CodeHint: Dynamic and Interactive Synthesis of Code Snippets

Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, Koushik Sen
University of California, Berkeley, USA
{joel,reames,bodik,bjoern,ksen}@cs.berkeley.edu

ABSTRACT

There are many tools that help programmers find code fragments, but most are inexpressive and rely on static information. We present a new technique for synthesizing code that is dynamic (giving accurate results and allowing programmers to reason about concrete executions), easy-to-use (supporting a wide range of correctness specifications), and interactive (allowing users to refine the candidate code snippets). Our implementation, which we call `CodeHint`, generates and evaluates code at runtime and hence can synthesize real-world Java code that involves I/O, reflection, native calls, and other advanced language features. We have evaluated `CodeHint` in two user studies and show that its algorithms are efficient and that it improves programmer productivity by more than a factor of two.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Experimentation, Languages

Keywords

Program synthesis, IDE

1. INTRODUCTION

Many code fragments are difficult to write, often because they involve using a new and unfamiliar API. Programmers have many tools at their disposal, from search and autocomplete to advanced synthesis techniques, but these often have only limited applicability. For example, autocomplete implementations can be helpful at finding a method to call, but they require some knowledge of the static type of the receiver and can only generate tiny code fragments.

ICSE '14, May 31 - June 7, 2014, Hyderabad, India

To solve the problem of synthesizing code fragments, we believe that programmers need to be able to give partial specifications (rather than full correctness conditions) that may depend on concrete program state. Our approach thus generates and evaluates code at runtime at a programmer-chosen location while executing a specific concrete input.

We propose a new approach for synthesizing code that is dynamic, easy-to-use, and interactive. Running in the dynamic context both allows us to find and filter out candidates that static techniques could not and allows users to reason concretely about their desired result. We support a wide variety of specifications so that we can even aid programmers with very little knowledge of their desired code. Our methodology is interactive, letting users incrementally give more information to refine the candidate code fragments.

Taking advantage of dynamic information allows our algorithms to be more accurate than static techniques by, for example, dereferencing exactly the expressions that do not evaluate to `null` in the current context and downcasting the result of a method call to its dynamic type to enable subsequent calls. In addition, users can use dynamic values in their specifications and see the results of executing the candidates, which can be helpful in choosing the correct result.

One major goal of our work has been to ensure that users can find code snippets using whatever partial information they have about the desired result. Our partial dynamic specifications or *pdspecs* can be any predicate in the host language. This flexibility allows us to represent the full spectrum of specification strength and context sensitivity. For example, our pdspecs can take the form of constraints on the desired value (including demonstrating a concrete value), dynamic type restrictions, or even full functional correctness specifications. We also allow users to write code *skeletons* to shape the search space by giving a syntactic outline of the desired code with holes marking unknown fragments.

Given a set of candidate statements synthesized by our tool, users can refine this set by continuing to run the program or by exercising it on different inputs in order to filter out more candidates that fail the specification. This refinement process can quickly remove many undesirable statements. Users can also sort and filter the candidates and their results. These features often allow users to find their desired code even with simple pdspecs.

Another goal of our work is to enable the synthesis of code in real-world Java programs. Because it actually executes candidates in a concrete program state, our implementation can synthesize code that uses I/O, reflection, native calls, and more. We propose novel new techniques for using stan-

```
1 final JComponent tree = makeTree();
2 tree.addMouseListener(new MouseAdapter() {
3   public void mousePressed(MouseEvent e) {
4     int x = e.getX(), y = e.getY();
5     Object o = null;
6     // Get the menu bar or the clicked element.
7   }
8 });
```

Figure 1: Code that handles clicks on a graphical tree of elements.

dard features of JVMs such as breakpoints to ensure that our evaluations have no undesirable side effects, which users can selectively enable or disable to control the tradeoff between efficiency, soundness, and completeness. By analyzing over ten million lines of code, we have developed a probabilistic model of real-world Java code that helps guide our search toward more common methods and fields.

To demonstrate the benefits of our approach, we conducted two user studies involving 28 subjects solving multiple programming problems in different domains such as GUIs, string parsing, and Eclipse plugins. The statistically significant results show that subjects using our tool complete more tasks in less time and with fewer bugs than those without it by more than a factor of two. Users gave our tool positive subjective ratings.

In summary, the main contributions of this work are:

- A new method (Section 3.1) for synthesizing code that is dynamic (giving accurate results and allowing programmers to reason about concrete executions), easy-to-use (supporting a wide range of correctness specifications), and interactive (allowing users to refine the candidate code snippets).
- An efficient algorithm (Section 3.2) that exploits the dynamic context to generate candidate statements that can include advanced features of the host language such as I/O, reflection, and native calls.
- An implementation (Section 4) as a plugin for the Eclipse IDE called `CodeHint` that synthesizes Java code, including Android programs.
- Empirical evaluations and user studies (Section 5) that show that `CodeHint` is efficient and significantly improves programmer productivity.

2. OVERVIEW

We now present two example problems and show how `CodeHint` can solve them. The first demonstrates `CodeHint`'s algorithms while the second shows the user's perspective. Both examples involve the GUI code using the Java Swing toolkit shown in Figure 1. Readers are also encouraged to watch a demo video of `CodeHint` at `http://www.cs.berkeley.edu/~joel/codehint/`.

2.1 Algorithm Example

Imagine that a user writes the partial code shown in Figure 1 and then wants to write code at line 6 to find the menu bar for the window that contains the graphical tree and store it in the variable `o`. A simple Internet search will reveal that the menu is represented by a `JMenuBar` object but will likely provide little information on how such an object can be acquired.

To use `CodeHint` to find this object, the user can set a breakpoint after line 5 (e.g., near the comment on line 6) and run the program to that breakpoint, which is the current context. Since she knows that she wants to assign a value of type `JMenuBar` to the variable `o`, she can provide the specification `o′ instanceof JMenuBar` to `CodeHint`. We call this specification a pdspec, and the `o′` denotes the value of the variable `o` after the code to be synthesized is evaluated. Thus this pdspec encodes the fact that `o` should change so that it contains an object of type `JMenuBar`.

Given this query, `CodeHint` will begin a search for expressions that it can assign to `o` to try to satisfy the pdspec. This iterative search will start with local variables and generate larger expressions with operations such as addition and method calls. `CodeHint` will evaluate these expressions in the current context, undoing side effects as they occur, to enable it to get precise results that satisfy the user's pdspec. A probabilistic model will guide the search toward more likely expressions and `CodeHint` will group equivalent expressions together to avoid duplicate work.

Once this search is complete, `CodeHint` will show the user approximately five results. She can then add a new testcase to the code and run that new input, which will allow `CodeHint` to remove two of the previous results that crash in the new context.

