Quantifying the benefits of code hints for refactoring deprecated Java APIs

1st Cristina David
University of Bristol
Bristol, UK
cristina.david@bristol.ac.uk

2nd Pascal Kesseli
BEO IT
Zurich, Switzerland
pk@beo-it.ch

3rd Daniel Kroening
Amazon Inc.
Seattle, US
dkr@amazon.com

4th Hanliang Zhang
University of Bristol
Bristol, UK
hanliang.zhang@bristol.ac.uk

***Abstract*—When done manually, refactoring legacy code in order to eliminate uses of deprecated APIs is an error-prone and time-consuming process. In this paper, we investigate to which degree refactorings for deprecated Java APIs can be automated, and quantify the benefit of Javadoc code hints for this task. To this end, we build a symbolic and a neural engine for the automatic refactoring of deprecated APIs. The former is based on type-directed and component-based program synthesis, whereas the latter uses LLMs. We applied our engines to refactor the deprecated methods in the Oracle JDK 15. Our experiments show that code hints are enabling for the automation of this task: even the worst engine correctly refactors 71% of the tasks with code hints, which drops to at best 14% on tasks without. Adding more code hints to Javadoc can hence boost the refactoring of code that uses deprecated APIs.**

I. Introduction

As a project evolves, there are certain fields, methods or classes that the developers are discouraged from using in the future as they've been superseded and may cease to exist. However, removing them directly would break the backward compatibility of the project's API. Instead, in Java, such elements can be tagged with the @Deprecated annotation in order to ease the transition.

The transformation of existing code such that it doesn't use deprecated APIs is not always straightforward, as illustrated in Figure 1. In the example, we make use of the getHours method of the Date class, which is deprecated. In this situation, in order to replace the use of the deprecated method, we must first obtain a Calendar object. However, we can't use the Calendar constructor as it is protected, and we must instead call getInstance. Furthermore, in order to be able to use this Calendar object for our purpose, we must first set its time using the existing date. We do this by calling setTime with date as argument. Finally, we can retrieve the hour by calling calendar.get(Calendar.HOUR_OF_DAY).

Another example, which is easy for humans but difficult for automatic techniques, is the call to the static java.net.URLDecoder.decode(s) method, which should be refactored to the call decode(s, "UTF−8"). However, this requires guessing the "UTF-8" constant denoting a platform-specific string encoding scheme, which is very difficult for automatic code generation techniques, especially those using symbolic reasoning [1].

Besides general challenges related to automatic code generation, there are other language-specific ones. For instance, the code to be refactored might be using abstract classes and

```
void main(String[] args) {
  // Deprecated:
  int hour=date.getHours();

  // Should have been:
  final Calendar calendar=Calendar.getInstance();
  calendar.setTime(date);
  int hour=calendar.get(Calendar.HOUR_OF_DAY);
}
```

Fig. 1: Deprecated method example.

abstract methods, and it may not be obvious how to subclass from the code to be refactored (e.g. engineGetParameter in java.security.SignatureSpi); it might call methods that, while not abstract, need to be overridden by subclasses (e.g., method layout in class java.awt.Component has an empty body); or it might be calling native methods whose implementation is written in another programming language such as C/C++ (e.g., weakCompareAndSet in java.util.concurrent.atomic.AtomicReference). In order to even understand the behaviour of the code to be refactored, one needs to know how to subclass the abstract classes, override methods, and to understand the behaviour of code written in other languages.

In this paper, we are interested in the automatic generation of refactorings for deprecated APIs (while we focus on the refactoring of deprecated methods, the same techniques can be applied to deprecated fields and classes). In particular, we are interested in investigating the benefits of Javadoc code hints when automating such a refactoring.

When deprecating a field, method, or class, the @Deprecated Javadoc tag is used in the comment section to inform the developer of the reason for deprecation and, sometimes, what can be used in its place. We call such a suggestion a *code hint*. These hints don't provide the entire refactoring, but can be used to guide the search process. For illustration, let's look at our running example in Figure 1. The source code for the getHours method in class Date is accompanied by the comment in Figure 2, where the @code tag suggests replacing the deprecated getHours by Calendar.get(Calendar.HOUR_OF_DAY). Using this code hint is not straightforward. Although it may seem as if method get is static, allowing an immediate call, it is actually an instance method, and requires an object of class

```
/*
 * @deprecated As of JDK version 1.1,
 * replaced by {@code Calendar.get
 *                (Calendar.HOUR_OF_DAY)}.
 */
```

Fig. 2: Code hints for the running example.

Calendar. However, no such object is available in the original code, meaning that it must be created by the refactored code. Consequently, we must find the necessary instructions that consume existing objects and create a Calendar object. Besides generating the required objects, we also need to set their fields. For instance, in Figure 1, we must call calendar.setTime(date) to set the calendar's date based on the existing date object.

In order to investigate the benefits of code hints when automating the deprecated refactoring, we build two code generation engines, namely a symbolic and a neural one. For each, we selected approaches with proven success in program synthesis, and that could also incorporate code hints. In particular, for the symbolic engine, we make use of component-based synthesis [2], [3], [4] and type-directed synthesis [5], [6], [7]. Essentially, we use types and code hints to populate a component library (i.e., a library of instructions), such that these components are then weaved together to generate the desired program. Given the recent success of Large Language Models (LLMs) for code generation [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], the neural approach generates candidate refactorings by iteratively querying an LLM – we used Claude 2.1 and Claude 3 [19].

Our experiments show that code hints are critical for the performance of both the symbolic and the neural engines: when code hints are given, both engines perform very well, making full automation of deprecated APIs refactorings feasible. Without code hints, the refactoring becomes much harder for both engines, such that the vast majority of benchmarks without code hints are failing. Thus, our conclusion is that, in order to facilitate the automation of the refactoring of client code that uses deprecated APIs, code hints should be added to all deprecated methods that have a replacement.

Comparing the symbolic and the neural approaches, code hints help the symbolic approach to efficiently prune the solution space, resulting in a better performance than the neural engine at a lower monetary cost. We believe this is important to note, especially as LLMs are often seen as a panacea for all tasks, including code generation. This work shows that symbolic methods can still be effective in specialised settings, where efficient pruning of the solution space is possible.

a) Contributions:

- We propose a symbolic and a neural refactoring technique for deprecated APIs. The former makes use of type-directed and component-based synthesis, whereas the latter is LLM based. In order to check the correctness of the refactorings, we design a symbolic equivalence check, which takes into consideration both the state of the stack and the heap.
- We implement our techniques and use them to refactor deprecated methods from the Oracle JDK 15 Deprecated API documentation [20].
- We investigate the benefits of code hints for both the symbolic and neural approach. Our results show that code hints are critical for the performance of both the symbolic and the neural engines: adding more code hints to Javadoc can substantially help automate the refactoring of deprecated APIs.

II. Our approach

In this section, we describe the two code generation engines. For both approaches, we make use of a CounterExample Guided Inductive Synthesis (CEGIS) [21] architecture, where we iteratively attempt to improve a candidate refactoring until it behaves indistinguishably from the original code, i.e., the original and the refactored blocks of code are observationally equivalent. In the rest of the paper, we will use the predicate $equivalent(P_1(\vec{i}), P_2(\vec{i}))$ to denote that code P_1 is observationally equivalent to P_2 for inputs $\vec{i}$, meaning that the two programs can't be distinguished by their behaviour on inputs $\vec{i}$. In general, we refer to the original code as P_1 and the refactored code as P_2. We will fully define what $equivalent$ actually means in Section II-C.

