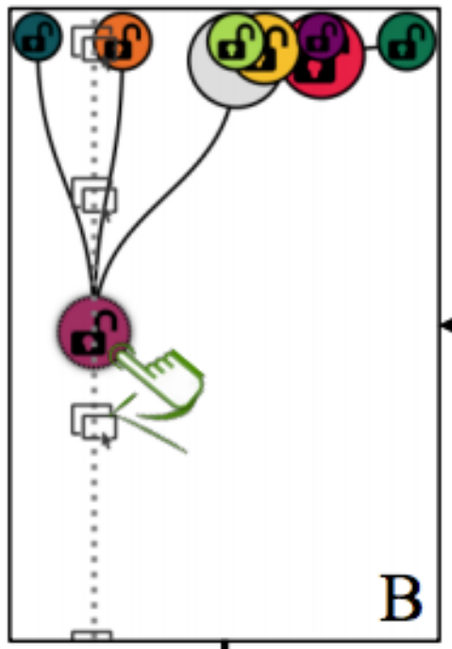
User examines alternative layouts by dragging the element along the ambiguity basis. He selects the desired layout by dropping the element into that position.

Specialization

PBMM



Disables constraints that set the vertical positions of inner nodes and leaves.

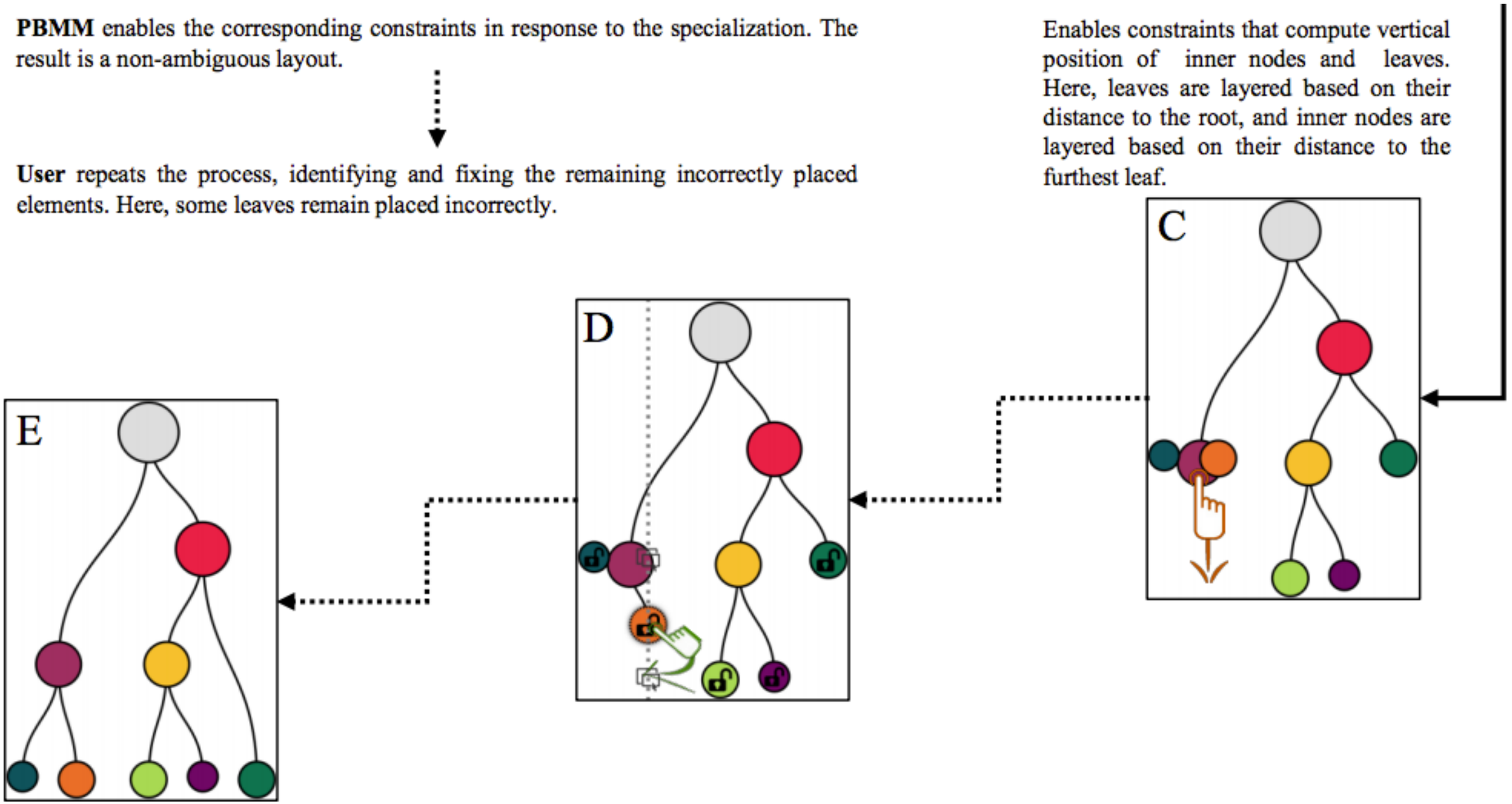


Approach

PBMM enables the corresponding constraints in response to the specialization. The result is a non-ambiguous layout.