We now walk through `CodeHint`'s algorithm in detail.

First iteration.

`CodeHint` will first query the debugger for all the variables in scope, evaluating them so it knows their dynamic types. These, along with the special values `null` and `this`, will become the initial set of candidate expressions that `CodeHint` is considering:

```
tree, e, x, y, o, this, null
```

Second iteration.

In its next step, `CodeHint` will combine these simple expressions into more complicated ones according to the Java grammar. To do this, it will first query the debugger for the dynamic type of each candidate.

For each object, it will query the debugger for all accessible methods available on that type (and its supertypes) and then call each of those methods with all type-safe combinations of the previous set of candidates as arguments (downcasting when necessary).

As an example, `CodeHint` will find that `tree` is a value of type `JTree` (which is a subtype of `JComponent`, the static type of the variable). It will then ask the debugger for the methods of `JTree`, one of which is `getPathForLocation`. This method expects two integer arguments, so `CodeHint` will find all of the integer-valued expressions in its previous set of candidates, which in this case are `x` and `y`. `CodeHint` will then call this method with all possible combinations of these arguments, producing the following four calls:

```
((JTree)tree).getPathForLocation(x, x)
((JTree)tree).getPathForLocation(x, y)
((JTree)tree).getPathForLocation(y, x)
((JTree)tree).getPathForLocation(y, y)
```

`CodeHint` will repeat this process for other methods and for

static methods of classes imported in the current file.

For each pair of primitives, such as integers, CodeHint will combine them with binary operations. Thus given x and y, it will generate the following expressions:

```
x + y, x - y, y - x, x * y, x / y, y / x,
x == y, x != y, x < y, x <= y, x > y, x >= y
```

It will similarly generate object comparisons, array accesses and length, field accesses, integer and boolean negation (e.g., -x), addition and subtraction with 1 (e.g., x + 1 and x - 1), comparisons with 0 (e.g., x > 0 and x < 0), and boolean conjunctions and disjunctions.

CodeHint will evaluate each expression as it is generated so that it knows its result. However, expressions with side effects must be handled correctly so they do not affect future evaluations. For example, tree.add(makeTree()) will modify the tree, potentially causing future evaluations to return different results in this new context.

To avoid this problem, CodeHint uses novel techniques based on breakpoints and the Java security manager to undo in-memory side effects after they occur and block harmful native calls. Harmless native calls, such as reading from a file, proceed normally. In practice, we have found that allowing users to relax these restrictions often gives correct results in less time. In this case, evaluating tree.add(makeTree()) will modify a field of tree. This will trigger a breakpoint installed by CodeHint, which will log the change and undo it after the evaluation finishes.

Once this process is complete, CodeHint will have a new set of approximately 240 candidate expressions:

```
tree, e, x, y, o, this, null, x + y, x < y,
tree.getTopLevelAncestor(), Window.getWindows(),
((JTree)tree).getPathForRow(x), ...
```

The set of candidates can grow quite large, so CodeHint applies some optimizations that make it significantly smaller:

- To avoid duplicating work, CodeHint builds equivalence classes of expressions based on their results and only retains one representative of each class in its set of candidates. For example, tree.getTopLevelAncestor() is equivalent to SwingUtilities.getRoot(tree) in the current context, so only one (the former in this case) will be included in the set of candidates. As we will see shortly, CodeHint will use the equivalent expressions that are not in the list of candidates to help generate the results it shows to the user by substituting equivalent subexpressions.

- To avoid spending time searching expressions that are rarely used in practice, CodeHint uses a probabilistic model to avoid unlikely method calls and field accesses. This model contains information from over ten million real-world lines of code. As one example, the JTree class contains a method called getNextMatch that was not called once in all of the analyzed code, so CodeHint will not call it in this iteration.

CodeHint would have approximately 370 candidates without either of these optimizations and approximately 560 without both. Since these candidates are used to generate more expressions in the next iteration, these optimizations significantly improve performance.

None of these candidates meet the user's specification, as none have type JMenuBar, so CodeHint will automatically continue this process of creating larger expressions from its current candidates.

Third iteration.

The third iteration proceeds exactly as the second: it combines the current candidates to produce larger expressions.

In this iteration, CodeHint will use its probabilistic model to avoid searching some additional expressions. Libraries such as Swing often contain many constants that are intended to be used only in certain contexts. For example, KeyEvent.VK_ENTER helps determine if the user pressed the Enter key. To avoid using such constants in unrelated contexts, CodeHint's probabilistic model stores exactly how they are used in practice. This allows it to avoid generating expressions such as tree.getComponent(KeyEvent.VK_ENTER), as it recognizes that KeyEvent.VK_ENTER was never used as an argument to getComponent in the analyzed code.

During this iteration, CodeHint will find that the expression tree.getTopLevelAncestor(), whose static return type is Container, has type JFrame at runtime. CodeHint will then explore all the methods it can call on a JFrame, including getJMenuBar, which returns a JMenuBar. It will thus add ((JFrame)tree.getTopLevelAncestor()).getJMenuBar() to its next set of candidates.

At this point, CodeHint will also explore some expressions that it avoided before due to its probabilistic model, such as calls to JTree.getNextMatch. This allows the algorithm to prioritize more likely expressions while remaining complete.

Since it calls each method with all possible type-safe combinations of the previous set of candidates, CodeHint might end up making an exorbitant number of calls to a single method. For example, in this iteration it attempts to call DefaultTreeCellRenderer.getTreeCellRendererComponent. This method has seven arguments, including one of type Object (of which CodeHint has seen over 70 unique values), one of type int (of which CodeHint has seen over one hundred unique values), and four booleans (each of which can be true or false), which would result in well over 100,000 calls. To avoid spending so much time calling a single method, CodeHint only calls it a small number of times. Specifically, if a method can be called more than a certain number of times that grows exponentially with the current iteration, CodeHint calls it at most that many times with arguments that were candidates from previous iterations.

At the end of this iteration, CodeHint will have generated and evaluated over 700 expressions.

Once it has finished this process, CodeHint will find which of its candidates satisfy the user's pdspec. In this case, ((JFrame)tree.getTopLevelAncestor()).getJMenuBar(), when assigned to o, is the only expression that satisfies the pdspec. Before showing it to the user, CodeHint will generate more satisfying expressions by replacing its subexpressions with equivalent expressions.

For example, because SwingUtilities.getRoot(tree) is equivalent to tree.getTopLevelAncestor(), CodeHint knows ((JFrame)SwingUtilities.getRoot(tree)).getJMenuBar() will yield the same result and hence also satisfy the user's pdspec. CodeHint will thus generate this expression and three others that are all equivalent to the original expression.

