Abstract
Most programs contain conceptually simple statements that are difficult to write. We propose a new methodology that allows programmers to give partial dynamic specifications of the results of statements in a particular program state, and which uses them to synthesize candidate statements. Building on previous work, these specifications can consist of information about the type or value of a desired expression or can be arbitrary predicates relating the input and output states. We implemented our methodology in a tool named CodeHint and ran a user study where we found that programmers used it when given the choice.

1. Introduction
Some statements are difficult to write, whether due to tricky logic, common coding errors (such as off-by-one errors), or a new or unfamiliar API. In these cases, it is often much easier to describe for a given input the result of the desired statement, either by directly stating the full effect or by giving some partial specification of the desired result.

In addition, human beings are better at working with concrete states than with abstract symbols. When explaining a new algorithm or working through the implementation of a new feature, it is common to walk through its execution on a concrete input. It is our belief that it is often easier for developers to reason about a particular concrete state of the program, as seen through the lens of a debugger or similar tool, than to reason abstractly about all possible combinations of inputs at once.

For example, a programmer writing a graphical interface might want to write a statement to find which element in a graphical list he clicked or find in which window that list is contained. Although these tasks are conceptually simple, finding the desired statements normally will be difficult if he is unfamiliar with the library he is using.

Previous work on API exploration [9, 16, 17, 20] has allowed users to find sequences of statements that transform input types into output types, but such queries are not always strong enough, such as when trying to find integers. Users of program synthesis tools [7, 11, 18] can use code generated from strong correctness conditions, but these correctness conditions are often hard to write. Programming by Demonstration approaches [5, 8, 15] allow users to demonstrate the results of statements (e.g., by giving the values that are being assigned) and find the statements that yield those results, but it is not always easy (or even possible) to demonstrate such a value.

The core contribution of our paper is a methodology that allows users to view the concrete state of a partial program and use it to provide a partial specification of the desired behavior in that state. This partial dynamic specification, which we call a pdspec, is used to synthesize a set of candidate statements that is shown to the user. If she desires, she can give further pdspecs in different program states to refine this set of candidates. These pdspecs are dynamic, and so can take advantage of the concrete state of the program, and may be as strong or weak as the user desires.

The type specifications, correctness conditions, and value demonstrations described above are all examples of pdspecs. In the example mentioned above where a programmer wants to find on which element of a graphical list he clicked, he can easily say that he perhaps clicked on the first element, then the second, and then clicked on nothing (which he might want to return a negative number). To find the window that contains the list, he can say that he wants a result of a certain type that is not null.

We specifically target program fragments that are not intrinsically complicated but tend to be difficult to write in
practice. Tasks such as parsing strings (perhaps command-line arguments or network packets), constructing a GUI, or validating user-provided input are classic examples of such program fragments. Additionally, we have found that our methodology works well for exploring new APIs. It can also be useful as an advanced form of auto-complete.

After developing an implementation of our methodology, we ran a user study to see how programmers like it. They were excited about our methodology and enjoyed using the tool; some of them installed it on their development machines shortly after the study ended.

Before defining our approach in detail, we walk through two examples of its use in Section 2. These should show how it works at a high level and display some of its benefits.

We then present a formalization of our methodology in Section 3. First, we formally define the notion of a partial dynamic specification (pdspec), which relates the desired output state and current input state. Then we state how pdspecs are used to synthesize candidate statements and give the soundness and completeness properties for our methodology. We discuss some properties of these pdspecs in Section 3.1, which we briefly use in Section 3.2 to relate our approach to previous work.

We have developed an implementation of our methodology, which we call CodeHint, as a plugin for the Eclipse IDE for Java, and in Section 4 we describe its design and implementation. The design was influenced by our desire to build a system that would be easy for developers to adopt, and we describe in Section 4.2 how this principle affected our implementation.

From a user study described in Section 5, we see that the subjects like using CodeHint and choose to use it when given the choice. We take advantage of this simulation of real-word use to present some benchmarks for how the tool performs in practice.

After a more detailed discussion of related work in Section 6, we present some ideas for how to improve our methodology and implementation in Section 7. Many of these ideas were inspired by seeing subjects in our user study use CodeHint.

We make the following contributions.

• We introduce a new methodology that allows programmers to give partial dynamic specifications, which we call pdspecs, and uses them to find the desired code.

• We develop an implementation of our approach that works for Java and is available as an Eclipse plugin. We evaluate CodeHint on real programmers and find that they gave it high ratings and chose to use it when given the chance, showing that they like our methodology.

2. Overview

We now present two examples of how to use our methodology to help readers understand the intended workflow and to motivate our work. Both of these examples are problems that subjects in our user study struggled to solve without our approach. Neither is intrinsically difficult for someone intimately familiar with the libraries and problem domain, but both are surprisingly difficult even for experienced programmers unfamiliar with them.

Example 1  A programmer is presented the code below and asked to convert the array of arguments into a List to be passed to a function helper, which takes an argument of type List<String> and an argument of type String[].

```java
public static void main(String[] args) {
    String[] validArgs = {"--baz", "--bar"};
    List<String> argList = null;
    // Convert args into a list.
    helper(argList, validArgs);
}
```

He could use his editor’s auto-complete features to try certain obvious approaches, such as creating a new list or calling a method on the array, but none of these obvious approaches would work in this case as the correct solution involves a separate helper class. A search through the API documentation will likely prove equally fruitless as he does not know which class to examine.

Using CodeHint, he will run this code with real inputs and then demonstrate a property that should hold for the desired list after a new assignment statement is added. In this case, assume he invokes the program with the arguments {"--foo", "--bar", "--arg=5", "--baz"}.

