CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites

Valerio Terragni, Yepang Liu and Shing-Chi Cheung
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China
{vterragni, andrewust, scc}@cse.ust.hk

ABSTRACT

Popular Q&A sites like `StackOverflow` have collected numerous code snippets. However, many of them do not have complete type information, making them uncompilable and inapplicable to various software engineering tasks. This paper analyzes this problem, and proposes a technique CSNIPPEX to automatically convert code snippets into compilable Java source code files by resolving external dependencies, generating import declarations, and fixing syntactic errors. We implemented CSNIPPEX as a plug-in for Eclipse and evaluated it with 242,175 `StackOverflow` posts that contain code snippets. CSNIPPEX successfully synthesized compilable Java files for 40,410 of them. It was also able to effectively recover import declarations for each post with a precision of 91.04% in a couple of seconds.

CCS Concepts

•**Human-centered computing** → **Social networking sites;** •**Software and its engineering** → **Source code generation;** *Software libraries and repositories; Compilers; Runtime environments;*

Keywords

Crowd-generated snippets, developer social networks, program synthesis

1. INTRODUCTION

Increasing number of software developers collaborate and share experience over social networks. In particular, *Question & Answer websites* (Q&A) [44], such as `StackOverflow`, `CodeRanch` and `CodeProject`, are those frequently visited by developers. On these Q&A websites, developers collaborate by raising questions, sharing their solutions and rating related contents. Over time, these sites have collected large volumes of crowd knowledge. For example, as of January 2016, `StackOverflow` had indexed 11 million questions and 18 million answers [6].

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...
http://dx.doi.org/10.1145/2931037.2931058

This crowd knowledge is useful to recent software development practice [25] because many posts contain high quality *code snippets* representing solutions of programming tasks, bug fixes [15, 20, 27] or API usage examples [32, 33]. These posts are continually rated, discussed and updated by the crowd, providing valuable resources for code reuse and analysis [42]. For example, to boost productivity, developers often reuse or draw inspiration from these code snippets [25]. Recent surveys reported that developers search for code snippets in Q&A sites frequently and extensively [34, 40, 41, 42]. As a result, the popular Q&A site `StackOverflow` receives ∼500M views per month [5]. Besides code reuse, this large volume of crowd-generated code snippets can be leveraged to accomplish various software engineering goals. For example, API library developers may leverage those snippets that use their API to test their revised API implementation or to collect possible usage profiles. Such profiles can be useful for mining temporal specifications [26, 8]. Moreover, developers may analyze these snippets for debugging [15, 27] or bug fixing [20].

However, there is a major obstacle in reusing or analyzing Q&A code snippets. The majority of Q&A posts (91.59% of the 491,906 posts we collected from `StackOverflow`) contain uncompilable code snippets [15, 45], indicating that they are non-executable and semantically incomplete for precise static analysis [26]. This phenomenon occurs because code snippets on Q&A sites are written for illustrative purposes, where compilability is not a concern. In fact, Q&A sites rarely check the syntax of submitted snippets. The problem is further exacerbated by the fact that submitted snippets are often concisely written without implementation details in order to convey solution ideas at high level [29]. Although the missing implementation details could be synthesized manually, it is tedious and often requires substantial familiarity with various libraries. Such manual synthesis effort is, however, not scalable for those crowd-based software engineering goals that typically involve many code snippets [26, 15, 27, 20].

Analyzing the uncompilable snippets collected from 450,557 `StackOverflow` posts, we observed that the most common compilation error (38%) is `compiler.error.cant.resolve`, which is a consequence of missing declarations of program entities (e.g., classes, variables and methods). This observation is in line with Chen et al.'s finding [15].

Two technical challenges have to be addressed to automatically resolve errors of this kind. First, it is difficult to resolve external dependencies, e.g., inferring an appropriate type for referred classes [46, 38, 18]. This is because the majority of code snippets refer to library classes using simple names [18].

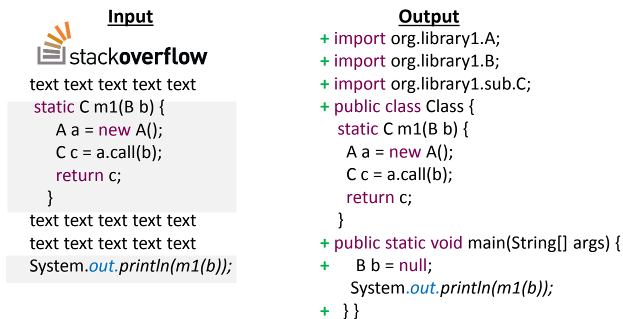


Figure 1: Example of a Q&A code snippet synthesis

Without fully qualified names (e.g., `org.library1.sub.C`), naming ambiguities can easily arise, as many simple class names across different libraries collide with one another [46, 38]. Recently, Subramanian et al. have attempted to address this issue by leveraging fields, method signatures and scope to recover the links between source code in Q&A sites and API documentations [46]. However, the technique rarely provides sufficient information to help users choose from a set of candidates, since method name conflicts are also common. For example, Dagenais et al. showed that 89% of method names in widely-used libraries are ambiguous and on average each method has name conflicts with 13 others [18]. Second, it is also hard to automatically partition multiple code snippets in a post into an appropriate number of source files. Simple heuristics such as always or never merging the code snippets in a post into a single source file can hardly cope with the complex real-world situations.

This paper presents a code snippet synthesis framework CSnippEx (C*ode* Snipp*et* Ex*tractor*) and its implementation for the Java language. CSnippEx can automatically synthesize a working context by converting the code snippets extracted from a post into Java source files, properly including library JARs in the build path, and generating import declarations for referred classes. Once the working context is set up, CSnippEx leverages *Eclipse Quick Fix* to correct bad syntax, typos, and undeclared variables. CSnippEx addresses the above-mentioned challenges as follows.

First, CSnippEx adopts a feedback-directed approach that leverages compiler error messages to automatically infer how to partition code snippets into multiple Java source files and whether the dependency resolution is correct. Second, to efficiently find the correct import combinations in a myriad of candidate solutions, CSnippEx leverages a key observation confirmed by analyzing ∼18 million real-world Java source files. The observation, which we call *clustering hypothesis*, is that the referred library classes in a Java source file often come from the same libraries and hence their import declarations tend to share common package names and form clusters (e.g., `A`, `B`, `C` and their import declarations in Figure 1). The hypothesis enabled us to design a linear time algorithm for recovering import declarations for a code snippet.

We implemented CSnippEx as an Eclipse plug-in and evaluated it with 242,175 `StackOverflow` posts that contain code snippets. Given a short time budget, CSnippEx successfully synthesized compilable Java files for 40,410 posts. Code snippet merging and dependency resolution on average only took 19 and 61 milliseconds. CSnippEx was also able to effectively recover import declarations with a precision of 91.04%, outperforming the existing API link recovering technique that relies on the oracle of method signatures [46].

In summary, this paper makes four major contributions:

- A technique CSnippEx to automatically convert Q&A site code snippets into compilable Java source code files by fixing missing declarations of classes, methods and variables.
- We implemented CSnippEx as an Eclipse plug-in and evaluated it by extensive controlled experiments. The results confirmed the effectiveness, efficiency of CSnippEx.
- We conducted statistical analysis on ∼18M real-world Java source code files [19] to demonstrate that the *clustering hypothesis* exploited by our technique is valid in practice.
- We release CSnippEx and our synthesized code snippets [3]. Our tool and dataset can facilitate future research on analyzing crowd-generated big data by various static and dynamic code analysis techniques.