Now that it has expressions to show to the user, CodeHint will use its probabilistic model to present the user with the more likely options closer to the top. A call's likelihood is the number of calls to it that the model contains and an

expression's likelihood is the product of its subexpression's likelihoods. In this case, the `getTopLevelAncestor` method was called 19 times in the model while `getRoot` was called 15 times, so the former expression is shown first.

`CodeHint` will then show the user these expressions that, when assigned to `o`, meet the specification (as well as their values, side effects, and string representations):

```
((JFrame)SwingUtilities.getWindowAncestor(jtree))
  .getJMenuBar()
((JFrame)tree.getTopLevelAncestor()).getJMenuBar()
((JFrame)SwingUtilities.getRoot(tree)).getJMenuBar()
...
```

If `CodeHint` had not found any results that satisfied the user's pdspec at this point, it would have notified the user and asked if it should continue for another iteration.

Note that the results `CodeHint` found all rely on being able to discover the dynamic type of an expression and downcast to it, something that static tools will find difficult to do.

Refinement.

The user can now examine these expressions, their results, and their documentation to try to pick the one she wants to use. If she is unsure which is correct, she can refine the set of results by running the program on a different input.

Let us assume that the user modifies the code so that the tree is contained within an applet that is itself contained within the top-level window, which might be useful for running the same code both by itself and within a web browser. Once she makes this change, she can run the new program until it reaches the point where she ran the previous search. `CodeHint` will then evaluate the previous results in the new context. In this case, two of them, including `((JFrame)tree.getTopLevelAncestor()).getJMenuBar()`, will crash. `CodeHint` will show this smaller set of results to the user, who may select one of them to use or continue the process with yet another input.

2.2 User Perspective Example

A common task when writing GUI code is to detect clicks on a graphical tree of elements. A programmer might be unsure how to find the clicked element so she can use it. Unfortunately, as she does not know the API, she is not even sure what type this object has; it could be a node object, the data it represents, or the displayed string. As she does not even know the type of the expression she desires, it is difficult to find. Using `CodeHint`, she can easily run the code, click on a node, and give a pdspec that expresses which node she clicked.

To use `CodeHint` to find the code she desires, she can set a breakpoint after line 5 and then execute her code and click on the top element. Knowing that in Java most objects have a `toString` method that gives a string representation of their value, she realizes that if she clicks on an element labeled "Alice", the `toString` of her desired result should contain that string. This insight can lead her to enter the pdspec `o'.toString().contains("Alice")`, which encodes the fact that the value of the variable `o` should be updated by the desired statement so that its `toString` contains "Alice". `CodeHint` will then generate eight expressions to assign to `o`:

```
((JTree)tree).getPathForLocation(x, y),
((JTree)tree).getSelectionPath(),
((JTree)tree).getLastSelectedPathComponent(), ...
```

To reduce the number of candidates, the user can continue the execution (or restart it) and click on a different element. Assume she does so and clicks on an element labeled "Bob". When the execution suspends at the breakpoint, `CodeHint` will show her all eight of the previous expressions with their results in the new context. She can then give a new pdspec to filter out some expressions. Giving `o'.toString().contains("Bob")` will remove one expression. Alternatively, by looking at the results of the eight expressions she might see that many return an object of type `TreePath` that seems to do what she wants, so she can use the pdspec `o' instanceof TreePath` to keep only those.

Assume that now the user clicks below all the elements. She can then see all of the remaining candidate expressions with their new values and keep only those that evaluate to `null` with the pdspec `o' == null`. `CodeHint` will then eliminate all but one candidate and find the correct code:

```
o = ((JTree)tree).getPathForLocation(x, y);
```

This example shows how our approach can be useful even when a programmer has very little knowledge about her desired result, perhaps not even its type, by allowing her to refer to values in the concrete program state. The programmer narrows down the set of candidate statements by incrementally providing pdspecs for different test scenarios. A key advantage of our methodology is that programmers can mix and match value demonstrations, type specifications, and arbitrary pdspecs as desired. In addition, the interactivity of `CodeHint` allowed the user to find the correct result from the initial candidates. This example was inspired by how one subject in our first user study solved this problem.

This example also shows how our approach can easily synthesize real code involving complicated libraries. While the final synthesized code appears simple, executing it involves making over 70 method calls that allocate new objects and use complex objects, generics, and binary-only libraries.

Using skeletons.

Perhaps now the user wants to get the data representing the object she clicked out of the `TreePath` object she just assigned to `o`. Let us assume that she changes `o`'s type in the code and discovers from the type's documentation that it contains a `getPathComponent` method that she thinks will help her. However, she is unsure what argument to pass to this method.

The user can encode this knowledge into a skeleton that `CodeHint` will use to guide its search. Specifically, she can enter the skeleton `o.getPathComponent(??)` where the `??` represents the missing portion of the code. Given this skeleton, `CodeHint` will search for expressions that can be used as arguments to `getPathComponent`, which expects an integer.

As before, the user can set a breakpoint where she wants to insert code, run the program on a test, and click on an element to hit the breakpoint. If she clicks on "Eve", she can enter the pdspec `_rv'.toString().equals("Eve")` to show that she now wants some object that represents "Eve" (the `_rv'` represents the return value of the expression) along with the skeleton discussed above to guide the search.

`CodeHint` will now search for integers that, when passed to `getPathComponent`, meet the user's specification, which include the following:

```
o.getPathComponent(e.getClickCount())
o.getPathComponent(o.getPathCount() - 1)
```

By refining the set of candidates by giving different pdspecs in different states, the user can remove incorrect expressions such as the first one above.

This example shows how users can encode their partial knowledge of the correct code to allow `CodeHint` to focus its search on relevant code snippets.

3. PDSPECS AND SYNTHESIS ALGORITHM

We now define pdspecs and our synthesis algorithm.

3.1 Pdspecs and Our Approach

Without loss of generality, let us assume that we have an incomplete program in which a statement, say s_T, is missing. The developer wants to synthesize this statement using `CodeHint`. She creates a test input so that the execution of the program on the test input reaches the program location, say ℓ, that is just before the missing statement. Let σ be the program state when the program reaches the program location ℓ. We use Σ to denote the set of all feasible program states. The goal of `CodeHint` is to discover the missing statement s_T. Let $\text{exec}(\sigma, s)$ be the program state obtained by executing the statement s in state σ. Let us use σ' to denote $\text{exec}(\sigma, s_T)$, i.e., the program state reached after the user executes the missing statement s_T in the state σ. As a running example, let us assume that σ is $\{x \mapsto 42\}$ and s_T is `x = 2 * x`. Then σ' is $\{x \mapsto 84\}$.