Using this concrete example, he will navigate in the debugger to line 4 and provide the following pdspec to specify the type of his desired statement:

```java
argList instanceof List
```

From this, CodeHint presents him with a list of all the ways to generate a list in the current context and embeds them directly in the source file. The programmer will see:

```java
argList = CodeHint.choose(Arrays.asList(args), Arrays.asList(validArgs));
```

At this point, he navigates back to the same line and gives the pdspec argList.get(0).equals("--foo") to eliminate the incorrect choice. He could also have simply selected the correct statement if he had recognized it. In either case, he has successfully completing the task. The final statement is:

```java
argList = Arrays.asList(args);
```

The second step of this example shows the power of our approach over API exploration tools such as Prospector [16]. Users of our approach can leverage information about the concrete program states of an execution to help eliminate undesired candidates.

Example 2  A programmer is writing GUI code using the Java Swing toolkit. A common task when writing such code
is to use a JTree to display data and write a mouse listener to detect and handle clicks. The fragment below is a standard way to write this code.

```java
final JTree tree = ...
tree.addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    int clickedRow = 0;
    // Figure out which row the user clicked.
    handleClick(clickedRow);
  }
 .handleClick(int clickedRow)
});
```

A programmer might be unsure how to find the clicked row. She can, however, easily demonstrate the expected value. She will set a breakpoint on line 7 and then execute her code and click on the top row. She knows the index of the row she clicked is 0. Once she encounters her breakpoint, she provides this value to CodeHint, which will synthesize the following statement:

```java
clickedRow = CodeHint.choose(0, y/x, tree.getMinSelectionRow(), tree.getRowForLocation(x,0), tree.getRowForLocation(x,y), ...);
```

Overall, there will be approximately 40 expressions; the exact number will depend on the details of the actual click. To reduce the number of candidates, she will repeat the process but click on a different row. After she does so, CodeHint will modify the statement to the following:

```java
clickedRow = CodeHint.choose(tree.getMinSelectionRow(), tree.getRowForLocation(x,y), ...);
```

There will now be only approximately seven expressions. Finally, she provides a negative example by clicking outside the bounds. This time she does not know the exact value she will get, but she does know that negative values are often used as error codes. As such, she gives the pdspec `clickedRow < 0`. From this, CodeHint eliminates all but one candidate and finds the correct code.

```java
clickedRow = tree.getRowForLocation(x,y);
```

This example demonstrates how we build on previous programming by demonstration work by adding the ability to specific arbitrary pdspecs. The true power of our methodology is that programmers can mix and match value demonstrations, type specifications, and arbitrary pdspecs as desired.

3. Approach

Our methodology allows users to describe their intentions, which are used to synthesize the statement they desire. To allow concrete reasoning by the user, all of the specifications given by the user are in the context of a particular program state. We now formalize these notions.

We define $\sigma \in \Sigma$ to be a program state, which maps variables to their values, and $s \in S$ to be a statement in the programming language. The function $\text{exec} \in S \times \Sigma \rightarrow \Sigma$ executes a statement from a given initial state and returns the resulting output state. As a running example, given an initial state $\sigma = \{ x \mapsto 42 \}$ and the statement $x = x + 1$, which we will call $s^*$, we have that $\text{exec}(s^*, \sigma) = \{ x \mapsto 43 \}$.

We define $S^*$ as the set of statements that correctly model the user’s desired behavior at a particular program location. These statements need only be equivalent in states in which they can actually be executed. We thus formally define the notion of valid states, which are those in which the user implicitly expects statements in $S^*$ can be executed. We refer to the set of all valid states as $\Sigma_{S^*}$. Formally, any two statements in $S^*$ are equivalent under all valid initial states:

$$\forall s, s' \in S^*, \forall \sigma \in \Sigma_{S^*}. \text{exec}(s, \sigma) = \text{exec}(s', \sigma)$$

Continuing our running example, $x = x + 2 - 1$ is equivalent to $s^*$. In addition, if $s^*$ can never be executed when $x$ is negative, then $\text{if } (x < 0) x = 5 \text{ else } x = x + 1$ is also equivalent to $s^*$.

The user provides a partial specification of the desired statement (i.e., any element of $S^*$) in the current program state, which we henceforth call a pdspec. As we will discuss in Section 3.1, a pdspec does not necessarily need to hold for all inputs or even completely specify the behavior of the desired statement in the current state. Formally, a pdspec is a logical predicate $\phi \in \Sigma \times \Sigma \rightarrow \{ \text{true}, \text{false} \}$ where $\phi(\sigma, \sigma')$ checks whether $\sigma'$ is a desired output state given the input state $\sigma$.

As a notational convenience, for a pdspec $\phi(\sigma, \sigma')$, we refer to variables in the input state $\sigma$ using their names and variables in the output state $\sigma'$ using their primed names. All variables not given in a pdspec must be side-effect free. Some pdspecs for our running example include $x' = 43$, $x' > 0$, $x' \mod 2 = 1$, $x' = x + 1$, and $\text{true}$.

A user might mistakenly provide an invalid pdspec that does not hold for the statement $s^*$, such as $x' = x + 2$. We say that a pdspec $\phi$ is valid if it is a correct description of the user’s desired behavior (i.e., that of the statements in $S^*$) in a given valid state. A pair $(\sigma, \phi)$ is valid, denoted $V(\sigma, \phi)$, if both $\sigma$ and $\phi$ are themselves valid, i.e.,

$$V(\sigma, \phi) = \sigma \in \Sigma_{S^*} \wedge \forall s \in S^*. \phi(s, \text{exec}(s, \sigma)).$$

To find a statement in $S^*$, which represents the user’s desired behavior, we guide the user through presenting a sequence of initial state and pdspec pairs. We then want to

---

1This is a conservative definition. Technically, their outputs need only match for the parts that affect future execution. We avoid this definition as it would be hard to reason about or compute.
present the user with all of the statements that satisfy those specifications. As the number of statements in a programming language is likely infinite, we are forced to restrict our search to statements in some finite \( F \subseteq S \). This \( F \) is an abstraction of the search space explored by our implementation.