In each iteration of the synthesis process, there are two phases, a synthesis phase and a verification phase. The synthesis phase generates a *candidate refactoring* that is equivalent to the original code on a finite set of inputs. Then, the verification phase tries to find a new *counterexample input* that distinguishes between the current candidate refactoring and the original code. If it manages, this input is added to the set of finite inputs used by the next synthesis phase, and if it fails, then the current candidate is indeed a solution refactoring.

A. Synthesis phase

In this phase, we have a finite set of input examples $\{\vec{i_1} \cdots \vec{i_n}\}$ and we attempt to find a candidate refactoring that is observationally equivalent to the original code for the given inputs.

a) Symbolic engine: We construct the following Synthesise method, which takes the refactored code P_2 as input.

```
Synthesise ($P_2$) {
  if (equivalent($P_1(\vec{i_1})$, $P_2(\vec{i_1})$) && ...
      && equivalent($P_1(\vec{i_n})$, $P_2(\vec{i_n})$))
    assert(false);
}
```

This method consists of only one conditional saying that if P_1 and P_2 are equivalent on all given inputs $\vec{i_1}, \cdots, \vec{i_n}$, then assert(**false**) is reached. Given that this assertion always fails if reached, it means that the method is unsafe when P_1 and P_2 are equivalent on the given inputs. Thus, we can reduce the synthesis problem to the problem of checking the safety of Synthesise. If a safety checker manages to find an input P_2 for which the assertion fails (meaning that Synthesise is unsafe), then this P_2 must be equivalent to P_1 on the given inputs.

This P_2 is the *refactoring candidate* that the synthesis phase is supposed to generate.

We use an existing fuzz testing platform for Java, JQF [22], as the safety checker. JQF is designed to handle structured inputs, where inputs of type `T` are generated by a backing `Generator<T>`. JQF provides a library of generators for basic types such as primitive values. We implement custom JQF generators based on a library of instructions built as explained in Section III, enabling JQF to construct P_2 by weaving together instructions from the library.

If the safety checker fails to find a P_2 for which the assertion fails, then the overall synthesis technique fails to generate a solution refactoring. There could be two reasons for this situation, which we cannot differentiate between. Firstly, there may indeed be no P_2 that can be constructed with the available components in the component library such that it is equivalent to P_1 on the given inputs. Secondly, this could be caused by JQF's unsoundness. Given that fuzzers rely on testing, they may fail to find inputs that trigger unsafe behaviours even when such inputs exist. Consequently, we cannot guarantee that we will always find a refactoring whenever one exists. We will provide more details about our decision to use fuzzing as the safety checker in Section IV.

b) Neural engine: For each deprecated method, we query the LLM by constructing a prompt as given in Figure 3. The method definition and the JavaDoc comment (if present) are provided as context. Furthermore, we provide a code snippet of the use of the deprecated method that is to be refactored (e.g., `this.minimumSize();`). Finally, we attach a list of formatting constraints.

The code returned by the LLM is verified against the original code. If the verification fails, we attach the set of counterexamples generated by the fuzzing engine to subsequent prompts. We apply object serialisation to obtain structured-views of the input and output states. In addition to this, we followed the prompting guidelines provided by Anthropic Claude[1] (e.g. using XML tags). Do note that, as opposed to the symbolic approach, the LLM doesn't give any guarantees that the counterexamples were actually taken into consideration.

B. Verification phase

This is exactly the same for the two approaches. We are provided with a candidate refactoring P_2 and we must check whether there exists any input $\vec{i}$ for which the original code and the candidate refactoring are not observationally equivalent. To do this, we build the following `Verify` method, which given some input $\vec{i}$, asserts that the two programs are equivalent for $\vec{i}$.

Verify($\vec{i}$) { assert(equivalent($P_1(\vec{i})$, $P_2(\vec{i})$)); }

In other words, answering the question posed by the verification phase is reduced to checking the safety of this method: if `Verify` is safe (i.e., the assertion is not violated) for any input $\vec{i}$, then there is no input that can distinguish between the original code and the candidate refactoring (this is indeed a sound refactoring). However, if `Verify` is not safe, then we want to be able to obtain a *counterexample input* $\vec{i_{cex}}$ for which the assertion fails. This counterexample will be provided back to the synthesis phase and used to refine the current candidate refactoring. Again, we check the safety of `Verify` with JQF.

[1]https://docs.anthropic.com/en/docs/prompt-engineering

```
Initial Prompt

User:
# Context
The method ${METHOD_NAME} of the class ${CLASS_NAME}
is deprecated. Below is the method definition:
${METHOD_DEFINITION}

Here are its Javadoc comments that may contain a
@deprecated tag explaining why the item has been
deprecated and suggesting what to use instead.
${JAVADOC_COMMENT}

However, I used this method call in my code base,
the code snippet is given below:
${CODE_SNIPPET}

# Instruction
Help me refactor this code snippet so that it
doesn't use the deprecated method. Do not simply
inline the method body, use APIs suggested by the
Javadoc comments if there are any.

# Constraints
Take the following constraints into consideration:
${FORMATTING_CONSTRAINTS}

Subsequent Prompt

Additionally, here is a set of input/output examples
that you should respect,
${EXAMPLES}

Assistant:
```

Fig. 3: LLM Prompt Template.

In this section, we hid the complexity of the equivalence check inside the *equivalent* predicate. This is far from trivial as it requires handling of notions such as aliasing, loaded classes, static fields etc. This will be described in detail next.

C. Checking program equivalence

A core part of the synthesis procedure is the $equivalent(P_1(\vec{i}), P_2(\vec{i}))$ predicate, which checks that the original code P_1 and a candidate refactoring P_2 are equivalent for a given input $\vec{i}$. In this section, we provide details on how we check this equivalence.

The state of a Java program is modelled by the current program stack (consisting of method-specific values and references to objects in the heap) as well as its heap (consisting of instance variables and static field values). Static field values are stored with their respective classes, which in turn are loaded by class loaders. Our equivalence check must take all these into consideration. One of the main challenges is aliasing, which we will discuss later in the section.

We start by introducing some notation:

- $loadedClasses(P)$ returns the set of classes loaded by the class loader in which P is executed. Note that Java allows to load the same class in different class loaders, which creates independent copies of its static fields.

- $\mathit{liveVars}(P)$ provides the set of variables that are live at the end of P.
- $\mathit{staticFields}(C)$ returns the set of static fields of class C.
- $\mathit{exc}(P, \vec{i})$ returns the exception thrown by P's execution on input $\vec{i}$.