The user does not know the missing statement s_T, but she might have a good idea about what the program state should look like after the execution of s_T (i.e., σ'). With `CodeHint`, the user can indirectly provide a hint about the statement s_T by giving information about the state σ' using pdspecs. A pdspec can give absolute information about σ' by describing the updated program variables and their corresponding values or it can be a predicate relating the states σ and σ'. For example, a pdspec can specify `x' == 84`, where `x'` represents the value of `x` in state σ', or it could specify `x' > x`, a predicate that relates the states σ and σ'. In general, a pdspec can consist of any expression in the host language.

Note that a pdspec that is specified in the state σ may not be a correct pdspec if the program is in a different state at the location ℓ. (A different state at location ℓ can be reached by executing the program on a different test input.) For example, `x' > x` is not a correct pdspec if the program state at location ℓ is $\{x \mapsto -20\}$, but it is a correct pdspec if the state is $\{x \mapsto 10\}$. Similarly, `x' == 84` is not a correct pdspec for any program state with `x != 42`. Note that `x' == 2 * x` is a correct pdspec for all program states that reach the location ℓ.

Formally, a pdspec is a logical predicate $\phi \in \Sigma \times \Sigma \to \{\text{true}, \text{false}\}$ where $\phi(\sigma, \sigma')$ checks whether σ' is a desired output state given the input state σ. As a notational convenience, for a pdspec $\phi(\sigma, \sigma')$, we refer to variables in the input state σ using their names and variables in the output state σ' using their primed names. All variables not given in a pdspec must be equal in the two states and all expressions in a pdspec must be free of side effects.

The user of `CodeHint` gives a sequence of pairs of program states at ℓ (reached by executing the program on different test inputs) and their corresponding correct pdspecs: $(\sigma_0, \phi_0), (\sigma_1, \phi_1), \cdots$. Let S be the set of all syntactically valid statements that can be written at location ℓ. Based on the sequence of pairs of states and pdspecs, `CodeHint` returns a set of candidate program statements that could replace the missing statement. For example, if the user provides the sequence of pairs $(\sigma_0, \phi_0), \cdots, (\sigma_i, \phi_i)$, the set of suitable statements for the location ℓ is reduced to the set

$$C_i = \left\{ s \in S : \bigwedge_{0 \le j \le i} \phi_j(\sigma_j, \text{exec}(\sigma_j, s)) \right\}.$$

The predicate $\phi_j(\sigma_j, \text{exec}(\sigma_j, s))$ is true if the state σ_j and the state obtained after executing statement s in the state σ_j satisfy the pdspec ϕ_j. Statement s is in the candidate set if this predicate is true for all j. If a statement s is in the candidate set, it implies that at least for the program states $\sigma_1, \cdots, \sigma_i$ observed at the location ℓ, the execution of the statement s will result in the user's expected program state. If we could compute the candidate set for all possible pairs of programs states reachable at ℓ and their corresponding correct pdspecs, the set is guaranteed to contain the target statement s_T. However, in practice it not possible to enumerate all such possible pairs. Instead, the user demonstrates such pairs one-by-one and `CodeHint` computes a candidate set from the pairs provided by the user. Note that $C_{i+1} \subseteq C_i$, so the size of the candidate set shrinks as `CodeHint` receives more pairs. At any point, if the user notices a statement she could use as a substitue for the missing statement, she stops the process and uses the statement.

As the set of legal statements S could be infinite, `CodeHint` restricts it to a finite set. For example this set could only include statements whose abstract syntax trees have height at most k and only use the variables available in the current scope. We call k the depth of the search space.

In practice, users often have some idea of the structure of the statement they desire. In our running example, the user might know that she wants to multiply `x` by something but might not know exactly what should be multiplied. By providing a *skeleton* (discussed more in Section 3.2), the user can further restrict the search space. Here, this skeleton might be `x = ?? * x`, where `??` could be replaced by any valid expression.

In our running example, given the initial state $\{x \mapsto 42\}$, the pdspec `x' > 42`, and the skeleton `x = ?? * x`, we present the user with the following set of candidate statements:

$C_0 = \{$ `x = 2 * x, x = x * x, ` $\cdots$ $\}$

Without the skeleton, we would additionally have included statements like `x = 84`, `x = x + 1`, and `x = x + x`. In either case, the number of candidate statements might be large but will be finite.

Given another initial state $\{x \mapsto 6\}$, the pdspec `x' == 12`, and the same skeleton, we present the user with the subset $C_1 = \{$ `x = 2 * x` $\}$.

3.2 Synthesis Algorithm

Our algorithm generates statements containing Java expressions that include variables, array accesses, casts, field accesses, method calls, constructor calls, and unary and binary operators, including calls to static methods and fields of imported classes.[1] It begins by collecting all the variables in scope and then iteratively uses the current set of statements

[1]There seems to be no technical barrier to extending this language to cover all Java statements (except for anonymous class declarations), but we have seen no need to do so yet.

to generate larger ones. When it reaches a fixed maximum size, it removes those that do not satisfy the user's pdspec.

In particular, our algorithm generates statements in increasing order of *depth*, which is the height of the parse tree. As examples, `x` has depth 1, `foo.bar(x,y)` has depth 2, and `foo.bar.baz(x+y)` has depth 3.

A key insight is that if two statements have the same effects, we can treat them as equivalent when generating further statements. We take advantage of this fact by grouping statements into equivalence classes based on their effects and values. As an example, there might be hundreds of pure boolean-valued expressions at a given depth, but they all evaluate to either `true` or `false`. We henceforth describe our algorithm in this special case of expressions but note that it works for the more general class of statements.

We define an equivalence relation on expressions such that two expressions are equivalent if they have the same side effects and yield the same value in the current state. We write $e_1 \sim_\sigma e_2$ to denote that the expressions e_1 and e_2 have the same side effects and yield the same value in the state σ. Following standard Java idioms and practices, two expressions $\texttt{e}_1$ and $\texttt{e}_2$ are equivalent if both are primitives and $\texttt{e}_1$ `==` $\texttt{e}_2$, both are objects of the same type and $\texttt{e}_1$ `==` `null` `?` $\texttt{e}_2$ `==` `null` `:` $\texttt{e}_1$`.equals(`$\texttt{e}_2$`)`, or both are arrays of the same type with the same number of elements and all corresponding elements are equivalent.

Our algorithm iteratively builds all expressions up to a fixed depth, evaluating them with a timeout and creating equivalence classes as it goes. For each equivalence class, we use only one representative to generate further expressions. For example, if `x` and `z.f(0)` are equivalent, we generate `x+1` and `foo(x)` but not `z.f(0)+1` or `foo(z.f(0))`.

We take the current set of side effects into account when generating new expressions from representatives of these equivalence classes. For example, when `x` is `42` we will notice that `x` and `42` are equivalent. Later, when considering expressions to add to `x++`, we will use the equivalence class for the state where `x` is `43` and consider `x` and `42` separately.