Given a sequence of pairs of input states and their corresponding pdspecs \((\sigma_0, \phi_0), (\sigma_1, \phi_1), \ldots, (\sigma_n, \phi_n)\), we find a set of candidate statements \( C_n \), which is defined as

\[
C_n = \left\{ s \in F : \bigwedge_{0 \leq i \leq n} \phi_i(\sigma_i, \text{exec}(s, \sigma_i)) \right\}.
\]

Since \( C_{n+1} \subseteq C_n \), the user can refine the set of candidate statements by providing an addition pdspec in some initial state. We call this process refinement. At any point, if the user notices the statement he wants, he can simply use it and end the refinement process.

The new input states used for refinement can come from different test cases. In Section 4, we describe how in practice we can sometimes continue with the current execution.

In our running example, given the initial state \( \{ x \mapsto 42 \} \) and the pdspec \( x' > 0 \), we present the user with the following set of candidate statements:

\[
C_0 = \{ x = x + 1, x = x + 2, x = 7, x = 8, \ldots \}
\]

Given another initial state \( \{ x \mapsto 6 \} \) and the pdspec \( x' = 7 \), we present the user with the subset:

\[
C_1 = \{ x = x + 1, x = 7, \ldots \}
\]

If a user gives an invalid pdspec that does not hold for all of the statements in \( S^* \), then none of those statements will be in the corresponding \( C_n \). However, if all of his pdspecs are valid, then \( C_n \) will include all statements in \( S^* \) that are also in \( F \). Formally, we define our version of qualified completeness as

\[
\forall 0 \leq i \leq n. V(\sigma_i, \phi_i) \Rightarrow S^* \cap F \subseteq C_n.
\]

In general, \( C_n \) might contain statements that do not have the behavior desired by the user on all states. The user can remove the statements from subsequent \( C_n \) by providing well-chosen input states or pdspecs. Formally our version of soundness is

\[
\forall s \notin S^*. \exists 0 \leq i \leq n. \neg \phi_i(\sigma_i, \text{exec}(s, \sigma_i)) \Rightarrow C_n \subseteq S^*.
\]

We note that our formalism could be extended to search for sequences of statements by considering a block of statements as a single statement. It can also extend to search expressions by generating an assignment to a fresh temporary variable.

### 3.1 Classification of pdspecs

As we saw in the previous section, a valid pdspec only needs to hold for the desired statement in the current state. It does not necessarily need to reject all undesired statements or work on all states. We now define and discuss these properties in more detail.

As logical formulae, it makes sense to talk about the strength of a pdspec using logical implication. A pdspec \( \phi \) is clearly stronger than \( \phi' \) if \( \phi \Rightarrow \phi' \).

However, our pdspecs are given in the context of a program state and are used to refine the \( C_n \) generated by previous states and pdspecs. It thus seems natural to compare the strength of pdspecs by the number of statements in the current set of candidates that satisfy them in the given state.

Formally, we say that a pdspec \( \phi \) is stronger than \( \phi' \) in a state \( \sigma \) if, given a set of candidates \( C_n \),

\[
|\{ s \in C_n : \phi(\sigma, \text{exec}(s, \sigma)) \}| \leq |\{ s \in C_n : \phi'(\sigma, \text{exec}(s, \sigma)) \}|.
\]

If the user gives a stronger pdspec, the corresponding set of candidates \( C_n \) will be smaller, which means that the user will need to give fewer pdspecs during refinement before finding the desired statement. However, even weak pdspecs can be helpful as \( n \) increases.

Some pdspecs, such as \( x' = x + 1 \) above, are valid for all valid states, while others, such as \( x' = 43 \), are only valid for certain input states. We can discuss the context-dependence of a pdspec, which describes how much the pdspec depends on the corresponding state. Formally, we say that a valid pdspec \( \phi \) is more context-dependent than a valid pdspec \( \phi' \) if it holds in fewer valid states, or

\[
|\{ \sigma \in S^* : V(\sigma, \phi) \}| \leq |\{ \sigma \in S^* : V(\sigma, \phi') \}|.
\]

We can now discuss a couple of useful special cases of pdspecs in the case where the user is trying to find a single assignment statement that has no additional side effects.

Users often know the value that should be assigned to a variable, in which case they can demonstrate that value directly (e.g., \( x' = 42 \)). These value demonstrations are stronger than all other valid pdspecs and are very context-dependent.

Users can provide the dynamic type of the variable being assigned (e.g., \( x' \text{ instanceof } \square \)). In a statically-typed language, there is an implicit pdspec associated with each assignment. The user can either use this very weak but entirely context-independent pdspec or provide a stronger check for some subtype.

In addition, a classical full correctness condition is stronger and more context-independent than all other valid pdspecs.

We show this space of pdspecs graphically in Figure 1, with the special cases mentioned above in their respective locations.

### 3.2 Relation to Previous Work

The classifications of pdspecs we developed in Section 3.1 can help us explain how our approach relates to previous work.
Work on Programming by Demonstration \cite{5, 8, 13, 15} synthesizes statements from programmer-provided demonstration of their results. These demonstrations correspond to our value demonstrations, and so Programming by Demonstration fits in the upper-left part of Figure 1.

Programmers using tools from program synthesis \cite{7, 11, 18} can give functional specifications of their desired program and get back code that was synthesized using techniques such as SMT solvers. As these specifications usually need to be quite strong for the synthesis to be successful, these approaches fall in the lower-left part of Figure 1. These techniques usually give their specifications statically; we note that we could allow programmers to give completely context-independent pdspecs statically as well.