2. BACKGROUND AND MOTIVATION

2.1 Problem Formulation

The input of CSnippEx is a Q&A post (a unit of text) that contains one or more *code snippets*, i.e., a collection of Java source code lines. A Q&A post could contain developers' communications in the form of natural language. On popular Q&A sites, the code snippets are mostly enclosed between dedicated HTML tags (e.g., `<pre><code>` on `StackOverflow`), and thus can be unambiguously identified.

The output of CSnippEx is one or more **compilation units**. A compilation unit (**c-unit** for short) is a legitimate Java source code file, which can be accepted by standard compilers. A c-unit declares Java reference types whose implementation is given by the source code lines in the code snippets (see Figure 1). Every c-unit and all its declared types belong to a namespace or *package*. There are two ways in which the code of a c-unit can refer to a Java type: by its *simple name* (e.g., `List`), or by its *fully qualified name* (e.g., `java.util.List`), which is the concatenation of its package name, a "." token, and its simple name. Types in the same package can refer to each other by simple names. To refer to external types defined in other packages (e.g., a third-party library), *import declarations* are needed to allow Java compilers to unambiguously resolve the referred types. It is worth mentioning that Java allows two kinds of (non-static) import declarations: the *single-type-import* declaration, which imports a single type into a c-unit by specifying its fully qualified name (e.g., `import java.util.List;`), and the *on-demand-type-import* declaration, which imports all the public types of a package into a c-unit (e.g., `import java.util.*;`). A c-unit also has a *name*, which needs to be identical to the name of its declared public class or interface type (if any).

We refer to the set of all c-units synthesized for a given Q&A post as a **compilation group** (**c-group** for short). Each c-group is associated with a *build path*, which contains the declarations of external types referred by the c-units (usually in the form of JAR archives). The build path needs to be correctly configured because in a statically typed language like Java, the declaration of each referred type has to be available at compile time.

Our research problem is how to automatically convert a Q&A post to a c-group that can be successfully compiled by standard compilers without any errors.

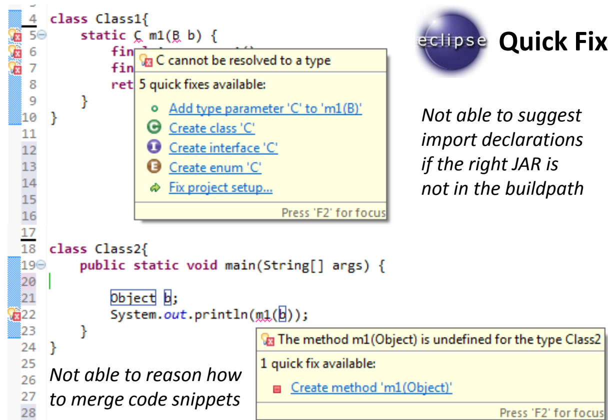


Figure 2: Quick suggestions of IDE like Eclipse.

2.2 Problem Analysis Based on StackOverflow

Achieving a successful conversion is challenging, as code snippets are, by definition, *incomplete fragments of code* [46]. They might not be enclosed in methods or class declarations, they might reference external types by simple names and they might refer to undeclared variables (see Figure 1).

To understand the research problem and quantify to what extent code snippets in Q&A sites are incomplete, we analyzed the Java code snippets in `StackOverflow`, which is the most popular programming Q&A site [24]. Our goal is to determine how many Q&A posts can be trivially converted into *compilable* c-groups. To obtain these results, we constructed a *baseline synthesis* technique described in the following.

Dataset. We conducted our analysis on the latest `StackOverflow` data dump [7], which contains 907,226 questions tagged with `Java` and 1,660,039 posts answering these questions. To select high quality code snippets (low quality ones may be of less interest to real-world developers), we restricted our analysis to 1,103,464 answer posts that are either accepted by the original questioner or have a score of at least one. Among these posts, 44.58% (**491,906**) contain Java code snippets. We identified them by extracting the text between the code block HTML tags, i.e., `<pre><code>`.

Code wrapping. In a well-formed c-unit, no *dangling* method declarations or statements are allowed. That is, they must be enclosed in a class or interface declaration. Among the 491,906 posts, we observed that only 22.87% (112,493) contain class or interface declarations, 21.34% (104,983) contain dangling method declarations, and the majority of posts 55.79% (274,430) contain dangling statements, which are not embedded in any class or method declarations. To convert each code snippet in a post into a well-formed c-unit, in the baseline approach, we generate a synthetic public class to embed dangling method declarations and statements (if any). The dangling statements are first embedded into the `main` method before being embedded into the synthesized class.

Baseline synthesis. From the 491,906 posts, we observed that 35.71% (175,653) contain multiple code snippets like the example in Figure 1. Following Subramanian et al.'s work [45], which analysed Android code snippets in `StackOverflow`, for the baseline synthesis we considered each code snippet in a post as an individual c-unit. For each c-unit, we also include all import declarations that are present in the corresponding code snippet. Following the Java language specification, we name each c-unit with the name of the declared top level public type (if any). If such a public type

Table 1: Top 3 of the 3,905,444 compilation errors

Top@	Error code		freq.	%
	`compiler.err.cant.resolve`			
		class	950,324	24.33%
Top 1		variable	484,035	12.39%
		method	50,677	1.30%
		others	590	0.02%
		total	1,485,626	38.04%
Top 2	`compiler.err.expected`		1,188,663	30.44%
Top 3	`compiler.err.not.stmt`		256,926	6.58%

does not exist, we will give the c-unit an arbitrary name. As we mentioned earlier, all such synthesized c-units for a post will form a c-group. Then, we orgnize all c-units in a c-group into the same package so that they can refer to each other by simple names. The build path of each c-group is initialized to include Java JDK 1.7 and Android-20 SDK. If a c-unit contains import declarations, we automatically query the *Maven Central Repository* [1] to download and include necessary library JARs in the build path, which can be unambiguously identified due to Java package naming conventions [9] being widely adopted. In case the library JARs have multiple versions, we select the latest one, since libraries usually guarantee backward compatibility [12].

Results. From the 491,906 posts, we obtained 764,258 c-units, which in total contain 10,701,347 (non-blank) lines of code (LOC). On average, each c-group has 1.55 c-units (median is 1) and 21.75 LOC (median is 15). We then compiled the obtained 491,906 c-groups into bytecode class files using the *javac* tool [21]. We observed that only **8.41%** (41,349) can be successfully compiled without any errors. We analyzed the 3,905,444 compilation errors reported by the *javac diagnostic* tool. Table 1 lists the top three most common errors grouped by error code prefix. The most common error (38%) is `compiler.error.cant.resolve`, which is a consequence of missing declarations. We further classified these errors according to the types of the symbols, of which the declarations are missing. Table 1 provides the breakdown. We can see that in many cases, the compilers failed to resolve the types of clasess (24.33%) and variables (12.39%). This classification is partially performed by the *javac diagnostic*, using heuristics. For example, if the symbol `A` cannot be resolved in `A a;`, the *javac diagnostic* considers `A` as a class, while in case of `A.a`, the symbol `A` can be either a class or a variable. For the latter ambiguous case, we use *the Java naming conventions* [9]. For example, if a symbol name contains an "_" or it starts with a lower case char, we consider it as a variable. The second and third most common errors are generic ones that likely result from broken code snippets (e.g., those containing natural language, pseudo-code, place holders). These errors can hardly be automatically fixed.