Example 1. *For our running example in Figure 1, P_1 and P_2 are represented by the following lines of code. Note that both P_1 and P_2 take variable date as input.*

```
// P1
int hour=date.getHours();

// P2
final Calendar calendar=Calendar.getInstance();
calendar.setTime(date);
int hour=calendar.get(Calendar.HOUR_OF_DAY);
```

Then, we have:

$$
\begin{aligned}
\mathit{liveVars}(P_1) &= \{\mathit{date}, \mathit{hour}\} \\
\mathit{liveVars}(P_2) &= \{\mathit{calendar}, \mathit{date}, \mathit{hour}\} \\
\mathit{loadedClasses}(P_1) &= \{\mathit{Date}\} \\
\mathit{loadedClasses}(P_2) &= \{\mathit{Calendar}, \mathit{Date}\} \\
\mathit{staticFields}(\mathit{Date}) &= \emptyset \\
\mathit{staticFields}(\mathit{Calendar}) &= \{\mathit{DATE}, \mathit{YEAR}, \cdots\} \\
\mathit{exc}(P_1, \_) = \mathit{exc}(P_2, \_) &= \emptyset
\end{aligned}
$$

In order to extract the (last) object assigned to a variable v by the execution of P on a specific input $\vec{i}$, we will use the notation $E[P(\vec{i})](v)$. Essentially, if we consider the trace generated by executing $P(\vec{i})$, then $E[P(\vec{i})]$ maps each variable defined in P to the last object assigned to it by this trace. Next, we define the notion of equivalence with respect to a concrete input $\vec{i}$ (this definition is incomplete and we will build on it in the rest of the section). We overload equality to work over sets (for live variables and loaded classes).

Definition 1 (Program equivalence with respect to a concrete input $\vec{i}$ **[partial]**). *Given two code blocks P_1 and P_2 and concrete input $\vec{i}$, we say that P_1 and P_2 are equivalent with respect to $\vec{i}$, written as $\mathit{equivalent}(P_1(\vec{i}), P_2(\vec{i}))$ if and only if the following conditions hold:*

$$
\begin{aligned}
&(1)\ \mathit{exc}(P_1, \vec{i}) = \{e_1\} \wedge \mathit{exc}(P_2, \vec{i}) = \{e_2\} \wedge \mathit{equals}(e_1, e_2) \vee \\
&\quad \mathit{exc}(P_1, \vec{i}) = \mathit{exc}(P_2, \vec{i}) = \emptyset \\
&(2)\ \mathit{liveVar}(P_1) = \mathit{liveVar}(P_2) \wedge \forall v \in \mathit{liveVar}(P_1). \\
&\quad \mathit{equals}(E[P_1(\vec{i})](v), E[P_2(\vec{i})](v)) \\
&(3)\ \mathit{loadedClasses}(P_1) = \mathit{loadedClasses}(P_2) \wedge \\
&\quad \forall C \in \mathit{loadedClasses}(P_1). \forall f \in \mathit{staticFields}(C). \\
&\quad \mathit{equals}(E[P_1(\vec{i})](f), E[P_2(\vec{i})](f))
\end{aligned}
$$

The above definition says that in order for P_1 and P_2 to be equivalent with respect to input $\vec{i}$, (1) either they both throw an exception and the two exceptions have the same type, or none does, (2) the set of variables live at the end of P_1 is the same as the set of variables live at the end of P_2, and must be assigned equal objects by the executions of P_1 and P_2 on $\vec{i}$, respectively, and (3) the classes loaded by P_1 must be the same as the classes loaded by P_2, and all the static fields in the classes loaded by both the class loaders of P_1 and P_2 must be assigned equal objects by the two executions, respectively. In our implementation, if either the deprecated or the refactored code didn't load a class, we load it for it, and check that the initial state of that class is the same for both blocks of code.

Equality refers to value equality, which we check by recursively following attribute chains until we reach primitive types. We generally do not consider existing `equals` methods unless (i) the method was not written by the user (i.e., JCL classes), (ii) the type inherits from `java.lang.Object`, (iii) the class implements `java.lang.Comparable` and (iv) the type declares an `equals` implementation. Examples of classes that satisfy these strict requirements are `java.lang.Integer` or `java.util.Date`. All other implementations of `equals` are considered unreliable and ignored.

Example 2. *When applying Definition 1 to P_1 and P_2 given in Example 1, the following conditions must hold (given that class Date has no static fields, the third condition is trivial):*

$$
\begin{aligned}
&(1)\ \mathit{exc}(P_1) = \mathit{exc}(P_2) = \emptyset \\
&(2)\ \mathit{equals}(E[P_1(\vec{i})](\mathit{hour}), E[P_2(\vec{i})](\mathit{hour})) \wedge \\
&\quad \mathit{equals}(E[P_1(\vec{i})](\mathit{date}), E[P_2(\vec{i})](\mathit{date})) \\
&(3)\ \mathit{Date} \in \{\mathit{Date}, \mathit{Calendar}\}
\end{aligned}
$$

The challenge of aliasing. When expressing the equivalence relation between P_1 and P_2, we intentionally missed one important aspect, namely aliasing. To understand the problem let's look a the following example, which makes use of java.awt.Container, where a generic Abstract Window Toolkit (AWT) container object is a component that can contain other AWT components. Method `preferredSize` used by the original code below returns the preferred size of the calling container. Method `preferredSize` is deprecated, and, instead, the refactored version uses `getPreferredSize`.

Example 3.

```
// P3:
Dimension dim1=container.preferredSize();
Dimension dim2=container.preferredSize();

// P4:
Dimension dim1=container.getpreferredSize();
Dimension dim2=dim1;
```

We note that, the original code above defines two variables dim1 and dim2, each assigned an object of type `Dimension` *returned by calling $\mathit{container.preferredSize}()$. Conversely, in the refactored code, dim1 and dim2 are aliases, i.e., they point to the same object of type* `Dimension`.

The original and the refactored code are equivalent according to Definition 1. Let's next assume that the following code is used after both the original and the refactored code, respectively.

```
dim1.setSize(1,2);
dim2.setSize(2,3);
```

If the original and the refactored code were indeed equivalent, then we would expect dim1 and dim2 to have the same value *at the end of both blocks of code. However, this is not the case. In the original code, dim1 will have width 1 and height 2, whereas in the refactored code, dim1 will have width 2 and height 3. This is due to the fact that, in the refactored code, dim1 and dim2 are aliases. Thus, when dim2 has its size set to (2,3), this also affects dim1.*

Intuitively, any aliases between live variables at the end of the original code should also be present at the end of the refactored code. For this purpose, we use the notation $aliasEquivClass(V)$ which returns the set of all equivalence classes induced over the set of variables V by the aliasing relation. Notably, the aliasing equivalence relation must also hold over the static fields of the classes loaded by both programs.

Example 4. *For P_1 and P_2, there are no aliases. However, for P_3 and P_4 we have:*

$$\begin{aligned}&aliasEquivClass(liveVar(P_3) \cup \\ &staticFields(loadedClasses(P_3)) = \emptyset \\ &aliasEquivClass(liveVar(P_4) \cup \\ &staticFields(loadedClasses(P_4)) = \{\{dim1, dim2\}\}\end{aligned}$$

In P_4, the aliasing relation induces one equivalence class, namely $\{dim1, dim2\}$.

Next, we complete Definition 1 by capturing the aliasing aspect.