As we will describe in Section 5.2, in practice users seem to avoid giving pdspecs that resemble full correctness conditions, perhaps because they are often difficult to write and our implementation cannot currently take advantage of their strength.

Research on exploring new APIs \cite{9, 16, 17, 20} generally finds code given queries that specify desired input and output types. These queries are fairly weak and often context-independent, placing this body of work in the bottom-right part of Figure 1.

Our approach allows pdspecs that fall anywhere in the spectrum depicted in Figure 1, although as we describe in Section 4, we gain this generality at the cost of efficiency.

As we will describe in Section 5.2, in practice users seem to avoid giving pdspecs that resemble full correctness conditions, perhaps because they are often difficult to write and our implementation cannot currently take advantage of their strength.

Research on exploring new APIs \cite{9, 16, 17, 20} generally finds code given queries that specify desired input and output types. These queries are fairly weak and often context-independent, placing this body of work in the bottom-right part of Figure 1.

Our approach allows pdspecs that fall anywhere in the spectrum depicted in Figure 1, although as we describe in Section 4, we gain this generality at the cost of efficiency.

We believe that our work integrates very nicely with a Test Driven Development \cite{2} methodology where tests are written before the corresponding code is actually implemented.

4. Implementation

We have developed an implementation of our approach, which we call CodeHint, as a plugin for the Eclipse IDE \cite{1} for Java \cite{6}.

```java
public class CodeHint {
    public static <T> T choose(T first, T... rest) {
        for (T choice : rest)
            assert first == null ? choice == null : first.equals(choice);
        return first;
    }

    public static <T> T chosen(T choice) {
        return choice;
    }
}
```

We currently generate assignment statements rather than the arbitrary statements we have discussed so far. Similarly, the pdspecs provided by the user are expressed as predicates over the input state and the value being assigned rather than the entire output state. It would be straightforward to extend this to support statements with side effects, and we plan to do this in the near future.

To use CodeHint, a programmer must start a debug session and then navigate to the program location and state in which she wishes to insert code. She selects the variable she wishes to change in the Eclipse UI and enters a pdspec through a textbox. There are also shortcuts to demonstrate a value (by giving an expression and evaluating it) and the desired dynamic type (by giving the name of a valid type).

We then synthesize potential expressions, wrap them in a call to a special choose method, and assign the result to the selected variable. This is a source code representation of the set of statements $C_n$ from Section 3. When $C_n$ contains only a single element, we replace the choose call with a call to a chosen method that has identical semantics as an indicator to the user that the process is complete. Figure 2 shows our Java implementation of these two methods (with a few minor details omitted).

After providing a pdspec and being shown the set of candidate expressions, a user of CodeHint can end the debug session and continue with a new input (perhaps from a different test case). Alternatively, if the user provides a pdspec for which all candidate statements generate the same concrete output state, we can continue executing the program. A value demonstration fulfills this property, but other pdspecs might as well. This allows us to work well for statements that are contained in loops or otherwise might be executed multiple times.

Repeat encounters with an inserted choose statement are handled by inserting a breakpoint at the line and registering a
listener that activates when that breakpoint is hit. This handler overrides the default implementation of choose given in Figure 2 and is responsible for prompting for additional pdspecs when required.

If all the candidate statements give the same value, that value is used as the result of the expression and we do not prompt for another pdspec. If at least one statement in $S^*$ is in $F$ and the user has given only valid pdspecs, this is safe to do. Otherwise, we may diverge from the correct execution. In practice, we have not seen this to be a problem. An alternate implementation would be to prompt the user for a pdspec.

If we prompt the user for a pdspec, we refine the candidate expressions and remove those that do not satisfy the new pdspec from the code. There are two cases in which there may be no remaining statements. First, the user could have provided an invalid pdspec at some point. As mentioned in Section 3, we can provide no completeness guarantee without valid pdspecs. Second, none of the statements in $S^*$ may exist in our search space $F$. We cannot distinguish between these two cases and inform the user that an error has occurred.

Since the candidate expressions are inserted directly into the code, the developer can directly modify $C_\nu$, at any time. For example, he can simply select the desired expression and delete everything else, or he can remove expressions that he knows are incorrect so they are not considered again. We note that there is a spectrum between editing the code to select the correct expression after the first pdspec and refining the list of candidate expressions until only one remains.

Our simple runtime library provides the choose and chosen methods in an executable form so that the program remains executable at all times and can be used outside of our environment. Since the inserted statements refer to valid methods, they will not break the compilation of the code, and will even execute correctly in certain cases. An interesting side effect of this is that programs that use these constructs can be executed for specific inputs even if they are not fully implemented. In this use the assertion on line 5 of Figure 2 is necessary to ensure correctness. We can provide no correctness guarantee if an invalid state or pdspec was given.

4.1 Statement Generation and Evaluation

We define our $F$ – i.e., all possible statements we can generate – as the set of assignments formed by assigning fixed-depth type-safe expressions from the grammar given in Figure 3 to the variable specified by the user. This grammar contains most Java expressions, including array accesses, field accesses, method calls, and constructor calls. It uses all of the variables visible in the current scope. If the user demonstrated a value, we include it as a possible constant. We include calls to static methods and fields of imported classes. Although we currently only support adding assignment statements, we plan to extend our implementation to support synthesis of arbitrary sub-expressions (rather than entire statements) and some forms of control flow (see Section 7).

Our current implementation generates expressions by recursively expanding this grammar up to a fixed depth. With a depth of one, which we currently use, we can generate expressions like $x+1$ and $\text{foo.bar}(x,y)$ but not $x+y+z$ or $\text{foo.bar}(x+y)$. Once we have generated all the expressions in this bounded space that return a legal type (a subtype of the static type of the variable being assigned), we evaluate them in the context of the current stack frame and keep only the ones that satisfy the user’s pdspec.