Our research scope. In light of these findings, we narrow the scope of our research problem on the most common error type, i.e., missing declaration errors.

2.3 Limitations of IDE Quick Fix Tools

Modern Integrated Development Environments (IDE) (e.g., `Eclipse`, `IntelliJ IDEA`) offer full-fledged and sophisticated techniques (*Quick Fix*) that offer suggestions and automated fixes to correct bad syntax, complete partial expressions, suggest and insert import declarations. Unfortunately, these techniques cannot effectively recover missing type information in Q&A code snippets. Let us use the example in Figure 1 to illustrate the limitations of Quick Fix tools.

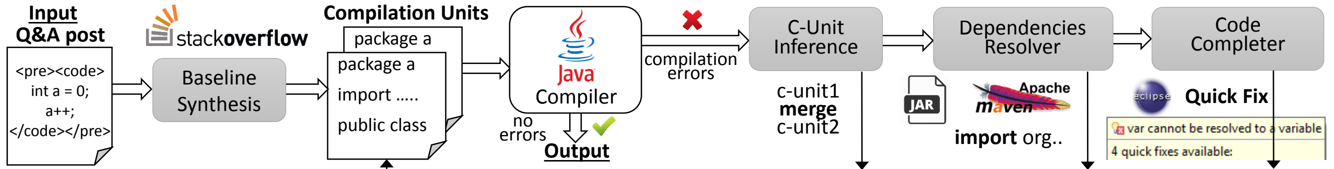


Figure 3: Overview of CSnippEx

The `Input` column of Figure 1 shows an example post containing two Java code snippets without class (i.e., `A`, `B`, and `C` in first code snippet) and variable (i.e., b in the second code snippet) declarations. The `Output` column of Figure 1 gives the corresponding compilable Java code. Here, let us assume the three concerned classes `A`, `B` and `C` are defined by some external sources (e.g., in a library JAR). Despite wrapping the code snippets in class and method declarations, their compilation leads to `cannot.resolve` errors.

1) Cannot find symbol classes `A`, `B` and `C`. Quick Fix tools fail to suggest the right import declarations if the appropriate JAR is not in the build path. Figure 2 shows the suggestions of Eclipse Quick Fix for the code snippet in Figure 1. The code completions that it suggests are to create an empty class, interface or enum with that name. This will resolve the compilation error but mock declarative completeness, compromising the usefulness of the synthesized code. Even if the right JAR is included in the build path, due to name ambiguity problems of class names [46], there could be many candidate import declarations with the same class name. Quick Fix cannot help decide which one to generate.

2) Identifier `b` cannot be resolved to a variable. Quick Fix can automatically generate the missing declaration of variables, for example line 21 in Figure 2. However, without knowledge of the type of variable `b`, it ends up creating a generic type `java.lang.Object`.

3) The method `m1(Object)` is undefined. To fix this bug, the two code snippets should be merged in the same class. Such a resolution is not supported by Quick Fix, which leaves the decision of how to construct a c-unit to developers. The code completion that it suggests is again to mock declarative completeness by creating method `m1`.

2.4 Challenges

Addressing the above-mentioned compilation errors automatically is challenging.

First, although creating synthetic class and method declarations is trivial, it is hard to automatically infer whether the multiple code snippets in a post should be merged to the same c-unit/class. A strategy that always synthesizes each code snippet as an individual c-unit can lead to compilation errors as occurred at line 22 in Figure 2. On the other hand, a strategy that always merges code snippets in a post to one c-unit might lead to `already.defined` errors.

Second, it is often difficult to automatically identify which libraries a Q&A post refers to [46, 38, 45]. This is because the majority of code snippets refer to library classes using simple names. We found that only 6.88% (33,833) of the posts we extracted contain import declarations. A simple name provides insufficient information to link it to the correct fully qualified type name [46] as a unqualified name can match many fully qualified type names in different libraries [46, 38]. Identifying the right external dependencies in the absence of fully qualified names is a major challenge for correct code snippet synthesis.

3. CSNIPPEX OVERVIEW

CSnippEx is designed and built to synthesize compilable Java source code files from Q&A code snippets. Figure 3 gives an overview of CSnippEx. First, it obtains a c-group by applying the baseline synthesis (as explained in Section 2.2) to a given Q&A post. Then it iteratively refines the c-group within a given time budget until the c-group can be successfully compiled. In a nutshell, CSnippEx adopts a *feedback-directed approach* that guides the synthesis of code snippets based on the feedbacks returned from the Java compiler, in the form of compilation error messages. In each iteration, it refines the c-group based on the compilation errors returned in the previous iteration. It then compiles the refined version of the c-group and the resulting compilation errors, if any, will be an input for the next iteration. Specifically, CSnippEx comprises three components to infer c-units, recover dependencies and support Eclipse Quick Fix.

- **C-Units Inference** (Section 4). This component resolves those missing type declarations that can be fixed by merging multiple code snippets in a post. It automatically partitions these multiple code snippets into c-units, while leveraging the feedback from the Java compiler to decide whether some code snippets should be merged.
- **Dependencies Resolver** (Section 5). This component resolves the external dependencies of a c-group. It automatically identifies the appropriate external libraries, downloads the JAR archives, and generates import declarations. It achieves this goal by using the synergy of our *clustering hypothesis* (Section 5.1) and the feedback-directed approach. The former helps identify the most suitable import declarations, while the latter helps refine the solution by excluding those import declarations that lead to compilation errors.
- **Code Completer** (Section 6). This component resolves compilation errors caused by bad syntax, typos and missing variable declarations. It automatically applies code completions to fix compilation errors based on the suggestions of Eclipse Quick Fix.

Note that the order in which these three components are executed is important. As discussed in Section 2.3, applying Quick Fix before the first two components can only mock declarative completeness. C-units inference and dependency resolution, the main contribution of this work, are required to prepare the *working environment* for *Code Completer*.