Definition 2 (Addition to Definition 1)**.** *In addition to Definition 1, two programs P_1 and P_2 are equivalent with respect to $\vec{i}$ iff:*

$$\begin{aligned}&(4)\ \forall v_1, v_2 \in liveVar(P_1) \cup staticFields(loadedClasses(P_1)). \\ &\quad aliases(P_1, v_1, v_2) \iff aliases(P_2, v_1, v_2)\end{aligned}$$

where, given a program P, variables v_1 and v_2 are aliases, i.e., $aliases(P, v_1, v_2)$ holds if and only if they are in the same equivalence class induced by the aliasing relation, i.e., $aliasEquivClass(\{v_1, v_2\}) = \{\{v_1, v_2\}\}$.

We abuse the notation to use *staticFields* over a set of classes, rather than just one class. The objective is to return the union of all static fields defined in all the classes in the set of classes taken as argument.

III. Seeding of the component library

In this section, we describe how we populate the instruction library (also referred to as the component library) used by the symbolic approach starting from code hints. We'll refer to this library as the CodeHints-library. This step is critical as, in order for the symbolic engine to succeed, the library must be as small as possible while containing all the instructions necessary for constructing the solution.

For illustration, let's go back to the running example in Figure 1 (with the corresponding Javadoc comment in Figure 2). As mentioned in Section I, when building the corresponding component library for this example, we must find the necessary components that consume existing objects and create a Calendar object, so that we can call method get as suggested by the code hint. Besides adding components that allow us to generate the required objects, we also need components for setting their fields. In our example, we must call calendar.setTime(date) to set the calendar's date based on the existing date object.

Adding too many components to the library will make the code generation task infeasible. In particular, we should be able to differentiate between components that we can use (we have or we are able to generate all the necessary arguments and the current object for calling them), and those that we can't because we can't obtain some of the arguments and/or the current object. Adding the latter components to the library will significantly slow down the code generation process by adding infeasible programs to the search space. We address these challenges by dynamically building the component library for each refactoring such that it only contains components specific to that particular use case.

Throughout the seeding process, we keep track of the following sets: *consumable_objs* (inputs to the code to be refactored, which need to be consumed by the refactoring), *available_types* (types for which we either have consumable objects or the corresponding generators to create them) and *target_types* (types for which we must be able to generate objects). To start with, the *target_types* set contains the types of the original code's outputs.

For the running example, we start with: $consumable\_objs = \{\texttt{date}\}$, $available\_types = \{\texttt{Date}\}$, $target\_types = \{\texttt{int}\}$. The objective of the seeding algorithm is to add components to the library that make use of the *available_types* to generate objects of *target_types*. At the same time, we want to consume the objects from the *consumable_objs* set, i.e., the inputs of the original code.

The seeding algorithm for the CodeHints-library is provided in Figure 5 and consists of three phases, which we discuss next.

a) **Phase 1: Initialise with code hints***:* During the first phase, the initialisation, we add all the constants and instructions from the code hints to the library. For our running example, the hints in Figure 2 instruct us to add method "**int** get(**int** field)" from class Calendar and constant "Calendar.HOUR_OF_DAY" to our library. As a side note, finding the right constants is a well known challenge for program synthesis [1], and thus the subsequent synthesis process will always attempt to use the constants provided in the code hints before generating new ones.

Intuitively, we need to make sure that the Javadoc suggestions are realisable as captured by Definition 3, where $required\_types(method)$ refers to the types of the objects required to call $method$ (i.e., the types corresponding to its arguments and current object). In our running example, $required\_types(\texttt{get}) = \{\texttt{int, Calendar}\}$ given that, in order to call get, we must provide an argument of type **int** and a current object of type Calendar. Method get is not realisable as the library doesn't contain any generator for Calendar. Consequently, Calendar is added to *target_types*, resulting in: $target\_types = \{\texttt{int, Calendar}\}$.

```
/* @deprecated As of JDK version 1.1,
* replaced by {@code contains(int, int)}.
*/
@Deprecated
public boolean inside(int X, int Y) { ... }
```

Fig. 4: Javadoc hint example.

Definition 3 (Realisable method). *Method i is* realisable *iff* $\forall t \in required\_types(i).t \in available\_types \vee t$ *is a primitive type.*

One challenge in this phase is interpreting the @code blocks inside @deprecated sections, which we attempt to parse as Java expressions. In particular, we parse each hint as if it were invoked in the context of the method to refactor, such that imports or the implicit **this** argument are considered during parsing.

In order to address the challenge that code hints are not always expressed as well-formed Java, we customised the GitHub Java parser to accept undeclared identifiers and type names as arguments. For instance, the Javadoc hint for deprecating **boolean** inside(**int** X, **int** Y) in Figure 4 suggests using contains(**int**, **int**), which would normally cause a parsing error as the **int** type appears in the place of argument names. In our setting, we accept this hint as valid. There are still situations where our parser is too strict and fails to accept some of the code hints. Additionally, there are scenarios where, while the Javadoc does contain a useful code hint, it is not tagged accordingly with the @code tag.

b) **Phase 2: Add generators for** *$target\_types$:* By generators we refer to constructors and any other methods returning objects of that particular type. In the algorithm in Figure 5, once we added a generator for a new type, we must add that type to $available\_types$. Additionally, if the new generator consumes any objects from $consumable\_objs$, we must remove them from the set.

For our running example, we must seed our component library with generators for Calendar. We first scan all the public constructors of class Calendar and all the public methods from class Calendar that return an object of type Calendar. We find the following four options:

1) **static** Calendar getInstance() – creates the object using the default time zone and locale.
2) **static** Calendar getInstance(Locale) – creates the object using the default time zone and specified locale.
3) **static** Calendar getInstance(TimeZone) – creates the object using the specified time zone and default locale.
4) **static** Calendar getInstance(TimeZone, Locale) – creates the object with the specified time zone and locale.

Out of the four methods, only the first one is realisable. Conversely, in order to call the second method, we would need to generate an object of type Locale, for which we don't have a corresponding component in the library, and the same applies to the last two methods. Thus, we only add the first method to the CodeHints-library.

```
Output: CodeHints-library
// Phase 1: Initialise
Add constants and instructions from the code hints to the
  library;
for each unrealisable instruction $i$ in library do
    $target\_types = target\_types \cup required\_types(i)$;
end

// Phase 2: Add generators for $target\_types$
for each $t \in target\_types$ for which there is no generator do
    Add realisable generators for $t$ to library;
    $available\_types = available\_types \cup \{t\}$;
    Remove consumed objs from $consumable\_objs$;
end

// Phase 3: Add transformers for $target\_types$
while $consumable\_objs \neq \emptyset$ do
    Add realisable transformers to the library for types in
      $target\_types$ that consume objects from
      $consumable\_objs$;
    Remove consumed objs from $consumable\_objs$;
end
```

Fig. 5: Seeding algorithm for the CodeHints-library

c) **Phase 3: Add transformers for** *$target\_types$:* The third and last phase adds transformers for the target types. By transformers we refer to methods that modify the value of an instance variable. Given our objective to generate objects for the $target\_types$ while consuming objects from $consumable\_objs$, we prioritise transformers that consume such objects. For instance, for our running example there are 16 public transformers for the Calendar class. However, instead of adding all of them to the CodeHints-library, we prioritise those that consume the date object. There is only one such transformer **void** setTime(Date date). Once we added any new transformers to the library, we remove the consumed objects from $consumable\_objs$.

Notably, all the components that we add to the library must be accessible from the current location.