We apply a number of optimizations to improve the efficiency of our implementation. We use a variety of simple structural techniques to avoid enumerating obviously equivalent expressions (such as $x+y$ and $y+x$). We avoid generating expressions we know will throw exceptions (e.g., if we know $x$ is $\text{null}$, we will not generate $x.\text{foo}$). Our current implementation scales well when enumerating expressions that do not contain arbitrary method calls. These can run for arbitrary amounts of time and can throw exceptions, which disrupts the batching we perform to reduce evaluation overheads for large numbers of expressions and can force us to evaluate each expression individually. We believe both issues are specific to our current implementation and are investigating more efficient implementations. With such optimizations, we should be able to increase the depth of statements we generate so we can search more statements, al-

---

Figure 3. The grammar of expressions we consider while generating candidate expressions. Expressions are enumerated up to a fixed depth and evaluated to select those that meet the user-provided pdspec.

| (demonstration) | $d$ |
| (type) | $t$ |
| (local/arg) | $v$ |
| (method) | $m$ |
| (field) | $f$ |
| (constant) | $c ::= 0 | 1 | 2 | \text{null}$ |
| (infix op) | $op ::= + | - | * | / | && | || | == | != | < | <= | > | >=$ |
| (expression) | $e ::= d | c | \text{this} | v | e \ op e | -e | !e | e[e] | e.f | t.f | e.m(e,e,...) | t.m(e,e,...) | new t((e,e,...)$ |
though as shown in Section 5.2, increasing the depth increases the search space by many orders of magnitude.

In our current implementation, we have limited support for candidate statements with side effects (other than assignment statements). We do blacklist certain methods known to have externally visible side effects (e.g., deleting a file), but our current implementation does not sandbox the expressions that are evaluated and as a result can reach invalid states that are not reachable from any real execution of the program. In practice, this has not been an issue during the development of the tool or the user study. Going forward, we plan to enable sandboxing for expression evaluation and use a conservative purity analysis as an optimization. By definition, a pspec cannot have side effects, although it would be useful to enforce this.

4.2 Usability

One goal of our work was to develop a tool that had the potential to be widely adopted. We are not focused on user adoption per se, but we believe that ignoring practical and usability factors leads to having less impact than might otherwise be achievable. We describe some of the design choices this dictated below, but this desire influenced the rest of our system in subtle ways that are not entirely apparent even to us.

When designing our implementation, we had a choice of a number of different programming languages and editors to target. We ultimately decided to target Java as it is a popular language whose users are open to using new tools. We decided to implement our tool as a plugin for Eclipse because it is a mature platform with many users.

To use our tool, users simply need to install a single plugin into their Eclipse installation. The plugin interoperates with existing Eclipse tools, and users may use it only when they desire. When they choose not to use it, our tool does not interfere with their normal programming.

Our choice of algorithms was affected by this principle. There are many more advanced synthesis techniques we could have used instead of brute force, but our simplicity actually ends up being a strength. Unlike techniques based on solving constraints with SMT solvers, our generator works for arbitrary classes of statements (e.g., it can execute arbitrary Java method calls, including binary libraries). We believe that integrating alternate synthesis techniques into our methodology is a promising avenue for future work.

Prior research [12] has shown that usable synthesis tools must be easy to understand and correct. We thus chose to encode the state of our synthesis algorithm directly in the user’s code, so users can easily understand and edit the results.

Encoding our state directly into the user’s code has the additional benefit of allowing us to ensure that the program is always executable. Thus our approach can be used even on a large project where some developers are using our tool and some are not. The program may well abort for most inputs not seen during synthesis, but it will still compile and run. If a developer uses the existing test suite to aid the synthesis, the program may even be able to pass all the tests despite being incomplete. If the user is only interested in the output with a particular set of inputs – of a one-time script, perhaps – she may never need to find the exact statement.

5. Evaluation

We conducted a user study to evaluate CodeHint. Specifically, we sought to answer the following questions:

Question 1. Does CodeHint help make programmers more productive? How do the time taken and quality of resulting code compare with and without it?

Question 2. Do programmers like using CodeHint? Would they adopt it in practice?

Question 3. What pspecs do users write most often? Is the depth one search space used by CodeHint sufficient for many needs?

We also wanted to see how people other than us use CodeHint to help give us ideas for how to improve it and discover other potential application areas. We describe below some observations from watching users, and describe how they influenced some of our future work in Section 7.

5.1 Methodology

We created 15 tasks, organized into three example scenarios, each of which represents code real users might want to write. The Parse example manipulated strings and parsed email headers and command-line arguments. The Swing example initialized a small GUI using the Swing toolkit. The RandomWriter (or RW) scenario created a Markov model based on input text and used it to generate text that looked similar to the original. The examples in Section 2 both came from these scenarios.

Each scenario consisted of mostly-complete wrapper code with five tasks left unwritten. We decided to present users with wrapper code rather than making them write all the code from scratch to reduce the amount of time spent figuring out the algorithms. This way, all the time the users spent on a scenario was directly related to the tasks they were trying to solve. We picked thirteen of the tasks so that CodeHint was able to solve them; the remaining two were purposefully included to see how subjects handled cases it could not solve.

Nine subjects completed our user study. All were undergraduates or graduate students in Computer Science at UC Berkeley. None had ever used CodeHint before, but all were to various degrees familiar with both Java and Eclipse.

5 This was directly inspired by an assignment given in the Stanford introductory programming courses.

6 Both could have been addressed with a slightly higher search depth.
Each subject was initially walked through an introductory scenario that presented simple tasks to solve. This introduction showed how to use CodeHint to give pdspecs, suggest candidate expressions, and refine the list of expressions. It also showed users how to use basic Eclipse features such as auto-complete. The subjects solved each introductory task themselves, so by the end they had some experience using both Eclipse and CodeHint. This introduction took approximately 20 minutes to complete.