When the desired depth has been achieved, we test the equivalence classes for the initial state against the pdspec, filter out those that do not satisfy it, and recreate the full set of candidate expressions from the equivalence classes. Once we discover that an expression satisfies the pdspec, we know that all expressions generated from it by replacing its subexpressions with expressions that are equivalent in the current state will also satisfy the pdspec. Continuing the previous example, if `p.bar(x)` satisfies the pdspec, so will `p.bar(z.f(0))`.

We apply a number of optimizations to improve the efficiency of our algorithm. We use a variety of simple structural techniques to avoid enumerating obviously equivalent expressions (such as `x+y` and `y+x`). We call the `hashCode` method to speed up checking equivalence between objects. We cache the result of each expression and use that result instead of the expression itself in future evaluations (and replay its side effects) to avoid duplicating work.

By actually evaluating the generated expressions, our algorithm can synthesize real-world Java code, including file I/O, binary libraries, reflection, foreign function calls, and calling the user's own methods.[2] Thus while the individual expressions we generate appear somewhat simple, they can in fact be quite complicated. In addition, we were able to synthesize code for Android without any extra modeling.

[2]Some of these features require the user to allow searching native calls, in which case we cannot undo side effects.

Probabilistic model and pruning.

To guide our algorithm toward exploring more likely expressions, we have added a probabilistic model based on an offline analysis of over ten million lines of code. This allows us to compute how often certain types, methods, and fields are used. Having analyzed how often each method and field is accessed, we define the probability of accessing member `m` (calling a method or accessing a field) of type `T` as

$$P(m|T)P(T) = \frac{\#\text{ accesses of } m \text{ on } T}{\#\text{ of accesses on } T} \times \frac{\#\text{ of accesses on } T}{\#\text{ of accesses}}.$$

The probability of an expression is then the product of the probabilities of its individual accesses. In addition, as Java classes often contain many constants, we store, for each constant field, all the places it is used as an argument to a method. This model is somewhat simplistic, as it assumes that all method calls are independent, but we have found it very helpful in practice.

We use this probabilistic model to avoid calling rare methods and using constants in places they were rarely used as well as to sort the candidates presented to the user.

The size of the space of expressions that our algorithm must search at a given depth is exponential due to method calls. If at one point during a search we have seen 50 different integer values, there will be 50^3 different ways to call a method with three integer arguments. Such growth can easily overwhelm our algorithm, so we heuristically prune calls to methods that would otherwise be called with a large number of different arguments, especially ones our probabilistic model defines as uncommon. That is, if a method can be called more than a certain number of times that is exponential in the current depth, we artificially increase the depth of calls to it. As we will see in Section 5.1, such calls occur infrequently in practice. We believe this would be a promising area in which to integrate symbolic techniques.

Skeletons.

As mentioned earlier, users may provide a skeleton to shape the search space explored and guide it toward candidates they know to be likely. Skeletons consist of normal Java code with *holes* for unknown code that should be synthesized. The language contains two types of holes: simple holes (denoted `??`) and list holes (denoted `**`). Simple holes can take the value of any expression or name in the language or be annotated with a set of candidates. List holes are used for calling functions with an unknown number of arguments, each of which is a separate expression hole. Users may additionally check a box indicating whether or not constructor calls and infix/prefix operators should be searched.

To give some examples, `??.??` represents accessing some field of an unknown expression, `foo.??{bar,baz}(??)` represents calling either the `bar` or the `baz` method of the `foo` object with a single unknown argument, and `??(**)` represents calling an unknown method with any number of unknown arguments.

Since the given pdspec applies to the whole statement, we must explore the cross product of all the specified holes. Given a skeleton, we fill each hole with type-correct values using the algorithm described above (unless the hole has been annotated with candidate values, in which case we

simply try those values in lieu of a search). If a skeleton has many holes, we reduce the depth of our search for each hole.

4. IMPLEMENTATION

We have developed an implementation of our approach, which we call `CodeHint`, as a plugin for the Eclipse IDE for Java.[3] This allows users to develop normally, using our approach only when they wish to do so.

To use `CodeHint`, a programmer must start a debug session and navigate to the program location and state in which she wishes to insert code. She then enters a pdspec and possibly a skeleton through a dialog.

We then synthesize candidate expressions and show them, their results and `toString`s, and their side effects to the user, who can select which to keep. The user may also view the Javadocs of methods and fields used in the candidates and sort or filter them. If she does not find any expressions she wants, she can continue the search with an increased depth.

When execution encounters the location of a previous synthesis, we begin our refinement process by showing the user the previously-chosen candidates and their values in the new context. She may enter a new pdspec to refine the current set of candidates or filter or sort the results. In addition, she may abandon the current results and start a new search, which can be useful if she previously gave an invalid pdspec or if our search did not find any correct statements.

Our implementation detects in-memory side effects while evaluating expressions by installing watchpoints to listen for field accesses. This allows us to log all the side effects of an evaluation and afterwards undo them and show them to the user. We use Java's security manager to disable external side effects such as deleting files and we block unknown native calls (which do not go through the security manager). Users can selectively enable or disable these features to control the tradeoff between efficiency, soundness, and completeness.

5. EVALUATION

We now show, through empirical analysis and two user studies, that `CodeHint` is sufficiently scalable and that it makes users more productive.

5.1 Empirical Evaluation

Scalability.

To analyze the efficiency of our implementation and demonstrate its scalability with regards to depth (as defined in Section 3.2), we ran the tasks used in our first user study (described in Section 5.2.1) with a typical pdspec (i.e., the one most frequently used by subjects in the study) and a skeleton indicating that we should not search constructors and operators. For each task, we varied the maximum depth searched, and for each depth, we recorded the number of unique expressions evaluated and the total time taken.

The experiments were performed on an Intel Core 2 Duo E6850 with two 3 GHz processors and 3 GB RAM (although our current implementation is single-threaded) running Linux 3.13.5 and Java 1.7.0_51.

The results in Table 1 show that our current implementation, when not undoing side effects, can explore a search space of depths 1, 2, and 3 in one second and depth 4 in five seconds. We do not show depth 5, for which we have not optimized and all but four benchmarks timeout after two minutes. When undoing side effects, our unoptimized implementation takes a couple of seconds to search depth 2. We believe these results are more than sufficient for a useful tool, and results from the user studies show that users agree.

As part of our methodology, `CodeHint` finds all expressions in its search space that satisfy the user's pdspec, not just one. Since we display each such candidate expression as soon as we discover it, users can usually see some candidate expressions well before the search terminates.

To measure the effectiveness of various parts of our algorithm, we searched depth 3 with pruning disabled and with pruning, equivalence classes, and the probabilistic model disabled. The results, given in the rightmost columns of Table 1, show that these improvements significantly reduce the search space, enabling `CodeHint` to find larger expressions. Without them, it would not be able to search even depth 3.