It may be the case that there is no realisable generator for a target type, or no transformers for the $target\_types$ that can consume all the $consumable\_objs$. When we finish exploring all the available methods, we exit the corresponding loops in phases 2 and 3. As a consequence, the synthesiser may fail to generate a valid refactoring from the CodeHints-library. A possible future direction is allowing unrealisable generators and transformers to be added to the library and iterating phases 2 and 3 a given number of times (potentially until reaching a fixed point).

A. Types-library

For cases when code hints are not provided, in addition to the CodeHints-library, we also provide a component library that is populated based only on the type signature of the deprecated method. Next, we describe how this Types-library is being built.

For the CodeHints-library, the Javadoc code hints guidance results in the code hints being used to collect the *target_types* set during the initialisation phase. If we don't have access to code hints and we are seeding solely based on types, we start with the *target_types* set containing the types of the outputs of the original code. Phases 2 and 3 of the seeding algorithm are the same as those in Figure 5, where we attempt to add generators and transformers for the *target_types* to the library.

Compared to the CodeHints-library, the seeding process for the Types-library may end up missing critical types provided by the Javadoc code hints. For instance, in our running example, the `Calendar` class is only mentioned by the code hints and would not be included in the seeding of the Types-library. One possibility for adding `Calendar` to the target types without considering the Javadoc hints, would be to add all the classes from the `java.util` package, which contains `Date`. However, doing so would result in a very large component library, most likely outside the capabilities of existing synthesis techniques.

IV. Design and implementation choices

A. Abstract classes and interfaces

The programs that need to be checked for equivalence may refer to abstract classes and interfaces. These are by default instantiated using the mocking framework Mockito. Alternatively, we also curate a list of explicit constructors of subclasses to be used in favour of Mockito mocks for certain types, and users have the option to extend this list with a custom configuration. This is particularly useful when attempting to avoid constructors that are not amenable to fuzzing, such as collection constructors that take an integer capacity argument, where an unlucky fuzzed input might lead to an out of memory error.

B. Observable state

Our equivalence check uses the Java Reflection API to observe the side effects of programs, such as assigning new values to fields. As a consequence, we cannot observe effects that are not visible through this API, such as I/O operations or variables maintained in native code only. I/O operations could be recorded using additional bytecode instrumentation in future work, but for the scope of our current implementation, if programs only differ in such side effects, our engine is prone to producing a refactoring that it's not fully equivalent to the original.

C. Instrumentation and isolation

We use reflection to invoke the original method to refactor with fuzzed inputs, as well as our synthesised refactoring candidates. This allows us to dynamically invoke new candidates without the need for compilation. Regarding static fields, Java allows to load the same class in different class loaders, which creates independent copies of its static fields.

In order to fully isolate the program state during the execution of the original and refactored code from each other, we load all involved classes in separate class loaders. These class loaders are disposed immediately after the current set of fuzzed inputs are executed, and new class loaders are created for the next inputs. Classes that do not maintain a static state are loaded in a shared parent class loader to improve performance.

The OpenJDK Java virtual machine implementation does not allow us to load classes contained in the `java.lang` package in such an isolated class loader. For such classes we cannot apply refactorings that depend on static fields, as they cannot be reset and will retain the state of previous executions. Instead of observing the effect of one isolated refactoring candidate, we would observe their accumulated effect, which would be incorrect. For the scope of our experiments this did not pose a problem, since most of the affected classes in the `java.lang` package do not maintain a static state, and the ones who do were irrelevant to our refactorings.

D. Checking aliasing

While in Section II-C, we discussed aliasing preservation in terms of preserving the equivalence classes induced by the aliasing relation, in our implementation we enforce a simpler but stricter than necessary check. In particular, all the objects that are being referenced during the code's execution are assigned increasing symbolic identifiers. Then, we enforce aliasing preservation (condition (4) in Definition 2), by checking that, for all variables defined by both the original and the refactored code, the objects they reference have the same symbolic id.

For illustration, in the original code in Example 3, the object of type `Dimension` referenced by variable *dim1* is assigned symbolic value s_1, whereas the object referenced by *dim2* is assigned symbolic value s_2. In a similar manner, the object referenced by both *dim1* and *dim2* in the refactored code is assigned value s_1. Then, the aliasing check fails as, in the original code, the object referenced by *dim2* has symbolic value s_2, whereas, in the refactored code, the object referenced by *dim2* has symbolic value s_1. While this is a stronger than needed requirement, it was sufficient for our experiments. In particular, we didn't find any benchmark where a sound refactoring was rejected due to it.

V. Experimental evaluation

We implemented the refactoring generation techniques in two tools called RefSym and RefNeural, corresponding to the symbolic and the neural approach, respectively (available together with all the experiments at https://github.com/pkesseli/refactoring-synthesis/tree/hanliang/dev).

A. Experimental setup

Our benchmark suite contains 236 out of 392 deprecated methods in the Oracle JDK 15 Deprecated API documentation [20]. We included all the deprecated methods except those that were deprecated without a replacement (e.g. `java.rmi.registry.RegistryHandler.registryImpl(int)`), methods inaccessible by non-JCL classes (i.e., all the finalize methods, which are called by the garbage collector, not user code, meaning that we can't modify their calls), methods that only perform I/O (as our equivalence check doesn't include I/O side effects), and native methods.

For REFNEURAL, we use Claude 2.1 and Claude 3, denoted as REFNEURAL-Claude2.1 and REFNEURAL-Claude3, respectively. Claude 2.1 and Claude 3 are proprietary LLMs likely to be very large (1T+ parameters), which have been shown to be among the highest performing on coding tasks [23], [24], [25], [26]. We accessed the LLMs via Amazon Bedrock. To make our results more deterministic, we use a lower temperature (i.e. less random) of 0.2. We also repeated the experiments three times, and the results are summarized as averages. For all engines, we bound the search by at most 500 inputs and 5 minutes per verification phase, and at most 2 minutes per synthesis phase.

Experiments were performed on an Ubuntu 22.04 x64 operating system running in a laptop with 16 GB RAM and 11th Gen Intel Core i7-11850H at 2.50 GHz. The JVM used was Oracle JDK 15.02.

For REFSYM, if code hints are present, we start with the CodeHints-library and fall back to the Types-library if the synthesis phase (as described in Section II-A) times out for the CodeHints-library; if no code hints exist then we use the Types-library.

B. Results

Configuration	✓	✗	ϟ	%	∅ runtime (s)
REFSYM	7	82	16	6	213.7
REFNEURAL-Claude2.1	12	91	2	11	197.14
REFNEURAL-Claude3	10	93	2	9	203.32
Best Virtual Engine	15	77	13	14	195.9
REFSYM (CH)	104	19	8	79	220.8
REFNEURAL-Claude2.1 (CH)	93.3	36.7	1	71	212.19
REFNEURAL-Claude3 (CH)	94	36	1	72	217.9
Best Virtual Engine (CH)	107	13	11	82	210.28

TABLE I: Experimental results for all benchmarks.

Table I provides an overview of the number of sound refactorings (✓), missed refactorings (✗), unsound refactorings (ϟ), the percentage of sound refactorings (%) produced per configuration, as well as the average runtime per refactoring. For the symbolic engine, refactorings are guaranteed to compile so missed refactorings are those that failed our equivalence check, whereas for the neural engine, they either didn't compile or failed the equivalence check. Unsound refactorings are those that passed the equivalence check, but were manually detected by us as not being equivalent to the original code. We describe the reasons why this can happen later in this section. The average runtime includes both the time to generate a refactoring and to verify it.