All subjects were assigned to solve all three scenarios in a random order. For each scenario, a particular user was assigned to be in exactly one of the control, experimental, or choice groups. Every user was part of each group for exactly one scenario. The choice scenario was always last, but the order of the experimental and control was random. After completing all three scenarios, participants filled out a short questionnaire.

The control group was told to write code normally without using CodeHint. The experimental group was required to attempt to use CodeHint to solve each task, although they could write code normally if it failed. Users in the choice group could decide whether or not to use CodeHint for each task.

Subjects were allowed to use a web browser to look through the Java API or to search the Internet for help. All user studies were performed on an Intel Core 2 Duo machine with two 2.5 GHz processors and 3 GB RAM. The machine was configured to record a screencast of the users’ actions.

5.2 Results and Discussion

**Productivity and Quality** Overall, our results show that developers using CodeHint write code slightly more slowly but with fewer bugs than the same developers using their normal development style.

Users in the experimental group, who were forced to always use CodeHint, took approximately 40% longer to complete each scenario than those in the control group. Users in the choice group took approximately 25% longer than those in the control group and 10% less time than those in the experimental group. We do not present a breakdown of information about how long it took the subjects to complete each task due to noise in the data. For example, users often interspersed attempts to solve the tasks with suggestions about how to improve the tool. We considered this informal feedback to be more important than precise timing information.

We believe most of the difference in time can be attributed to the fact that our users were unfamiliar with the tool. Other contributing factors were:

- Many users had difficulty discovering good pdspecs to give. Some users spent minutes trying to specify complete correctness conditions that were difficult (or even impossible) to write directly instead of a much simpler and weaker pdspec. We believe that with more training and practice, these users would become much more efficient at using CodeHint.
- Our implementation is currently unpolished, so launching and using it take extra time. For example, users must set a breakpoint and start the debugger each time they want to use CodeHint and cannot use auto-complete when entering pdspecs. This overhead will be reduced in future versions.
- A few users struggled to handle the tasks that CodeHint could not solve. As part of the user study, we deliberately included two cases our implementation could not synthesize. Eventually, users who encountered such a task solved it using traditional approaches and moved on.
- Some users experimented with CodeHint, exploring multiple ways to use it to solve certain problems.

Figure 4 shows the number of bugs written by subjects when using CodeHint and when writing code normally. There were ten bugs in code written with CodeHint and thirteen in code written without it, which suggests that it has a small but positive effect on the quality of the generated code. Unfortunately, the difference is not statistically significant.

When watching the subjects use CodeHint, we noticed some interesting trends. Some users gave a pdspec and then selected an expression that looked good without examining it closely. These expressions were sometimes subtly wrong (e.g., they contained a call to the correct method but with the wrong argument), and so by not carefully examining the generated expressions, the users created a bug.

![Figure 4](image-url)
Table 1. Some measures of the usefulness of CodeHint.

The second column shows the number of tasks (out of five) that subjects chose to solve with CodeHint when given the option of whether or not to use it. The last three columns show answers users gave on a questionnaire that asked them to evaluate the tool overall, the refinement methodology, and API exploration as a use case (all on a scale from one to ten with ten being the highest).

<table>
<thead>
<tr>
<th>User</th>
<th># solved with tool</th>
<th>Overall</th>
<th>Refine</th>
<th>Explore APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>N/A</td>
<td>10</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>User 2</td>
<td>3</td>
<td>6</td>
<td>N/A</td>
<td>8</td>
</tr>
<tr>
<td>User 3</td>
<td>4</td>
<td>7</td>
<td>N/A</td>
<td>9</td>
</tr>
<tr>
<td>User 4</td>
<td>4</td>
<td>8</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>User 5</td>
<td>4</td>
<td>8</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>User 6</td>
<td>2</td>
<td>7</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>User 7</td>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>User 8</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>User 9</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Average</td>
<td>3.3</td>
<td>7.6</td>
<td>8.8</td>
<td>8.4</td>
</tr>
<tr>
<td>Std. dev</td>
<td>1.3</td>
<td>1.1</td>
<td>1.0</td>
<td>1.3</td>
</tr>
</tbody>
</table>

On the other hand, when users did examine the entire list of candidate expressions, it sometimes helped them find a bug in code they had already written. For example, by considering the difference between two similar expressions (e.g., calls to the same method with different arguments), they recognized that they had written a bug in similar code earlier.

**User Perception**  In the questionnaire they completed at the end of the study, subjects described how useful they found various aspects of CodeHint on a scale from one to ten (with ten being the highest). The results, which are shown in the last three columns of Table 1, show that subjects like to use CodeHint. When describing the tool in conversation, users stated that “[Using the tool] seems natural” and was “way better” for some tasks than the traditional approach.

One subject, when in the control group, was unable to solve a task that involved using the Swing API in the allotted time. He then used CodeHint to find the correct expression in under a minute. This shows that CodeHint can be very helpful for certain expressions.

When asked to rate particular aspects of the tool, users thought it was especially good at API exploration, giving it an average rating of 8.4. Some representative quotes were, “when exploring a new API, it tells me where to look,” and “[I] don’t have to leave [my] work environment to search for APIs.”

Two users in the study did not use refinement at all (i.e., they always chose their desired expression after giving a single pdspec) and gave it no rating, but those who used it gave an average rating of 8.8. This suggests that some people really like using refinement, while others either do not or forgot about it after it was presented in the introduction.

Subjects rated the overall usefulness of CodeHint at an average of 7.6, and no subjects gave it below a 6.0. Interestingly, except in one case, each user gave a lower overall usefulness rating than for both of the more specific usefulness measures.