To show that `CodeHint` can synthesize large code fragments, we tracked the size of the expressions explored for each task. The largest such code snippet contained ten method calls, showing that our algorithms can indeed scale to non-trivial code fragments.

Analysis of expressions.

To justify our algorithm, we analyzed five medium to large open-source Java projects. The key result is that the vast majority (99%) of assignment statements are actually quite simple (with a depth of at most 4).

We analyzed Hadoop, a framework for distributed computing based on MapReduce with a distributed file system, Tomcat, a web server, FindBugs, a static analysis tool, Hibernate, which maps objects into a database, and JDK, the implementation of Java.

For each program, we analyzed the right-hand side of assignment statements, which is the exact class of statements we currently generate. Hadoop version 1.0.3 contains 47,248 such expressions, Tomcat 7.0.30 contains 30,740, FindBugs 2.0.1 contains 18,589, Hibernate 4.1.7 contains 54,580, and JDK 6.25 contains 227,731.

We began by analyzing the depth of the expressions. All five programs followed a similar pattern: on average, 27% of expressions had depth 1, 52% had depth 2, 15% had depth 3, 5% had depth 4, and 1% were more complicated. This shows that the vast majority (99% in our study) of statements in real Java code have depth at most 4, which our results above show that `CodeHint` can easily search.

To show that these results also hold for code programmers struggle to write, we repeated the same experiment on code snippets gathered from questions asked on the popular Stack Overflow website. We examined 43 code samples containing 3333 lines of code and found nearly the same distribution: 25% had depth 1, 59% had depth 2, 13% had depth 3, and 3% had depth 4. While users might want to synthesize multiple lines of code at a time, we believe these results suggest that our current algorithms can help real programmers.

We conservatively bounded the number of cases in which we pruned calls to methods that could be called with many different arguments by analyzing the number of calls with a given number of arguments. All five programs followed a similar pattern: 51% of calls had no arguments (excluding the receiver), 31% had one argument, 12% had two arguments, 3% had three arguments, and 3% had four or more.

[3] `CodeHint`, its source, and video demos are available at `https://github.com/jgalenson/codehint`.

Table 1: An empirical analysis of our algorithm. Each row represents one task from the first user study. The first six result columns show how our algorithm performs at different depths, the next shows the performance when we undo side effects, and the last two show the performance with various parts of the algorithm disabled. The # columns show the number of expressions searched and times are in seconds.

	Normal algorithm						Side effects	No pruning	Brute force
	Depth 2		Depth 3		Depth 4		Depth 2	Depth 3	Depth 3
	#	Time	#	Time	#	Time	Time	#	#
P 1	34	0.4	611	1.1	19259	9.1	0.4	54911	2034829
P 2	57	0.5	912	1.9	35232	7.7	0.8	10953	847418
P 3	124	0.4	1156	1.2	124991	17.8	1.0	157052	6200476
P 4	7	0.2	36	0.3	552	0.8	0.3	36	219
P 5	22	0.3	234	0.4	2565	1.1	0.3	4692	155774
S 1	8	0.9	223	2.0	1401	3.6	4.3	3775	39439
S 2	12	0.6	275	1.4	2043	3.6	4.5	4457	51079
S 3	70	0.7	814	2.8	6645	6.2	4.9	41359	1867350
S 4	103	1.1	842	3.7	22138	10.6	8.5	504018	61246626
S 5	32	0.8	595	2.8	179956	15.8	6.7	24409	272025
R 1	22	0.2	98	0.3	846	0.6	0.6	98	33112
R 2	12	0.2	137	0.3	1090	0.5	0.3	137	1004
R 3	8	0.1	13	0.1	51	0.2	0.5	13	19
R 4	7	0.2	19	0.2	80	0.3	0.3	19	33
R 5	24	0.3	229	0.3	1761	1.9	0.4	609	115410
Avg	36.1	0.5	412.9	1.3	26574	5.3	2.3	53769.2	4857654.2
Med	22	0.4	234	1.1	2043	3.6	0.6	4457	115410

Thus 94% of calls in practice contain two or fewer arguments and hence little combinatorial explosion, and so biasing our search to avoid the rest seems beneficial. We additionally note that just because a method has multiple arguments does not mean that there will be many ways to call it, as arguments are often of types that have few values in practice, such as singletons. Our heuristic will not avoid such calls.

5.2 User Studies

We conducted two user studies to evaluate the usability of `CodeHint` and learn how developers use it. The first study focused on constrained single line code edits and the second focused on larger open-ended tasks and used an improved version of `CodeHint`. Observations from these studies generated additional areas for improvement that we implemented in the meantime, including the skeletons of Section 3.2.

5.2.1 Methodology

Study 1: Line-level tasks.

For the first study, we created three scenarios with five sub-tasks each that mimic code completion tasks programmers face in practice. To focus on particular tasks where `CodeHint` could be applied, we provided working wrapper code that participants had to extend. The Parse (or P) scenario manipulated strings and parsed email headers and command-line arguments. The Swing (or S) scenario created a small GUI. The RandomWriter (or R) scenario created a Markov model to generate output that looked similar to input text. We chose the first two scenarios to represent common tasks involving APIs and the third as an example application. The first part of Section 2.2 is a slightly modified version of one of the tasks.

At the time of the study, `CodeHint` could solve thirteen of the fifteen tasks; we included the remaining tasks to see how subjects handled cases it could not solve (both could have been found with a slightly higher search depth and can be found by the current version of the tool).

In our within-subjects study, each user received a random assignment of *control*, *experimental*, or *choice* conditions to scenarios. In the control condition, users could not use `CodeHint`, in the experimental condition they were required to use it (although they could write code normally if it failed), and in the choice condition they could decide whether or not to use `CodeHint` for each task. We counterbalanced the order of the experimental and control groups and assigned the choice condition last so participants had experience both using `CodeHint` and writing code normally.

Study 2: Open-ended tasks.

The tasks in the second study were larger – each task required writing three to sixteen lines of code that used complex APIs. Two scenarios had three tasks each, again scaffolded by wrapper code. The Eclipse scenario implemented a simple Eclipse profiler plugin and the Note scenario involved writing a GUI note-taking application that synchronized data to the cloud. Example tasks included finding all the objects in the heap of a program running in Eclipse and adding a menu item that made selected text bold.

Each participant solved one scenario in the control group and the other in the choice group (as defined above). Subjects in the choice group could solve tasks using a combination of `CodeHint` and traditional techniques. Both the scenarios and the groups were ordered randomly. We additionally allowed subjects who could not solve Eclipse tasks in the control group to solve them with `CodeHint`. The tasks were designed to be difficult to solve, so we stopped subjects if they had not completed a task after twenty minutes.