We split our dataset into benchmarks where code hints could be extracted from the Javadoc, and benchmarks without code hints. The first four rows provide the results for benchmarks without code hints, whereas the last four (marked with "(CH)") show the results for those benchmarks where code hints were present. The rows denoted by "Best Virtual Engine" count all benchmarks (with and without code hints, respectively) solved by at least one engine.

The results support our hypothesis that code hints are very valuable for automating the refactoring process. For all engines, benchmarks with code hints have a considerably higher success rate than those without code hints, with the best virtual engine solving 82% of the benchmarks with code hints. In the absence of code hints, all the engines are struggling, solving only a few benchmarks.

To understand the discrepancy between the number of missed/unsound refactorings with the without code hints, we next investigate the main reasons for such refactorings.

a) Missed refactorings: For the symbolic engine, the majority of the missed refactorings were due to fuzzing timeouts, which are likely caused by the component libraries missing some instructions needed in the synthesis phase. As explained in Section III, one option for increasing the size of our libraries is allowing unrealisable generators and transformers to be added.

For the neural approach, we could only observe that the LLMs were unable to generate the refactoring from the provided prompt.

For both engines, when refactoring methods that eventually run native code, we encountered crashes of the verifier, which, in order to be conservative, we counted as overall missed refactorings. The reason for this is the fact that we use reflection to produce counterexamples, and some may violate internal invariants. If they execute methods that eventually call native code, this can lead to the entire JVM crashing rather than e.g. throwing an exception.

b) Unsound refactorings: In several cases, refactorings that are not equivalent to the original code managed to pass our verifier. While we expected to run into this problem because of the nature of fuzzing (the fuzzer may miss counterexamples that distinguish the behaviour of the original from the refactored code), we also encountered other problems:

Unobservable state (discussed in Section IV-B): I/O operations, static state in the boot class loader and native methods are not observable by our equivalence predicate implementation, since we rely on reflection to examine objects inside the Java runtime. As a consequence, we cannot distinguish programs that only differ in these aspects.

Abstract methods: There are classes in the JCL without any existing implementation (e.g. javax.swing.InputVerifier), and thus no concrete method against which to verify the equivalence between the original and the refactored code. For those, the symbolic engine will generate a no-op. We've been very conservative here, and counted this scenario as "unsound" because it doesn't match human intent for the deprecated method.

Insufficient counterexamples: Our fuzzing-based equivalence check is inherently incomplete, and for some benchmarks we do not explore sufficient counterexamples to identify unsound candidates. An example is javax.swing.JViewport#isBackingStoreEnabled, where only a single input out of the $2^{32} - 1$ possible input values will trigger an alternate code path. As a second exemplar, method java.rmi.server.RMIClassLoader#loadClass accepts a string as an input, but will throw when given any string that is not a valid class name. It is very unlikely that our fuzzer will randomly produce a string that matches a valid class name.

c) Discussion: The symbolic engine benefits significantly from code hints, as hints seed the component library, making it less likely that needed instructions are missing. For the neural engine, we hypothesise that, by providing additional context to the LLM, code hints are aiding the code generation task.

Benchmarks with code hints also have fewer unsound results. Our hypothesis is that both engines are much more likely to generate the expected refactoring before generating other candidate refactorings that may trick our equivalence checker.

C. Research questions

(RQ1) Can the refactoring of deprecated Java APIs be automated? Yes, it can be automated if code hints are added to the JDK, evidenced by the 82% success rate for those benchmarks. Without code hints, Java's complexity makes these refactorings very hard for both for the symbolic and neural engines.

(RQ2) Do code hints help the generation of refactorings? The performance of REFSYM, REFNEURAL-Claude2.1 and REFNEURAL-Claude3 improves considerably when code hints are present. All of them barely manage to solve any benchmark without code hints. When code hints are present, all engines solve at least 71% of benchmarks, with the best virtual engine solving 82%. For the symbolic engine, they enable the effective seeding of the instruction library, whereas for the neural engine, they provide additional context to the LLM.

(RQ3) How do the symbolic and the neural approach compare against each other? The symbolic and the neural engines have very similar performance (where the symbolic engine has a smaller monetary cost).

When code hints are present, as shown in Table I, the symbolic approach does slightly better than both REFNEURAL-Claude2.1 and REFNEURAL-Claude3. This supports the intuition that, if there is enough information about the solution to effectively prune the solution space (in our case, when code hints are present), then the symbolic approach works well. One example of a benchmark that was solved by REFSYM but where both REFNEURAL-Claude2.1 and REFNEURAL-Claude3 failed to find a solution is the running example in Figure 1. In all our runs, the LLMs failed to generate `calendar.setTime(date)`.

When code hints are not present, the neural engine does marginally better than the symbolic approach. Compared to the symbolic engine, the neural one is able to provide correct refactorings for deprecated concurrency primitives (e.g., `weakCompareAndSet` in `java.util.concurrent.atomic.AtomicReference`). These methods call native methods, for which we cannot observe side-effects (Section IV-B). Consequently, these are not captured by the counterexamples returned by the fuzzer, and are thus not taken into consideration by the symbolic engine during the synthesis phase. For the neural approach, the counterexamples are less critical, and the LLM can obviously generate the correct code even though they are incomplete.

The neural engine is also able to handle benchmarks that require special constants, such as `java.net.URLDecoder.decode(s)` described in the introduction.

VI. THREATS TO VALIDITY

a) Selection of benchmarks: All our benchmarks are Java methods deprecated in the Oracle JDK 15. The JDK is extremely well known to Java developers, and a lot of Java application code evolves similarly. However, our claims may not extend to other programming languages.

b) Quality of refactorings: Refactorings need to result in code that remains understandable and maintainable. It is difficult to assess objectively how well our technique does with respect to this subjective goal. This threatens our claim that refactoring of deprecated Java APIs be automated.

We manually inspected the refactorings obtained with both engines and found them to represent sensible transformations.

c) Efficiency and scalability of the program synthesiser: We apply program synthesis and fuzzing. This implies that our broader claim is threatened by scalability limits of these techniques. While for the majority of our experiments the synthesiser was able to find a solution, there were a few cases where it timed out, either because it could not generate a candidate, or it could not verify it. Component libraries with a diverse range of component sizes may help mitigate this effect.

d) Prompt engineering for Claude: In our current experiments, we engineered prompts for the Claude LLMs. While we made best efforts to follow the official prompt engineering guidelines[2], which presents prompt design techniques to improve model performance, there might be better ways of composing it. This might invalidate our claim that the symbolic and neural techniques deliver roughly the same performance.

VII. RELATED WORKS

a) Program refactoring: We first discuss works on the refactoring of deprecated instances. The work of Perkins directly replaces calls to deprecated methods by their bodies [27], but we argue this conflicts with the intent of language designers. Moreover, it can introduce concurrency bugs as inlining calls to deprecated methods can cause undesirable effects if the original function was synchronised. While it is not possible for two invocations on the same object of the synchronised original method to interleave, this is not guaranteed after inlining the method's body. Notably, the authors of [28] show how refactorings in concurrent programs can inadvertently introduce concurrency issues by enabling new interactions between parallel threads.