4. C-UNITS INFERENCE

In this section, we explain how CSnippEx automatically partitions multiple code snippets of a Q&A post into appropriate c-units. The high level idea is to merge code snippets together if they do not *conflict* with each other, i.e., merging them does not lead to `already.defined` errors. For instance, when dealing with the missing declaration error at line 22 of Figure 2, CSnippEx would merge the two code snippets into the same class.

```
input  : ⟨cs_i⟩ a sequence of code snippets from a Q&A post
output : c-group, a collection of c-units
1  current ← null
2  for each code snippet cs_i do
3      errors ← javac wrap(cs_i)
4      if compiler.err.already.defined ⊂ errors then
5          cs_i ← renaming(cs_i)
6      merge ← true
7      if cs_i is a class declaration then
8          merge ← false
9      else
10         temp ← current ∪ wrap(cs_i)
11         errors ← javac temp
12         if compiler.err.already.defined ⊂ errors then
13             merge ← false
14     if merge = true then
15         current ← temp
16     else
17         if current ≠ null then add current to c-group
18         current ← wrap(cs_i)
19 if current ≠ null then add current to c-group
```

Algorithm 1: C-Units Inference

```
1  Function recover (c-group)
2      errors ← javac c-group
3      if errors = ∅ ∨ time-out then
4          return c-group
5      for each c-unit cu ∈ c-group do
6          for each e ∈ errors(cu) do
7              if e = compiler.err.cant.resolve class t then
8                  T_cu ← T_cu ∪ t
9          cu.imports ← findImports(T_cu, errors)
10         T^F ← T^F ∪ cu.imports
11     c-group.buildPath ← getJarFromImports(T^F)
12     return recover(c-group);
```

Algorithm 2: Dependencies resolver

Algorithm 1 illustrates the c-units inference process. It takes as input a sequence of code snippets $\langle cs_i \rangle$ from a Q&A post, where the index i represents the order in which they appear in the post. The algorithm outputs a c-group synthesized from $\langle cs_i \rangle$. When merging code snippets, instead of trying all possible combinations, we leverage an observation that the order in which code snippets appear in a post is likely to follow some rational order. Odd combinations like cs_1 and cs_3 belonging to a same c-unit while cs_2 and cs_4 belong to another one are unlikely to occur in practice. In view of this, the algorithm performs a linear and sequential scan of $\langle cs_i \rangle$, and for each cs_i, it checks whether it should be merged into the latest c-unit or it should form its own c-unit. We now describe the whole process in detail.

Before inferring c-units, CSnippEx resolves any *"symbol σ already defined at line ϵ"* compilation errors that appear when compiling each code snippet cs_i in isolation (lines 3–5). σ could be a variable, a method or a reference type. Although these errors are uncommon in our dataset (0.03%), they must be fixed because CSnippEx relies on such compiler error messages to identify wrong compositions of c-units. If the `already.defined` errors appear when compiling a code snippet in isolation, it would jeopardize the c-units inference. To detect such errors, CSnippEx wraps each cs_i inside synthetic method and class declarations (when cs_i contains dangling method declarations or statements), and compiles the resulting c-unit (line 3). If these errors are encountered, CSnippEx performs renaming operations on cs_i to fix them. Specifically, for each duplicated declaration *"symbol σ already defined at line ϵ"*, it replaces σ with σ_1 for all the occurrences of σ in cs_i at the declaration line ϵ and all subsequent lines. The rationale of this choice is that all references after line ϵ are likely to refer to the latest declaration at line ϵ, rather than older ones before line ϵ.

The c-units inference process starts by checking the type of cs_i. If it is a (non-private) class declaration, the algorithm always decides not to merge cs_i with the current c-unit (line 8). CSnippEx makes this choice for two reasons. First, (non-private) class declarations form a natural boundary of c-units, as usually each c-unit is associated with a single class declaration. Second, all c-units in a c-group belong to the same package. In Java, all (non-private) classes declared in the same package are visible to each other, regardless whether they are declared in the same c-unit or not [21]. Therefore, merging cs_i with the current c-unit makes no influence on our goal of making the c-group compilable, but may potentially make our algorithm less efficient. If cs_i is not a class declaration, CSnippEx creates a *temp* c-unit by merging cs_i with the *current* c-unit (line 10). Specifically, if cs_i is a method declaration, it is enclosed in the latest top-level class declaration in *current*. Otherwise, if cs_i is a collection of dangling statements, it is appended to the end of the `main` method of *current*'s latest top-level class. Next, CSnippEx compiles *temp* and if it encounters any `already.defined` error, it decides that *temp* is not a valid composition of code snippets (line 13).

After processing each code snippet, CSnippEx checks the decision made: if *merge* is true, it updates *current* with *temp*, otherwise it saves the current c-unit and creates a new one enclosing cs_i. All such inferred c-units in the end are added to the same c-group.

5. DEPENDENCIES RESOLVER

Algorithm 2 describes how CSnippEx automatically resolves the dependencies to external types. Given a c-group, the algorithm returns a *dependency context* for this c-group, i.e., the necessary import declarations for each c-unit (if missing) and the build path containing the JAR archives of the recovered external libraries. For each c-unit *cu* in the c-group, our algorithm scans its `cant.resolve` errors and collects the set of missing references to external *class* types: $T_{cu} = \{t_i\}$ (lines 7–8). Note that multiple references to the same type are considered only once, and if a type is referred to by both its simple name and fully qualified name, the former would be removed from T_{cu}. To infer if t_i refers to a class type, our algorithm relies on the Java compiler error messages and Java naming conventions (as discussed in Section 2.2). Note that it is generally not possible to statically determine all missing types in a single pass of compilation [30]. This is mainly because of transitive dependencies. Besides, some type errors can also be masked by other errors and only appear when other missing types have been resolved. For this reason, Algorithm 2 is a recursive process. In each iteration, it checks for new `cant.resolve` errors and updates T_{cu}.

To recover the missing dependencies, CSnippEx relies on a **Type Repository (TR)** constituted by 123,591 pairs of fully qualified names and the latest version of the corresponding JAR archive. As Java naming conventions are widely adopted, such mappings are unique. We build TR by using the popular *Maven Central Repository*, which contains a large set of commonly used library artifacts. Let CT_i denote the set of *candidate fully qualified names* for a type $t_i \in T_{cu}$. To facilitate subsequent discussion, let

us denote a type t's simple name as t^S and fully qualified name as t^F. For a simple name $t_i^S \in T_{cu}, CT_i$ is defined as $\{t^F \mid t^F : p.t_i^S \in TR\}$, where p is the package of t^F. For a fully qualified name $t_i^F \in T_{cu}, CT_i$ is a singleton set $\{t_i^F\}$, if $t_i^F \in TR$. Let $\mathcal{I}$ denote the set of all possible dependency resolving solutions for types $t_1, \ldots, t_n$ in T_{cu}. Then $\mathcal{I}$ can be defined as the n-ary Cartesian product of sets $CT_1, \ldots, CT_n$, i.e., $\mathcal{I} = \{\langle t_1^F, \ldots, t_n^F \rangle \mid t_i^F \in CT_i\}$. For each c-unit cu in a given c-group, our algorithm returns a possible solution (i.e., finding import declarations) contained in $\mathcal{I}$ (line 9).

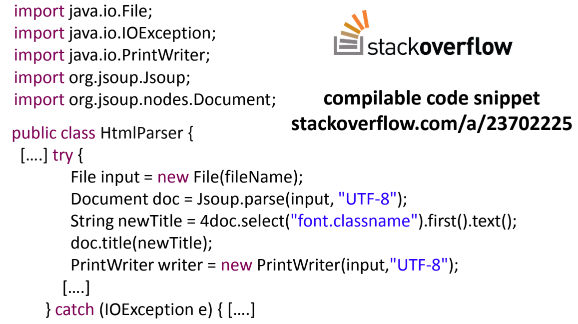
Build path recovering. After processing all c-units in the c-group, the algorithm queries the `Maven` repository to download the JAR archives that should be included in the build path (line 11). Note that each c-unit may contain references to multiple libraries, and the same import declarations might be shared across mutilple c-units of the same c-group. As such, CSnippEx identifies the minimum number of JAR archives that can cover all elements in T^F (i.e., the set of all import declarations obtained from the current iteration). It achieves this by using a greedy approximation algorithm of the set covering problem [16]. CSnippEx then starts a new iteration of Algorithm 2 (line 12) until there are no compilation errors or time is out (lines 3–4).