When in the choice group, each user was given the choice of whether or not to use CodeHint for each of the five tasks. Every user in the study chose to solve at least one task with it, and the average participant chose to use it to solve more than three out of five tasks.

From manual inspection, it seems that users chose not to use CodeHint mainly for simple statements that they knew how to write, often by using auto-complete. They seemed to use it when it was easy to give a pdspec (mainly as a value demonstration or type specification) or when the code involved using an API with which they were unfamiliar.

Overall, we can see that users gave CodeHint high ratings. In addition, all users reported that they would use it for their own development if it were available for their language and editor and had some simple flaws fixed. Three of the subjects actually asked for the plugin within a day of completing the user study and installed it.

**Properties of pdspecs**  In order to evaluate the types of pdspecs that are most useful to users in practice and discern if our current search space is sufficient for actual use, we gathered information on the pdspecs used by subjects in our user study and the effect of those pdspecs on the resulting synthesized code.

Figure 5 presents information about the number of candidates presented to the user after an initial pdspec is provided (i.e., the size of $C_0$), including tasks in the introduction. As can be seen, for the majority of all episodes (i.e., times when the subject gave a pdspec) there are a small number of candidates. The average number of candidates across all episodes was 9 and the median was 3. CodeHint generated and tested...
an average of 74 candidates (with a median of 21) for each pdspec, which reflected roughly the same distribution.\footnote{Our implementation uses an implicit pdspec to ensure that a variable is always assigned values that are a subtype of its static type. As a result, the user-provided pdspec is only tested against a small subset of the expressions in $F$.}

The few cases from the user study where $C_0$ was relatively large highlight an important point. There are some cases – such as boolean expressions or integer expressions when many integers are in scope – where there are many equivalent ways to generate the same value in a particular program state. While refinement will work for such cases, these may well be times when the user would be better off not using CodeHint.

When refining an existing set of candidates, users provided pdspecs that reduced the number of candidates by 40\% on average. However, this average is somewhat misleading, as 18 out of the 47 refinements did not reduce the size of the candidate set at all. All of these 18 were either repeated demonstrations of the same pdspec on the same input, cases where all the candidates were already equivalent on all possible inputs, or cases that contained some subtly different expressions (e.g., the Java Swing API provides multiple different methods for getting the clicked and selected elements of a JTree, which differ if the user right-clicks on an element). For refinements that reduced the size of $C_n$, the average reduction was 65\%.

Figure 6 presents the amount of time CodeHint took to synthesize the initial candidate set for each episode in the user study (including the introduction). On average, generating the candidate sets took 2.3 seconds, with a median of 0.9 seconds.\footnote{During the user study, we had screencasting software running that consumed an average of 80\% of both CPUs. This overhead does not appear to have skewed the results significantly, but it did prolong the generation time somewhat. For another purpose, one of the authors solved all of the tasks in the user study. The pdspecs used were similar to those given by the subjects. Those data, which are shown in Table 2, match up closely with those in Figures 5 and 6.}

\begin{figure}
\centering
\includegraphics[width=\columnwidth]{figure6}
\caption{The number of episodes synthesized in a given amount of time.}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\columnwidth]{figure7}
\caption{A classification of the pdspecs given by subjects during the user study. More than half were value demonstrations and more than a third simply specified the desired type.}
\end{figure}

finements since they involve very little work and are nearly instantaneous. This performance seems reasonable, but we have plans to reduce it further.

In general, finding the best pdspec requires trading the strength of the specification for the ease of typing it and the cost of evaluating it. Weak specifications are easy to write but might return many candidates while stronger ones can be very difficult to encode and might take longer to run. Our expectation before the user study was that weaker pdspecs with multiple refinements would be the best balance of complexity and time. To evaluate this, we classified all of the pdspecs used by subjects while completing the user study (excluding the introduction). We present the results in Figure 7. Over half of all pdspecs are value demonstrations, almost a third give the desired dynamic subtype, and only 12\% are arbitrary predicates.\footnote{These first two cases are two of the special cases we mentioned in Section 3.1. At the time of the user study, we had implemented a shortcut for value demonstrations but had not implemented one for dynamic type specifications.} This shows that users can get benefits with CodeHint even while demonstrating simple pdspecs, but the ability to provide more expressive pdspecs is valuable in some cases. One area we plan to explore in future work is how to make this tradeoff more transparent to users (see Section 7).

We list the arbitrary pdspecs we saw below, omitting four that were duplicates of others on the list.

\begin{verbatim}
1 x'.contains("-x") && x'.contains("-y")
2 x'.get(0).equals("-x") && x'.get(1).equals("-y")
3 x'.getWidth() == w && x'.getHeight() == h
4 x' >= 0 && x' < 3
5 x' >= 0 && x' < followers.size()
6 x'.toString().contains("home")
\end{verbatim}

The first two pdspecs express the same property, but the second is stronger than the first as it contains information about the order of elements in the List. Similarly, the fourth and fifth pdspecs also express the same property (in this case,
trying to find a random integer in a range), but the fourth uses the concrete value of `followers.size()` and so is more context-dependent. The sixth pdspec differentiated between many values of the same type by finding the one whose `toString` method returned a desired value (in this case, the element on which the user clicked), which allowed the user who wrote it to avoid figuring out how to express a property about a complicated piece of the Swing API.

When in the experimental group, one subject manually created a new test case that made giving a pdspec easier. This highlights the importance of the test case being exercised when using CodeHint; the same pdspec can yield very different results for different test cases.