5.2.2 Participants

Nine subjects initially completed the first study; another five were recruited later to complete only the choice con-

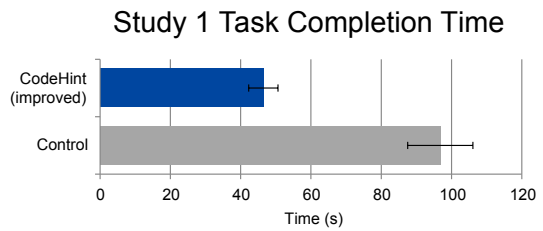

Figure 2: The task completion time of subjects in our first user study. The error bars show the standard error.

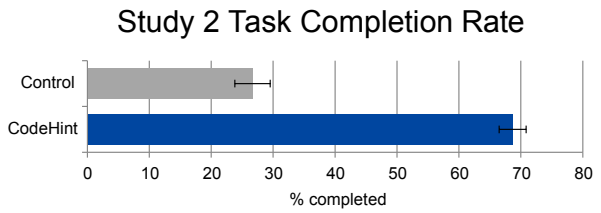

Figure 3: The task completion rate of subjects in our second user study. The error bars show the standard error.

dition of the same study with an improved version of the tool to collect additional data. Twelve were graduate students in Computer Science at UC Berkeley and two were undergraduates.

Another fourteen subjects completed the second study, which used a further improved version of `CodeHint`. Ten were undergraduates and four were graduate students in Computer Science at UC Berkeley.

In both studies, participants practiced on some training tasks first and completed a post-test questionnaire. They were allowed to use a web browser to search for help. None of the subjects had ever used `CodeHint` before, but all were somewhat familiar with both Java and Eclipse.

5.2.3 Measures

We defined the following measures:

- *Task completion time*: Time taken to either complete or abandon a task.
- *Task completion rate*: Percentage of tasks users successfully completed.
- *Code quality*: Number of bugs in participants' task code.
- *Tool choice*: Fraction of tasks in the choice condition for which participants opted to use `CodeHint`.

5.2.4 Results

We first discuss quantitative results followed by qualitative impressions of how our participants used `CodeHint`.

Productivity and preference.

Completion time: In all but one case, participants in the first study completed all tasks, so we focus our analysis on task completion times. On average, subjects using the improved version of `CodeHint` completed tasks in 46 seconds and control participants took 97 seconds (see Figure 2). This difference is significant (two-sample $t(11) = 2.42, p = 0.033$, two-tailed). This suggests that programmers are more productive when using `CodeHint`.

Completion rate: Participants in the second study did not complete many tasks, so we focus on task completion rates. Figure 3 shows the task completion rate for users in our second study with and without `CodeHint`. On average, subjects using `CodeHint` completed 69% of sub-tasks while those not using it completed 27% ($\chi^2(1, N = 14) = 26.06, p < 0.001$), which strongly suggests that programmers complete more difficult API tasks when using `CodeHint`.

Code quality: Participants introduced 11 bugs in 122 tasks solved with `CodeHint` in our first study and 24 bugs in 93 tasks in code written without it (two-sample $t(12) = 2.81, p = 0.015$, two-tailed). This suggests that `CodeHint` improves code quality. To focus on the completion rate, we gave subjects in the second study a comprehensive test suite, so they wrote almost no bugs.

Tool choice: In the choice condition, each user could elect whether or not to use `CodeHint` for each task. On average over both studies subjects used `CodeHint` 71% of the time, suggesting that users found `CodeHint` valuable.

In the questionnaire, participants in both studies rated the overall usefulness of `CodeHint` at an average of 7.7 out of 10 (with a standard deviation of 1.2). All users reported that they would use `CodeHint` for their own development if it were available for their language and editor and had some simple flaws fixed. Six of the subjects asked for the plugin shortly after completing the user studies and installed it.

Qualitative results.

`CodeHint` presented only a small number of candidates to the user after the initial pdspec: in the first study, the average number of candidates across all episodes was 13 and the median was 2, and in the second, the average was 34 and the median was 3. When refining an existing set of candidates, users in both studies provided pdspecs that reduced the number of candidates by 31% on average. However, 47 out of the 89 refinements did not reduce the size of the candidate set at all, mostly because all the candidates were already equivalent on all possible inputs. Ignoring those, the average reduction was 66%.

Choosing a pdspec requires trading the strength of the specification for the ease of encoding it and the cost of evaluating it. To examine this tradeoff, we classified all of the pdspecs used by subjects while completing the user studies and found that in the first study, 53% demonstrated the desired value, 33% gave the desired type, and 14% were arbitrary predicates. In the second study, subjects used only type demonstrations for the Eclipse scenario (as its difficulty lay in finding and using complex types), but for the Note scenario they gave 51% value demonstrations, 38% type demonstrations, and 10% arbitrary predicates. This shows that users can get benefits with `CodeHint` even while demonstrating simple pdspecs but that the ability to provide more expressive pdspecs is sometimes valuable.

6. RELATED WORK

Programming by demonstration, also called programming by example, is a popular area of research [10, 7, 19, 16], much of which deals with synthesizing macros [17] and scripts [18]. Unfortunately, these techniques have not been widely adopted, in part because users have difficulty understanding and cor-

recting the learned generalizations [15]. We attempted to avoid these problems by integrating `CodeHint` into the user's workflow and encoding its state directly in the code.

There has been much work on live programming and how it can benefit programmers [38, 3, 34]. This work inspired us to design our methodology to support concrete reasoning.

Program synthesis has had numerous successes at synthesizing code in small well-defined domains such as bitvector logic [13] and data structures [30, 11] as well as somewhat more general classes of programs [31, 14, 32]. As they are backed by decision procedures and SMT solvers, these techniques are very efficient in certain domains that have been fully modeled, while `CodeHint` is not domain-specific and works without any modeling (e.g., it can read from the file system). We thus view these as complementary techniques.

There has been much research on helping programmers explore new APIs by mining existing code to find snippets that are used in practice [20, 12, 27, 35, 9]. Unlike such systems, by evaluating code at runtime, we can differentiate between different values of the same type, downcast precisely, and use more general specifications, as in Section 2.

Test generation techniques [37, 39, 28] generate inputs to explore branches within code (which is equivalent to satisfying boolean specifications) but they do not always generate the code to construct those inputs. Seeker [36] synthesizes code fragments with a combination of dynamic and static analysis. Its reliance upon static analysis means it cannot generate certain code fragments that `CodeHint` can but allows it to be more efficient in many cases. We would like to integrate similar techniques into `CodeHint`.

Some existing code search tools [25, 4, 21, 1] allow more general specifications such as natural-language queries or testcases, but they lack the full power of our pdspecs and our ability to use dynamic information. Our skeletons are similar to the partial expressions of [24] and the holes of [31].