The combinatorial explosion challenge. Our algorithm aims to find the correct import declarations in $\mathcal{I}$ (line 9). Unfortunately, this is not an easy task. In practice, the situation like Figure 4 is rare. In most cases (92.3% of the 491,906 posts in our dataset), import declarations are missing for code snippets in Q&A posts. If the code snippets refer to external types by simple names, naming ambiguities can easily arise due to the collision of simple names across libraries [46, 38]. For example, let us assume that the code snippets in Figure 4 were originally posted without any import declarations. Our algorithm at line 8 would collect $T_{cu} = \{t_1^S = \texttt{File},\ t_2^S = \texttt{Document}, t_3^S = \texttt{Jsoup},\ t_4^S = \texttt{PrintWriter},\ t_5^S = \texttt{IOException}\}$. According to our TR, the cardinality of the corresponding sets of candidate fully qualified names would be $|CT_1| = 10, |CT_2| = 97, |CT_3| = 1, |CT_4| = 1, |CT_5| = 14$. Then in this simple case with even two singleton sets, the cardinality of $\mathcal{I}$ ($|CT_1| \times |CT_2| \times \cdots \times |CT_n|$ according to *rule of product* [39]) is 13,580. On average, in our experiments, the cardinality of $\mathcal{I}$ is 2.51×10^{34}. Clearly, finding a solution in such a huge search space is challenging. To address this challenge, CSnippEx exploits a property of the import declarations that we hypothesize to be often satisfied in real-world programs.

5.1 Clustering Hypothesis

Import declarations in the same compilation unit likely form clusters, each of which refers to a package or sub-package.

Consider the code snippet in Figure 4. There are five import declarations forming two clusters. The first three import declarations refer to the same package `java.io`, while the remaining two refer to the same sub-package `org.jsoup`. Our insight of the clustering hypothesis is that multiple imported type declarations in a c-unit often refer to the same (sub-)packages because the imported types from the same package more likely interact with one another than with those from other packages. For example, the method call `parse()` in Figure 4 returns an object of type `org.jsoup.nodes.Document` that shares the same sub-package with the type on which the method is called, i.e., `org.jsoup.Jsoup`. Similar observation




```
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;

public class HtmlParser {
 [....] try {
        File input = new File(fileName);
        Document doc = Jsoup.parse(input, "UTF-8");
        String newTitle = 4doc.select("font.classname").first().text();
        doc.title(newTitle);
        PrintWriter writer = new PrintWriter(input,"UTF-8");
      [....]
   } catch (IOException e) { [....]
```

Figure 4: Example of compilable code snippet.

can be made on the other cluster. This hypothesis can help CSnippEx efficiently identify correct import declarations from a myriad of possible solutions.

To validate our hypothesis, we performed a large-scale statistical analysis on real-world software repositories using the web service BOA [19]. Software repositories are suitable for our statistical analysis on import declarations because they usually contain compilable c-units. We considered the latest BOA dataset, which is composed by the snapshots of `GitHub` at September 2015 and `Sourceforge` at September 2013. This dataset contains ∼31 million Java c-units with a total number of import declarations of ∼198 millions.

From the analysis, we first observed that 4.48% of all import declarations are *on-demand-type-import*, which is a discouraged coding style [37]. The observation provides some support of our hypothesis because the imported types indeed belong to the same package or sub-package. To further validate our hypothesis, we performed a *Single-Linkage Hierarchical Clustering* [22] on the import declarations. Our goal is to test if indeed the import declarations of the same c-unit form clusters naturally. To identify the clusters, we propose a dedicated similarity metric on package names because standard metrics on strings are inadequate in our context. For example, the *edit distance* [23], which measures the minimum number of operations required to transform one string to the other, would attribute distances stochastically, based on textual similarity. For a given package p, let $len(p)$ be its length, which is given by the number of package levels that p prescribes. Recall that a package uses a hierarchical pattern name, where levels in the hierarchy are separated by dots ("."). We use p^i to denote the *i-th* level of p, where $i \in [1, len(p)] \cap \mathbb{N}$. For example, given p=`java.util.regex`: $len(p)$=3, p^1=`java`, p^2=`java.util` and p^3=`java.util.regex`. Let us define the distance between two packages p_a and p_b as the length of the longest uncommon suffix. More formally, $d(p_a, p_b) = max\{len(p_a), len(p_b)\} - k$, where k is the length of the longest (in terms of levels) common prefix, i.e., the largest natural number between 1 and $min\{len(p_a), len(p_b)\}$ such that $p_a^k = p_b^k$. For instance, d(`java.util`, `java.util`)= 0, d(`java.util`, `java.util.regex`)= 1.

Since there is no prior knowledge of the number of expected clusters (which is exactly what we want to find), we adopt a density threshold $\tau \in \mathbb{N}^+$. Given $I \in \mathcal{I}$, the import declarations of a c-unit and threshold τ, let $\mathcal{P}_I^\tau$ denote a *partition* $\{I_1, I_2, \ldots, I_n\}$ of I such that each pair of import declarations in the same subset (cluster) has a distance less than τ, i.e., for each $i = 1, 2, ..., n$, $(\forall j, w \in I_i)\ \big(d(j, w) < \tau\big)$.

The **heterogeneity degree HD** of the import declarations I with threshold τ is defined as follows:

$$HD_I^\tau = \frac{|\mathcal{P}_I^\tau|}{|\mathrm{I}|} 100 \in [100;0) \quad (1)$$

where $|\mathcal{P}_I^\tau|$ denotes the number of clusters and $|I|$ denotes the total number of import declarations of a given c-unit. For example, let us consider the five import declarations in Figure 4 ($|I| = 5$). For $\tau = 1$, three clusters are formed (i.e., $|\mathcal{P}_I^1| = 3$), thus HD_I^1 is 60. For $\tau = 2$, two clusters are formed (i.e., $|\mathcal{P}_I^2| = 2$), thus HD_I^2 is 40. Note that for any import declarations I, $|\mathcal{P}_I^\tau| \leq |I|$ as it is impossible to have more clusters than the number of import declarations, and $|\mathcal{P}_I^\tau| \geq 1$ as at least one cluster must be formed. Therefore, $HD_I^\tau \in [100; 0)$. HD_I^τ has the maximum value of 100 when each import declaration in a c-unit forms its own cluster. The lower the number of clusters is, the lower the value of HD_I^τ is. HD_I^τ attains the lowest value when $|\mathcal{P}_I^\tau| = 1$. When this occurs, we have the minimum heterogeneity degree as all import declarations are grouped in the same cluster. Thus, a low value of HD_I^τ supports our clustering hypothesis.

We computed the HD_I^τ of those c-units in the BOA dataset with at least two single-type and without any on-demand-type import declarations in I (∼18 millions c-units). We choose values of $\tau = 1, 2, 3, 4$ because the average number of levels per import declarations in the dataset is ∼4. Note that the lower the value of τ is, the stricter the clustering criteria is. For example, with $\tau = 1$, import declarations are grouped in the same cluster if and only if they have an identical package name (i.e., $d(p_a, p_b) = 0$). Table 2 shows the results. We can observe that the average and median is far less than 100, supporting our clustering hypothesis. In addition, Figure 5 shows the average values of HD^τ according to the number of imports declaration per c-unit ($|I|$). The heterogeneity degree HD decreases when $|I|$ increases. This is because: the higher the number of import declarations is, the higher the probability that they form clusters is.