A related class of techniques aim to adapt APIs after library updates. Such techniques automatically identify change rules linking different library releases [29], [30]. Conversely to our work, a change rule describes a match between methods existing in the old release, but which have been removed or deprecated in the new one, and replacement methods in the new release. However, they do not provide the actual refactored

[2]https://docs.anthropic.com/en/docs/prompt-engineering

code. In [31], Lee et al. address the problem of outdated APIs in documentation references. Their insight is that API updates in documentation can be derived from API implementation changes between code revisions. Conversely, we are looking at code changes, rather than documentation. Other works focus on automatically updating API usages for Android apps based on examples of how other developers evolved their apps for the same changes [32], [33]. Furthermore, [34] improves on [33] by using a data-flow analysis to resolve the values used as API arguments and variable name denormalization to improve the readability of the updated code. As opposed to these works, which rely on examples of similar fixes, our approach uses Javadoc code hints.

There are several rule-based source-to-source transformation systems that provide languages in which transformation rules based on the program's syntax can be expressed [35], [36]. As opposed to our work, such systems require the user to provide the actual transformation rules.

Search-based approaches to automating the task of software refactoring, based on the concept of treating object-oriented design as a combinatorial optimisation problem, have also been proposed [37], [38]. They usually make use of techniques such as simulated annealing, genetic algorithms and multiple ascent hill-climbing. While our technique is search based, the search is guided by types, code hints, as well as counterexamples, discovered by testing.

Closer to our work, several refactoring techniques make explicit use of some form of semantic information. Khatchadourian et al. use a type inference algorithm to automatically transform legacy Java code (pre Java 1.5) to use the `enum` construct [39]. Other automated refactoring techniques aim to transform programs to use a particular design pattern [40], [41]. Steimann et al. present Constraint-Based Refactoring [42], [43], [44], where given well-formedness logical rules about the program are translated into constraints that are then solved to assist the refactoring. Fuhrer et al. implement a type constraint system to introduce missing type parameters in uses of generic classes [45] and to introduce generic type parameters into classes that do not provide a generic interfaces despite being used in multiple type contexts [46]. Kataoka et al. use program invariants (found by the dynamic tool Daikon) to infer whether specific refactorings are applicable [47]. Finding invariants is notoriously difficult. Moreover, the technique is limited to a small number of refactorings and does not include the elimination of deprecated instances. In [48], Gyori et al. present the tool LAMBDAFICATOR, which automates two pattern-based refactorings. The first refactoring converts anonymous inner classes to lambda expressions. The second refactoring converts `for` loops that iterate over Collections to functional operations that use lambda expressions. The LAMBDAFICATOR tool [49] is available as a NetBeans branch.

Our techniques does not expect any precomputed information such as logical well-formedness properties or invariants. Instead, we make use of information that is already available in the original program and Javadoc, namely code hints and type information. Moreover, we explore the space of all potential candidate programs by combining techniques from type-directed, component-based and counterexample-guided inductive synthesis.

b) Symbolic program synthesis: CEGIS-based approaches to program synthesis have been previously used for program transformations such as superoptimisation and deobfuscation [2]. In [50], David et al. present an automatic refactoring tool that transforms Java with external iteration over collections into code that uses Streams. Their approach makes use of formal verification to check the correctness of a refactoring. Cheung et al. describe a system that transforms fragments of application logic into SQL queries [51] by using a CEGIS-based synthesiser to generate invariants and postconditions validating their transformations (a similar approach is presented in [52]). While our approach is also CEGIS-based, we are guided by code hints and types to efficiently prune the search space.

Type information has been extensively used in program synthesis to guide the search for a solution [5], [53], [6], [54], [7]. In our work, we combine type information with code hints. Another direction that inspired us is that of component-based synthesis [2], [3], [4], where the target program is generated by composing components from a library. Similarly to these approaches, we use a library of components for our program generation approach. However, our technique uses information about types and code hints to build the component library, which is specific to each refactoring.

c) Large Language Models: LLMs have been used successfully for code generation tasks, with applications ranging from code completions [12], [14], [13], [17], translations [55], [16] to repository-level generation [15] and general software engineering tasks [18]. Those models are typically pre-trained on vast amounts of data, fine-tuned for specific tasks, and require advanced prompt engineering [56]. Among the well-known LLMs, GPT-4 [57], Claude2.1 [19], Claude3 [19], LLaMa [58] are general-purpose models covering a diverse set of language-related applications; there are also models pre-trained/fine-tuned specifically for code generation and programming tasks, such as CodeLLaMa2 [59], StarCoder2 [60], DeepSeek-Coder [61], and GrammarT5 [62]. In this work, we have applied the Claude family of LLMs to our refactoring task, with the goal of investigating how much it benefits from Javadoc code hints.

VIII. CONCLUSIONS

In this paper, we investigated the benefits of Javadoc code hints when refactoring deprecated methods. For this purpose, we designed and implemented a symbolic and a neural automatic program refactoring techniques that eliminate uses of deprecated methods. In our experiments, both engines demonstrate strong performance when code hints were present, and fare much worse otherwise, leading us to conclude that code hints can boost the automation of refactoring code that uses deprecated Java APIs.

REFERENCES

[1] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen, "Counterexample guided inductive synthesis modulo theories," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as*

Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (H. Chockler and G. Weissenbacher, eds.), vol. 10981 of *Lecture Notes in Computer Science*, pp. 270–288, Springer, 2018.

[2] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010* (J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, eds.), pp. 215–224, ACM, 2010.

[3] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (M. W. Hall and D. A. Padua, eds.), pp. 62–73, ACM, 2011.

[4] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex APIs," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (G. Castagna and A. D. Gordon, eds.), pp. 599–612, ACM, 2017.

[5] S. Katayama, "Systematic search for lambda expressions," in *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005* (M. C. J. D. van Eekelen, ed.), vol. 6 of *Trends in Functional Programming*, pp. 111–126, Intellect, 2005.

[6] P. Osera and S. Zdancewic, "Type-and-example-directed program synthesis," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (D. Grove and S. M. Blackburn, eds.), pp. 619–630, ACM, 2015.

[7] M. Yamaguchi, K. Matsuda, C. David, and M. Wang, "Synbit: synthesizing bidirectional programs using unidirectional sketches," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–31, 2021.

[8] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "CodeT: Code generation with generated tests," 2022.

[9] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021.

[10] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "CodeBERTScore: Evaluating code generation with pretrained models of code," *CoRR*, vol. abs/2302.05527, 2023.

[11] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," *CoRR*, vol. abs/2202.13169, 2022.

[12] M. Izadi, J. Katzy, T. van Dam, M. Otten, R. Popescu, and A. van Deursen, "Language models for code completion: A practical evaluation," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, (Los Alamitos, CA, USA), pp. 956–968, IEEE Computer Society, apr 2024.

[13] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[14] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-t. Yih, S. I. Wang, and X. V. Lin, "LEVER: Learning to verify language-to-code generation with execution," in *Proceedings of the 40th International Conference on Machine Learning*, ICML'23, JMLR.org, 2023.