The Smalltalk method finder [2] allows programmers to specify concrete arguments and the desired result and then evaluates all methods of the given receiver with the given arguments to see which return that result. The algorithm is thus very simplistic, but it allows giving multiple demonstrations, similar to our refinement methodology.

In summary, the key benefits of our approach are the precision enabled by our dynamic nature, the generality of our pdspecs, and the fact that we can handle the full Java language without any modeling. The main disadvantages are that we are less efficient and provide fewer correctness guarantees than some existing techniques.

The way our algorithm iteratively generates expressions of larger depth is similar to the approach used by CHESS [22], as we both search the space in a way that prioritizes elements that are more likely to occur in practice. Our representation of the search space is similar to version space algebra [17, 16] but we can more efficiently eliminate redundant elements.

Our equivalence classes are similar to ideas from Daikon [8], Randoop [23], and model finding [40]. Unlike Randoop, we enumerate all solutions, guided by our probabilistic model toward more likely expressions, up to some bound instead of randomly searching the space. Our techniques are also more powerful than these, and they handle primitives, objects, and arrays. We also apply the algorithms in the new domain of program synthesis, using them to gain the ability to synthesize real Java code. Most importantly, unlike these approaches we are sound in the presence of side effects.

Our probabilistic model is currently simpler than those used in other work, e.g., [5]. As we have found our model to be very useful in practice, we would like to improve it.

Many existing techniques for undoing side effects in Java focus on transactions [26] or sandboxing [29, 33]. These approaches can be effective but often require modifying the client code or the runtime system. In contrast, our watchpoint-based algorithms are slower but require only a standard JVM. Research on speeding up watchpoints [41] could improve the efficiency of our approach, or we could use a more efficient technique. Some researchers have used standard JVM techniques to reset only static state [6]; our approach in some ways generalizes this work to reset all state.

7. FUTURE WORK

We plan to continue to improve our algorithms. One promising technique is to integrate a symbolic solver into our approach, as both have complementary strengths; our algorithm can handle arbitrary Java method calls while symbolic techniques can quickly explore a very large search space in certain domains. We would also like to continue to improve our probabilistic model of what code looks like in practice.

Our approach seems well suited to dynamic languages, as unlike static tools, we can leverage their runtime information. We would thus like to develop an implementation of `CodeHint` that works for a language such as JavaScript.[4]

8. CONCLUSION

We have presented a novel methodology that helps programmers write difficult statements. This methodology is dynamic (evaluating code and runtime and letting users inspect the results), easy-to-use (accepting a wide range of specifications), and interactive (helping users gain information about the missing code). Our algorithms are efficient and let us synthesize code that uses real-world Java features such as native calls and reflection. We have run two user studies that shows that our tool `CodeHint` significantly improves programmer productivity.

9. ACKNOWLEDGMENTS

We are very grateful to all of the participants in our user studies for their time and suggestions. We would also like to thank Leo Meyerovich for his detailed feedback. This work is supported in part by NSF Grants CCF-1018729, CCF-1018730, CCF-1139138, CCF-1017810, and CCF-0916351 and gifts from Samsung and Mozilla. The last two authors are supported in part by Sloan Foundation Fellowships.

10. REFERENCES

[1] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. OOPSLA '06, pages 681–682, 2006.

[2] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, et al. *Pharo by example*. 2009.

[3] M. M. Burnett, J. W. Atwood Jr, and Z. T. Welch. Implementing level 4 liveness in declarative visual programming languages. VL '98, pages 126–, 1998.

[4] A prototype implementation of `CodeHint` for JavaScript is available at `https://github.com/jgalenson/codehint.js`.

[4] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In FASE '09, pages 385–400, 2009.

[5] A. Cozzie and S. T. King. Macho: Writing programs with natural language and examples. Technical report, University of Illinois at Urbana-Champaign, 2012.

[6] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, Sept. 2004.

[7] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, 1993.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. ICSE '99, pages 213–224, 1999.

[9] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *Computer Aided Verification*, pages 418–423. 2011.

[10] D. C. Halbert. *Programming by example*. PhD thesis, 1984.

[11] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. PLDI '11, pages 38–49, 2011.

[12] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In ICSE '05, pages 117–125, 2005.

[13] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In ICSE '10, pages 215–224, 2010.

[14] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In PLDI '10, pages 316–329, 2010.

[15] T. Lau. Why pbd systems fail: Lessons learned for usable ai. In *Computer Human Interaction*, 2008.

[16] T. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In K-CAP '03, pages 36–43, 2003.

[17] T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In ICML '00, pages 527–534, 2000.

[18] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. CHI '08, pages 1719–1728, 2008.

[19] H. Lieberman, editor. *Your wish is my command: programming by example*. Morgan Kaufmann Publishers Inc., 2001.

[20] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In PLDI '05, pages 48–61, 2005.

[21] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. ICSE '11, pages 111–120, 2011.

[22] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. PLDI '07, pages 446–455, 2007.

[23] C. Pacheco. *Directed random testing*. PhD thesis, 2009.

[24] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. PLDI '12, pages 275–286, 2012.

[25] S. P. Reiss. Semantics-based code search. ICSE '09, pages 243–253, 2009.

[26] A. Rudys and D. S. Wallach. Transactional rollback for language-based systems. DSN '02, pages 439–448, 2002.

[27] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In OOPSLA '06, pages 413–430, 2006.

[28] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. ESEC/FSE-13, pages 263–272, 2005.

[29] J. Siefers, G. Tan, and G. Morrisett. Robusta: taming the native beast of the jvm. CCS '10, pages 201–211, 2010.

[30] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. ESEC/FSE '11, pages 289–299, 2011.

[31] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In PLDI '05, pages 281–294, 2005.

[32] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In POPL '10, pages 313–326, 2010.

[33] M. Sun and G. Tan. Jvm-portable sandboxing of java's native libraries. In S. Foresti, M. Yung, and F. Martinelli, editors, *Computer Security - ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 842–858. 2012.

[34] S. L. Tanimoto. Viva: A visual language for image processing. *J. Vis. Lang. Comput.*, 1(2):127–139, June 1990.

[35] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In ASE '07, pages 204–213, 2007.

[36] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. OOPSLA '11, pages 189–206, 2011.

[37] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. TAP'08, pages 134–153, 2008.

[38] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? CHI '97, pages 258–265, 1997.

[39] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In N. Halbwachs and L. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 365–381. 2005.

[40] J. Zhang and H. Zhang. Sem: a system for enumerating models. IJCAI'95, pages 298–303, 1995.

[41] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: efficient debugging using dynamic instrumentation. CC'08/ETAPS'08, pages 147–162, 2008.