Table 2: Average, median of HD_I^τ with τ = 1,2,3,4 of the ∼18 millions c-units in the BOA data-set[19]

	$\mathbf{HD^1}$	$\mathbf{HD^2}$	$\mathbf{HD^3}$	$\mathbf{HD^4}$
average	62.00	44.44	34.64	27.68
median	60	40	28.57	22.22

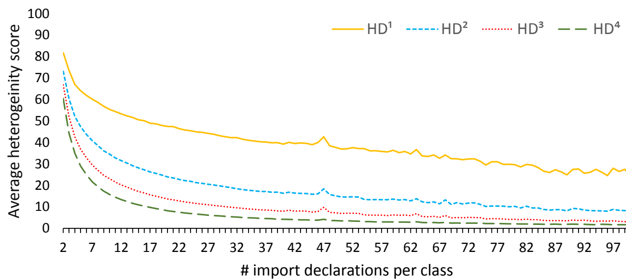


Figure 5: Average of HD_I^τ according to number of import declarations per c-unit/class ($|I|$)

5.2 Recovering Import Declarations

CSnippEx leverages the clustering hypothesis to prioritize the possible solutions in $\mathcal{I}$. Intuitively, those solutions that form a small number of clusters have higher priorities. However, it is infeasible to first compute HD_I^τ for each possible solution I in $\mathcal{I}$ and then rank the solutions according to HD_I^τ. This is because in practice, $\mathcal{I}$ is too huge to be exhaustively enumerated. To address this problem, we propose a greedy algorithm (Algorithm 3). The algorithm initially identifies

```
1  Function findImports(T,errors)
2    if top = null ∨ isNew(T) then
3      for each t_i ∈ T do
4        for each p : in p.t^S ∈ CT_i do
5          freq(p)++
6      for each t_i ∈ T do
7        for each t^F : p.t^S ∈ CT_i do
8          score(t^F) ← freq(p)
9        sort DESC CT_i by score, name
10     top ← ⟨CT_1[0], CT_2[0], ..., CT_n[0]⟩
11   if errors ≠ null then
12     replace top_i ⊂ errors with CT_i[1]
13   for each t_i ∈ T do
14     for each p in p.t^S ∈ CT_i do
15       freq(p)← 0
16   for each t_i ∈ T do
17     for each p in p.t^S ∈ CT_i do
18       if ∃t^F:p_1.t^S ∈ top such that p_1 = p then
19         freq(p)++
20   for each t_i ∈ T do
21     for each t^F : p.t^S ∈ CT_i do
22       score(t^F) ← freq(p)
23       for k = 1..3 do
24         if freq(p^{len(p)−k}) > score(t^F) then
             score(t^F) = freq(p^{len(p)−k})+1
25
26     sort DESC CT_i by score, name
27   return top ← ⟨CT_1[0], CT_2[0], ..., CT_n[0]⟩
```

Algorithm 3: Recovering Import Declarations

the solution that forms the highest cardinality cluster with density threshold $\tau = 1$ (line 2–10). Then (in case of unsuccessful compilation) it relies on compilation errors and higher density thresholds ($\tau = 2, 3, 4$) to further refine the solution (lines 11–27). In the following, we illustrate how our algorithm can successfully recover import declarations using the example code snippet in Figure 4. Let us assume the code snippet was originally posted with no import declarations.

Identifying the clusters with highest cardinality. If the top solution is empty and whenever new missing class declarations are found (line 2 of Algorithm 3), lines 3–5 scan each package p in $CT_1, CT_2, \ldots, CT_n$ and compute its global frequency. Then, for each candidate fully qualified name t^F: $p.t^S$ in each CT_i, the algorithm gives t^F a *score* of its package p's global frequency (lines 6–8). Next, line 9 orders all candidates in each CT_i by their score. In case multiple candidates in a CT_i share the same score, the algorithm orders them alphabetically by their package name. After ordering, the algorithm returns the *top* solution by picking the highest ranked candidate from each CT_i (line 10). The rationale of the algorithm is that by construction it guarantees that packages in *top* form clusters with the highest cardinality. Note that the alphabetic ordering is critical for guaranteeing this property, since it ensures a fixed order of candidates for each CT_i in tie cases. To ease understanding, Table 3 gives the algorithm's running result for our example. As we can see, CSnippEx identified the cluster `java.io`, which has the highest cardinality, and three correct import declarations. However, the import declaration of the type `Document` is still wrong (it should not be `org.bson.Document`). This can be fixed by the following refinement process.

Refining the solution by compilation errors. The first refinement takes into account the compilation errors

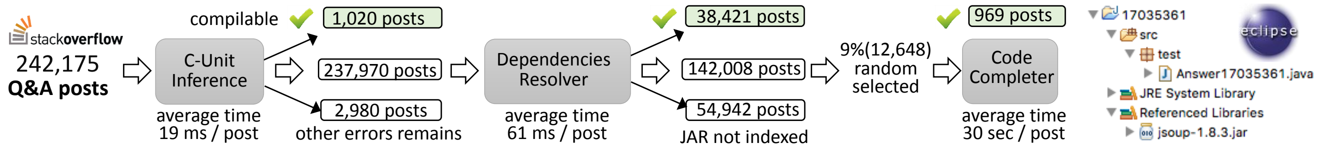


Figure 6: Experimental results of CSnippEx

returned when compiling the c-group with the dependency context given by the solution *top* returned at line 10. Note that this refinement is not applicable for our example at this stage, as a possible *dependency context* has not been generated yet (it will be generated after line 11 of Algorithm 2). The refinement works as follows. If an error (regardless of its error code) involves a fully qualified name t^F in *top*, t^F is removed from *top* (line 12 of Algorithm 3) and the next ranked (if it exists) candidate will become the highest ranked accordingly. It is worth mentioning that t^F is only suspicious at this moment (not definitively wrong), as it could be accidentally involved in a compilation error because of *other* wrongly recovered import declarations. For example, let us assume that our previous solution associated `File` in Figure 4 with a wrong fully qualified name `scala.io.File`, while `PrintWriter` was associated correctly with `java.io.PrintWriter`. The compilation error in this case would be: *the constructor java.io.PrintWriter(scala.io.File, java.lang.String) is undefined*. Although, `java.io.PrintWriter` is involved in the error, it is actually correctly resolved. If the algorithm simply discards `java.io.PrintWriter`, it would not be able to provide the correct solution in the end. Therefore, a suspicious t^F would not be definitively removed from CT_i: in case CT_i becomes empty before Algorithm 3 returns, CT_i is restored with its original (complete) form.

Refining the solution by higher density thresholds. The second refinement (lines 13–26 of Algorithm 3) computes the global frequency for only those packages that are contained in *top* (lines 16–19). Then it checks whether by considering different levels of package (i.e., different density thresholds), there exists another t^F (excluding those already in top) that can belong to a higher cardinality cluster (lines 20–26). For our running example, this refinement process can successfully help identify that candidate `org.jsoup.nodes.Document` is more suitable than `org.bson.Document`. This is because the frequency of the sub package `org.jsoup` has a higher frequency than `org.bson`.