**Implementation Limitations** In Section 4.1, we discussed how our implementation generates expressions up to a fixed depth (which was one for the user study). To demonstrate the importance of this factor, one of the authors attempted to complete the user study with a depth of two. In Table 2, we show timings and number of explored expressions for each task. The same pdspecs were used for both depths. As the data show, only four of the 15 tasks completed in under a minute, and they took an average of 16 seconds. The seven tasks that finished generating all the potential expressions found an average of 800 times more expressions than during the search with a depth of one. This shows the limitations of our current algorithms.\(^\text{10}\)

During the user study, several users found interesting ways to work around the restricted search space. We had included two tasks that were deliberately outside the scope of what was directly solvable with a single pdspec in the current tool. As expected, users tried pdspecs that did not have solutions within our search space. Two users, once they realized this, split the task into two sub-tasks by adding a temporary variable. They computed the temporary using the tool and then gave their original pdspec again, letting the new temporary be used in the generated expressions (essentially manually guiding our search). Another user gave a new pdspec that led to a related (but simpler) expression and then modified that by hand to find the correct expression. These two examples show that CodeHint can be useful even when it cannot find the desired expression.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Depth 1</th>
<th>Depth 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td># exprs Gen</td>
</tr>
<tr>
<td>Parse 1</td>
<td>3.9</td>
<td>174</td>
</tr>
<tr>
<td>Parse 2</td>
<td>4.8</td>
<td>216</td>
</tr>
<tr>
<td>Parse 3</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Parse 4</td>
<td>0.4</td>
<td>4</td>
</tr>
<tr>
<td>Parse 5</td>
<td>2.2</td>
<td>123</td>
</tr>
<tr>
<td>Swing 1</td>
<td>0.5</td>
<td>14</td>
</tr>
<tr>
<td>Swing 2</td>
<td>0.8</td>
<td>24</td>
</tr>
<tr>
<td>Swing 3</td>
<td>1.4</td>
<td>16</td>
</tr>
<tr>
<td>Swing 4</td>
<td>8.5</td>
<td>329</td>
</tr>
<tr>
<td>Swing 5</td>
<td>1.7</td>
<td>39</td>
</tr>
<tr>
<td>RW 1</td>
<td>3.1</td>
<td>175</td>
</tr>
<tr>
<td>RW 2</td>
<td>0.3</td>
<td>9</td>
</tr>
<tr>
<td>RW 3</td>
<td>0.3</td>
<td>5</td>
</tr>
<tr>
<td>RW 4</td>
<td>1.1</td>
<td>31</td>
</tr>
<tr>
<td>RW 5</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Average</td>
<td>2.2</td>
<td>89</td>
</tr>
</tbody>
</table>

**Table 2.** The time taken and number of expressions generated (Gen) and shown to the user (C₀) for search depths of one and two when we solved each task in the study. The N/A s represent tasks that could not be solved at the given depth and the dashes represent timeouts. (In some cases timeouts occurred during evaluation but after generation, in which cases we have numbers for generation but not C₀.)

\(^\text{10}\)It is worth noting that we have not attempted to optimize beyond depth 1; there are likely many opportunities to optimize our code for larger depths.
API Exploration There has been much research into tools to help programmers explore new APIs by querying databases of existing code to find code fragments that are used in practice [4, 9, 16, 17, 20]. As this work is targeted specifically at one domain, the algorithms used are more powerful than ours.

However, by evaluating code at runtime and allowing a wide variety of pdspecs, we gain two major benefits. First, we can differentiate between different values of the same type, as shown in the second example in Section 2, where we differentiated between many different integer expressions. Second, we allow users to give arbitrary pdspecs beyond just the desired type, such as x.getName().equals("foo.txt").

7. Future Work

We plan to explore options for expanding the class of statements we can synthesize with a particular emphasis on the synthesis of control flow constructs. We have already had some success synthesizing simple if-conditionals around assignments, but we hope to generalize this. The simplest approach is to merely enumerate options for both conditional and body, but we expect a different approach to synthesis will be needed to avoid a combinatorial explosion, especially once we consider loop constructs.

We would like to explore options for demonstrating values by manipulating a graphical representation of objects such as trees or graphs. We have implemented an early prototype of this approach, but it needs to be integrated into the conceptual framework of our current work. We could even investigate using domain-specific abstractions for domains such as cryptography that use graphical diagrams.

To give a pdspec, our implementation currently requires that users enter the debugger and select the variable to change in the Eclipse UI. We would like to allow users to write pdspecs directly into their code. For example, a user should be able to type something like
\[
x = \text{pdspec}(\text{prime}(x) == x + 1);
\]
and then tell the tool (perhaps through a right-click menu) that she wishes to synthesize expressions. We can then automatically save the code, set a breakpoint at that line, start debugging, and enter the desired text.

Reflections from User Study In general, finding a good pdspec can be difficult, as it requires a balance between those that are easy to write and those that prune many candidate statements. We want to explore giving users feedback as they type a pdspec. By immediately seeing how many statements are generated. By thus restricting the statements we generate, we would be able to do a deeper search and find more statements that fit the user’s restrictions.

By selecting a statement from the list of those synthesized and not carefully examining it, many users chose incorrect statements. We would like to help prevent this type of bug by integrating automatic testing into CodeHint. By running some tests whenever the user gives a pdspec, we may be able to detect certain incorrect statements that lead to crashes or assertion failures.

8. Conclusion

We have presented a novel methodology that helps programmers write difficult statements. They give partial dynamic specifications, which we call pdspecs, that show how the desired statement modifies memory in a particular program state. We use these pdspecs to synthesize candidate statements that are inserted into the code. Programmers can refine the candidates by providing further pdspecs.

We have implemented this approach as an Eclipse plugin for Java that is available at http://www.cs.berkeley.edu/~joel/codehint/. We conducted a user study and found that users rated the tool highly and chose to use it when given a choice.

Acknowledgments

We are very grateful to all of the participants in our user study for both their time and suggestions. We would also like to thank Leo Meyerovich for detailed feedback.

References