[15] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "RepoCoder: Repository-level code completion through iterative retrieval and generation," in *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[16] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[17] Y. Ding, Z. Wang, W. Ahmad, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang, "CoCoMIC: Code completion by jointly modeling in-file and cross-file context," in *LREC-COLING 2024*, 2024.

[18] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," 2024.

[19] "Claude." https://www.anthropic.com/index/introducing-claude.

[20] Oracle, "Deprecated list (Java SE 15 & JDK 15)," 2020.

[21] A. Solar-Lezama, C. G. Jones, and R. Bodík, "Sketching concurrent data structures," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008* (R. Gupta and S. P. Amarasinghe, eds.), pp. 136–148, ACM, 2008.

[22] R. Padhye, C. Lemieux, and K. Sen, "JQF: coverage-guided property-based testing in Java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019* (D. Zhang and A. Møller, eds.), pp. 398–401, ACM, 2019.

[23] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening, "Towards translating real-world code with llms: A study of translating to rust," *CoRR*, vol. abs/2405.11514, 2024.

[24] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024.

[25] W. Hou and Z. Ji, "Comparing large language models and human programmers for generating programming code," 2024.

[26] L. Murr, M. Grainger, and D. Gao, "Testing llms on code generation with varying levels of prompt specificity," 2023.

[27] J. H. Perkins, "Automatically generating refactorings to support API evolution," in *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005* (M. D. Ernst and T. P. Jensen, eds.), pp. 111–114, ACM, 2005.

[28] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip, "Correct refactoring of concurrent java code," in *ECOOP 2010 – Object-Oriented Programming* (T. D'Hondt, ed.), (Berlin, Heidelberg), pp. 225–249, Springer Berlin Heidelberg, 2010.

[29] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim, "AURA: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010* (J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, eds.), pp. 325–334, ACM, 2010.

[30] K. Huang, B. Chen, L. Pan, S. Wu, and X. Peng, "REPFINDER: finding replacements for missing APIs in library update," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pp. 266–278, IEEE, 2021.

[31] S. Lee, R. Wu, S. Cheung, and S. Kang, "Automatic detection and update suggestion for outdated API names in documentation," *IEEE Trans. Software Eng.*, vol. 47, no. 4, pp. 653–675, 2021.

[32] M. Fazzini, Q. Xin, and A. Orso, "Automated API-usage update for Android apps," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019* (D. Zhang and A. Møller, eds.), pp. 204–215, ACM, 2019.

[33] S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automatic Android deprecated-API usage update by learning from single updated example," in *ICPC'20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pp. 401–405, ACM, 2020.

[34] S. A. Haryono, F. Thung, D. Lo, L. Jiang, J. Lawall, H. J. Kang, L. Serrano, and G. Muller, "AndroEvolve: automated android API update with data flow analysis and variable denormalization," *Empir. Softw. Eng.*, vol. 27, no. 3, p. 73, 2022.

[35] E. Visser, "Program transformation with Stratego/XT. Rules, strategies, tools, and systems in Stratego/XT 0.9," Tech. Rep. UU-CS-2004-011, Department of Information and Computing Sciences, Utrecht University, 2004.

[36] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider, "Source transformation in software engineering using the TXL transformation system," *Information and Software Technology*, vol. 44, no. 13, pp. 827 – 837, 2002.

[37] M. O'Keeffe and M. Cinnéide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 345–364, 2008.

[38] M. O'Keeffe and M. Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502 – 516, 2008.

[39] R. Khatchadourian, J. Sawin, and A. Rountev, "Automated refactoring of legacy Java software to enumerated types," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pp. 224–233, 2007.

[40] A. Christopoulou, E. Giakoumakis, V. E. Zafeiris, and S. Vasiliki, "Automated refactoring to the strategy design pattern," *Information and Software Technology*, vol. 54, no. 11, pp. 1202 – 1214, 2012.

[41] S.-U. Jeon, J.-S. Lee, and D.-H. Bae, "An automated refactoring approach to design pattern-based program transformations in Java programs," in *Asia-Pacific Software Engineering Conference (APSEC)*, pp. 337–345, 2002.

[42] F. Steimann, "Constraint-based model refactoring," in *Model Driven Engineering Languages and Systems: 14th International Conference (MODELS)* (J. Whittle, T. Clark, and T. Kühne, eds.), pp. 440–454, Springer, 2011.

[43] F. Steimann and J. von Pilgrim, "Constraint-based refactoring with Foresight," in *ECOOP 2012 – Object-Oriented Programming: 26th European Conference* (J. Noble, ed.), pp. 535–559, Springer, 2012.

[44] F. Steimann, C. Kollee, and J. von Pilgrim, "A refactoring constraint language and its application to Eiffel," in *ECOOP 2011 – Object-Oriented Programming: 25th European Conference* (M. Mezini, ed.), pp. 255–280, Springer, 2011.

[45] R. M. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. Keller, "Efficiently refactoring Java applications to use generic libraries," in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, pp. 71–96, 2005.

[46] A. Kiezun, M. D. Ernst, F. Tip, and R. M. Fuhrer, "Refactoring for parameterizing Java classes," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pp. 437–446, 2007.

[47] Y. Kataoka, D. Notkin, M. D. Ernst, and W. G. Griswold, "Automated support for program refactoring using invariants," in *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '01, IEEE Computer Society, 2001.

[48] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring.," in *ESEC/SIGSOFT FSE*, pp. 543–553, ACM, 2013.

[49] L. Franklin, A. Gyori, J. Lahoda, and D. Dig, "LAMBDAFICATOR: from imperative to functional programming through automated refactoring," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pp. 1287–1290, 2013.

[50] C. David, P. Kesseli, and D. Kroening, "Kayak: Safe semantic refactoring to Java streams," *CoRR*, vol. abs/1712.07388, 2017.

[51] A. Cheung, A. Solar-Lezama, and S. Madden, "Optimizing database-backed applications with query synthesis," in *Conference on Programming Language Design and Implementation, PLDI*, pp. 3–14, 2013.

[52] M. Iu, E. Cecchet, and W. Zwaenepoel, "JReq: Database queries in imperative languages," in *Compiler Construction (CC)*, pp. 84–103, 2010.

[53] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (D. Grove and S. M. Blackburn, eds.), pp. 229–239, ACM, 2015.

[54] J. Lubin, N. Collins, C. Omar, and R. Chugh, "Program sketching with live bidirectional evaluation," *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, pp. 109:1–109:29, 2020.

[55] Z. Tang, M. Agarwal, A. Shypula, B. Wang, D. Wijaya, J. Chen, and Y. Kim, "Explain-then-translate: an analysis on improving program translation with self-generated explanations," in *Findings of the Association for Computational Linguistics: EMNLP 2023* (H. Bouamor, J. Pino, and K. Bali, eds.), (Singapore), pp. 1741–1788, Association for Computational Linguistics, Dec. 2023.

[56] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024.

[57] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, J. H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. H. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. B. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph, "GPT-4 technical report," 2024.

[58] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[59] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code Llama: Open foundation models for code," 2024.

[60] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "StarCoder 2 and The Stack v2: The next generation," 2024.

[61] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence," 2024.

[62] Q. Zhu, Q. Liang, Z. Sun, Y. Xiong, L. Zhang, and S. Cheng, "GrammarT5: Grammar-integrated pretrained encoder-decoder neural model for code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.