6. CODE COMPLETER

This component systematically applies code completions proposals to fix compilation errors based on the suggestions of Eclipse Quick Fix. It excludes those suggestions that propose to remove statements or mock declarative completeness (e.g., creating an empty class). After applying each suggestion, the c-group is recompiled to check weather the concerned error has been fixed or new errors arise. Iteratively, the code completer systematically applies Quick Fix suggestions until the compilation succeeds or time is out.

Specifically, in the completion process, CSnippEx uses a Breadth First Search (BFS) strategy to identify the solution that can fix all errors with a minimum number of steps. This strategy follows the principle of theoretical parsimony *Occam's razor* [13]: the simplest solution (i.e., that involving the minimum number of steps) is the one that should be preferred. This principle is also leveraged to suggest the best Eclipse Quick Fix suggestion in related work [28].

Table 3: Running example

File	IOException	PrintWriter	Document	Jsoup
3java.io	3java.io	3java.io	1org.bson	1org.jsoup
1scala..	1com.sun..		1org.jdom2	
1org.specs..	1net.kuujo..		1org.jsoup.nodes	
....	...		....	
↖ after line 10 of Algorithm 3 ↗				
3java.io	3java.io	3java.io	2org.jsoup.nodes	1org.jsoup
....	...		1org.jdom2	
			1org.bson	
			....	
↖ after line 27 of Algorithm 3 ↗				

7. EVALUATION

In this section, we present our experimental evaluation of CSnippEx. We start by introducing implementation details.

We implemented CSnippEx as an Eclipse plug-in. To sythesize compilable code snippets, users only need to provide the URL to the corresponding `StackOverflow` post. For performance consideration, the first two components of CSnippEx analyze, modify and compile the intermediate c-groups in memory using the `javax.tools.JavaCompiler` APIs. CSnippEx only writes Java source code files to disk if they are compilable or it needs to invoke Eclipse Quick Fix via the `org.eclipse.jdt` APIs. For dependency resolution, CSnippEx automatically accesses third-party web platforms to search those fully qualified names that match the simple names of the types whose declarations are missing and download necessary JAR archives to configure build path [1].

Experimental Setup. To perform large-scale experiments, we built a local database of `StackOverflow` posts to ease data retrieval [7]. In order not to violate the *Maven Central Terms of Service* [4], which limits the number of JARs that can be downloaded in one day, we cached 3,000 JARs locally (the collection process takes months). We obtained the list of these JARs by running CSnippEx on our dataset. These 3,000 JARs are those top ones that are requested to construct our type repository. To avoid conflicts between build paths of different c-groups, each c-group is extracted as a dedicated Eclipse project. All our experiments were conducted on a Windows 7 64-bit desktop PC with an Intel® Core i3-3220 CPU and 12 GB RAM.

7.1 Synthesis Effectiveness

In our first set of experiments, we evaluated CSnippEx's effectiveness of synthesizing compilable c-groups. As discussed earlier, the baseline synthesis failed for 450,557 `StackOverflow` posts (see Section 2.2). For our experiments here, we selected 242,175 of these 450,557 posts as subjects. We selected these posts because after compiling the corresponding c-groups synthesized by the baseline approach, Java compiler reports *at least one* missing declaration error. These errors are those that CSnippEx aims to resolve. We then ran the first two components of CSnippEx and set the time budget to one second per post. Figure 6 summarizes our the results.

The C-Units Inference component on average finshed processing each post in 19.04 milliseconds (median = 15). In

total, it successfully synthesized 1,020 compilable c-groups. Moreover, it also fixed all missing declaration errors for 2,980 c-groups, but these c-group were still not compilable due to other types of errors, which are out of our study scope. For 205 c-groups, the Java compiler crashed and we removed them from our experiments. The remaining 237,970 c-groups still contained missing declaration errors, which would be further fixed by other components. Besides, with this step, the average number of c-units per c-group decreased from 1.41 to 1.09, showing that the merge operations in Algorithm 1 were successful in many cases.

The Dependencies Resolver on average finished dependency resolution for each post in 61.53 milliseconds (median = 35). It successfully synthesized 38,421 compilable c-groups. The majority of the recovered external dependencies involve third-party libraries. For example, only 1,558 of the 5,826 unique import declarations recovered by CSnippEx for the 38,421 compilable c-groups are related to JDK or Android SDK. For 54,942 c-groups, CSnippEx requested JARs that were not cached by us (recall that we only cached 3,000 popular ones for experiments) and therefore CSnippEx cannot handle them. For 2,599 c-groups, the Java compiler crashed and we removed them from our experiments. The remaining 142,008 c-groups were still not compilable after timeout. This could be caused by two reasons: (1) CSnippEx recovered a wrong dependency context, or (2) the dependency context was correctly recovered, but other errors (e.g., missing variable declarations) in the c-units prevented a successful compilation. Next, these c-groups would be further handled by the code completer of CSnippEx . For the code completion experiments, the highest ranked dependency contexts recovered by CSnippEx would be chosen.

The Code Completer component creates an Eclipse Java project for each c-group and automatically explores the code completion possibilities suggested by Eclipse Quick Fix. Instead of handling all types of errors, we focused on fixing missing variable declarations, which are within our study scope. In the experiments, we randomly selected 12,648 (9%) c-groups and launched the code completer to process them with the time budget set to one second per c-group. It is worth mentioning that we only managed to experiment on 12,648 c-groups because in order to run code completer, Eclipse needs to create and set up a Java project for each c-group. The performance of Eclipse degraded over time when more projects were created, taking ∼30 seconds for each c-group. Finally, 969 c-groups were successfully synthesized.

From the above discussions, we can see that CSnippEx effectively synthesized a large number of c-groups from `StackOverflow` posts. Of course, there also exist many posts that CSnippEx cannot handle. This is because currently CSnippEx only focuses on resovling missing declaration errors, which are the most common in our dataset.

7.2 Precision of the Dependency Resolving

In our second set of experiments, we performed an in-depth evaluation of CSnippEx's Dependencies Resolver, which is a major contribution of this paper.

Subject selection and ground truth. Although it is impossible to recover the real missing implementation details for a post with incomplete code snippets, it is possible to construct a *golden set* from those posts with complete code snippets. In our dataset, 33,735 (6.86%) posts contain import declarations, which we could remove to construct the golden set for evaluating to what extent CSnippEx is able to recover these import declarations and synthesize compilable c-groups. To avoid potential biases or conflicts due to the presence of other types of errors or missing implementation details, we first selected those posts that can be successfully handled by the baseline approach (e.g., the example in Figure 4). We further excluded: (1) 525 (1.57%) posts that only contain import declarations, but no source code lines, (2) 285 (1.87%) posts that contain *static* import declarations, which are currently not supported by our tool, and (3) 1,016 (7.03%) posts containing multiple code snippets (for these posts, we could not unambiguously associate the import declarations to a single code snippet). A total of 13,444 c-units/posts were selected for the experiments. On average, each c-unit has 6.37 import declarations (median = 5). The average size of $\mathcal{I}$ according to our type repository (TR) is 2.51×10^{34}.