User repeats the process, identifying and fixing the remaining incorrectly placed elements. Here, some leaves remain placed incorrectly.

Enables constraints that compute vertical position of inner nodes and leaves. Here, leaves are layered based on their distance to the root, and inner nodes are layered based on their distance to the furthest leaf.



Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by manipulation for layout. In *Proceedings of the 27th annual ACM symposium on User interface software and technology* (UIST '14), 231-241.

Synthesizing transformations

- Let developers specify a *change* in behavior
  - e.g., corrections made to fix a bug

- Cluster code snippets
- Edit code snippet to fix defect
- Try to generalize to transformation that can be applied to other similar code snippets

Clustering mistakes

Figure 3. MISTAKEBROWSER interface: On the left panel, teachers can find information about the current cluster, such as an example of the synthesized fix (A); the total number of submissions in the cluster (B); the failing test case input, the expected output, and the actual output produced by the incorrect submissions (C). The center column shows the incorrect submissions before and after the synthesized fix (D). Finally, on the right panel, instructors can add explanations about student mistakes (E).

Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis,
A Head, **EL Glassman**, G Soares, R Suzuki, L Figueredo, L D'Antoni and B Hartmann, ACM Learning at Scale, 2017.

Authoring transformations

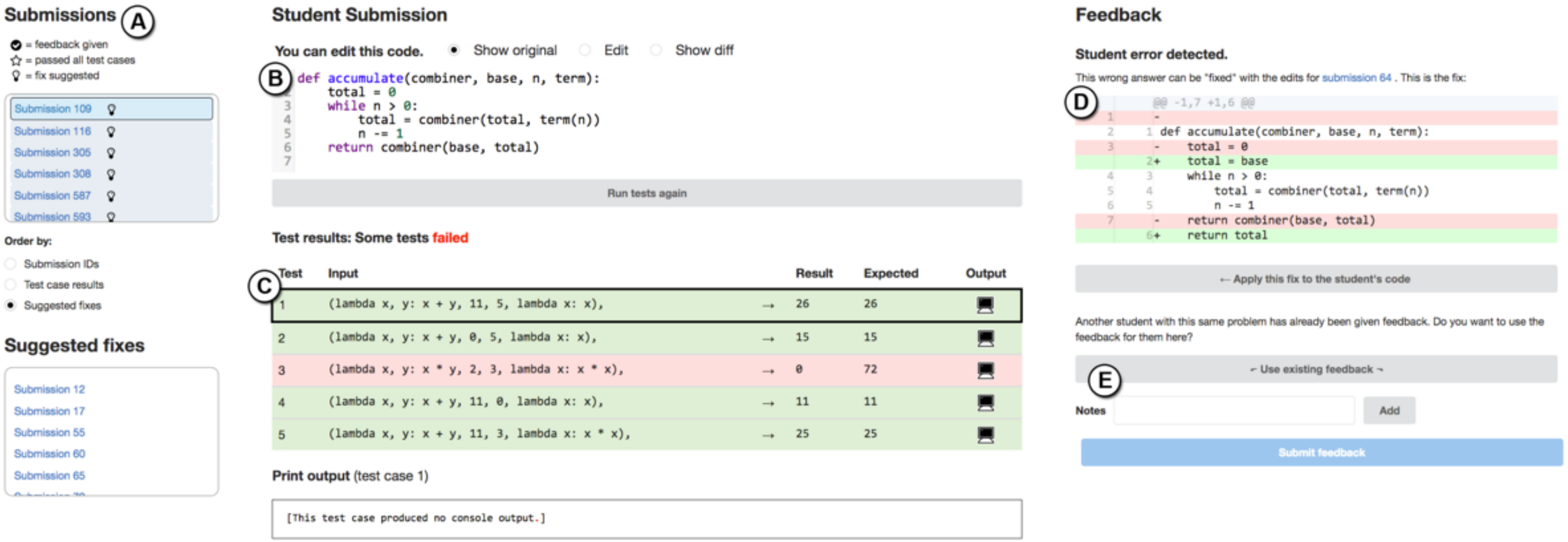


Figure 4. FIXPROPAGATOR interface: The left panel shows all of the incorrect submissions (A). When the teacher selects one, the submission is loaded into the Python code editor in the center of the interface (B). Then the teacher can edit the code, re-run tests, and inspect results. The bottom of the center panel shows the list of tests and console output (C). Once the teacher has fixed the submission, they add some hint that will be shown to current and future students fixed by the same transformation. The bottom of the left panel shows submissions for which the system is suggesting a fix. When the teacher selects a suggested fix, it is shown as a diff in the right panel (D). The teacher can reuse the previously written hint or create a new one (E).

Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis,
A Head, **EL Glassman**, G Soares, R Suzuki, L Figueredo, L D'Antoni and B Hartmann, ACM Learning at Scale, 2017.

Further Challenges

- Are synthesized programs as good as a human solution?
  - Is it as efficient?
  - Does it violate style rules?
  - Does it violate hidden design constraints?

- May depend on how synthesis is used
  - If humans are inspecting the code anyway (autocomplete), does it matter, since they correct it
  - If goal is to automate bug fixing, will they trust it?

Does synthesis help developers?

- If a synthesis suggests a snippet, can a developer judge if it is the right one?

- Brandt model suggests that might be most useful for *reminding* cases where developers knows API already
- May be less useful where developers are trying to learn API