Table 4: Results of the 13,444 posts in the golden-set

solution Top@	% C compile	% E equivalent	avg time (ms)	med time (ms)
Top 1	76.87%	76.30%	66.53	32
Top 10	89.66%	87.35%	103.79	47
Top 100	91.04%	88.27%	4,454.23	1,889

We then removed the user-specified import declarations from these c-units and ran CSnippEx (Algorithm 2) to recover them. In our experiments, the stopping criterion is: either the c-unit can be successfully compiled or the top 100 solutions (i.e., configurations of import declarations) in $\mathcal{I}$ are attempted. During the process, we also collected results for Top 1 and Top 10 solutions in $\mathcal{I}$.

Table 4 presents the results. Column "%C" shows the percentage of Q&A posts that were successfully synthesized into compilable c-units. Column "%E" shows the percentage of synthesized c-units whose import declarations are *equivalent* to the original user-specified ones. Note that if the original import declarations contain wild cards (e.g., `org.library.*`), we would use regular expressions to match it with single-type import declarations recovered by CSnippEx (e.g., `org.library.C`). The remaining two columns show the average and median time of the recovery process for each post, respectively.

The results confirmed the effectiveness of our clustering hypothesis in identifying the correct solutions in the huge search space of $\mathcal{I}$. For example, for 76.87% of the posts, the Top 1 solution returned by Algorithm 3 achieved successful compilation. The results of Top 10 and Top 100 are even better: for 91% of the posts, CSnippEx successfully synthesized compilable c-units after trying the Top 100 solutions. This also demonstrated the effectiveness of the refinement process in Algorithm 3. Moreover, a large percentage of compilable c-units synthesized by CSnippEx have identical import declarations as the the golden set (Column E%). This not only shows that compilability is a good proxy for correct synthesis, but also confirms that our identification of missing type information via compilation errors is precise. One may notice that the results in Columns %C slightly diviates from those in Column %E, indicating that in some cases, CSnippEx achieved a successful compilation but generated import declarations that differed from the user-defined ones. We investigated these cases and found two major reasons. First, some user-defined import declarations were never referenced in the code. Second, some projects make internal clones of third-party library code to avoid including external JARs in build path [46]. In such cases, CSnippEx would

not recover import declarations that are identical to user-specified ones. Finally, Table 4 also shows the efficiency of the recovery process. The average processing time ranges from 66.53 (for Top 1) to 4,454 milliseconds (for Top 100).

Comparison with *Baker* [46] , the state-of-the-art technique to build traceability links between source code elements and external libraries. Baker recovers the links by querying an oracle for types, methods, and fields to match the constraints imposed by a code snippet under analysis. Although Baker was not originally designed for synthesizing compilable c-units, it can help recover import declarations. Therefore, we conducted a comparison experiment to see if Baker could recover import declarations more effectively than CSnippEx.

The Baker tool is publicly available as a web service. Since the web service took minutes to process a single request, we were only able to apply Baker [2] on 4,947 (37%) of those code snippets (with import declarations removed) analyzed by CSnippEx in Section 7.2. For each code element identified as an API type, Baker gives a set of possible fully qualified names for concerned external class types. For comparison, we calculated the percentage of code snippets, for which Baker found unique matches for all concerned class types (i.e., all returned sets have a cardinality of 1). For fairness, if Baker returned multiple sets for a class type (when it appears at multiple lines in a code snippet), we would consider the set with lowerest cardinality as Baker's output. We observed from the experiment that Baker only found unique matches of external class types for 36.71% code snippets. In other cases, Baker returns ambiguous results (i.e., multiple matches for a class type). For each code snippet, we multiplied the cardinality of each class type matches returned by Baker, on average this number is 1,293,205. Although, this is an improvement over 2.51×10^{34}, it suggests that in many cases a code snippet does not contain enough discriminative references of class types to identify a unique match because the naming conflicts of fields and methods are also common [18]. This shows that Baker cannot effectively recover import declarations.

7.3 Discussion

Guarantees. It is impossible to guarantee that the synthesized c-units are semantically equivalent to the user-intended code. However, we believe that successful compilation is a good proxy to validate the correctness of the synthesized results. This is because CSnippEx obeys the following constraints: 1) It never mocks declarative completeness to resolve missing class/method declarations. 2) It never removes those lines of code that lead to compilation errors. In some cases these constraints could prevent a successful synthesis, but they give some guarantee on the usefulness of the results.

Applicability. Although this paper focuses on Java code snippets, many of the issues it addressed also apply to other *statically typed languages* (e.g., C++). While this paper focuses on code snippets found in Q&A posts, CSnippEx could also be applied to those found in tutorial sites and JavaDoc, which also enclose code snippets in dedicated tags.

8. RELATED WORK

We are not aware of any research or commercial tools that aim to synthesize compilable code snippets from non-code sources (e.g., Q&As). We now discuss the most related work.

Traceability Link Detectors. Some studies aimed to build traceability links between source code elements and external libraries [46, 45, 38]. In Section 7.2, we show that the state of the art [46] is ineffective when applied to our problem. Other pieces of work aimed to build links between natural language documents and source code elements [10, 18, 31]. These techniques are not suitable for recovering missing dependencies for code snippets. First, they try to identify code snippets from natural language text (e.g., emails), but for Q&A sites code snippets are usually well separated from text as they are enclosed by dedicated HTML tags. Second, they are not able to retrieve fully qualified names for class types unless such names are mentioned in the processed text.

Partial Program Analysis (PPA) [17] aims at generating fully-resolved abstract syntax trees in the face of missing types. However, PPA is not effective in analyzing code snippets from Q&A sites [38, 15, 45]. Partial programs are considered *complete* source files without external dependencies [17], while Q&A code snippets are *incomplete* code fragments. PPA does not try to recover missing import declarations, instead it relies on the existing ones to infer the missing types. Moreover, PPA assumes that the analyzed program is compilable given the required dependencies [17]. This assumption does not hold for Q&A code snippets.

Code Integration. There has been some work to facilitate the integration of software retrieved on-line. Ossher et al. proposed a technique to resolve dependencies for open-source software [30]. However, it is intended to work with complete source files like PPA. It relies on the fully qualified names specified by the import declarations to identify the dependencies. In contrast, CSnippEx works on incomplete snippets and identifies dependencies by simple names alone.

Code Search. There is a large body of work on improving code search in Q&A sites [26, 42, 43], and incorporating web code search into IDEs to reduce the context switching from IDEs to browsers [11, 35, 14, 36]. All such proposed techniques aim to improve the effectiveness of code search. As discussed, CSnippEx could further help extract, compile and integrate the code snippets retrieved by these techniques. Moreover, CSnippEx can precisely identify the referred libraries, which could help improve code search results.

9. CONCLUSION AND FUTURE WORK

In this paper, we presented CSnippEx to synthesize compilable Java source code files form Q&A code snippets. Our experimental results showed the effectiveness of our technique. Specifically, our greedy algorithm based on the clustering hypothesis is very effective in quickly identifying possible solutions of import declarations, while the feedback-directed approach can further refine the candidate solutions.

Compilability is only a necessary but not a sufficient condition to obtain runnable code. In future work, we plan to investigate how to synthesize executable context for the compilable code, e.g., addressing the initialization of variables, generation of files, creation of database connections. To do so, we are considering a feedback-directed technique guided by runtime exceptions.

10. ACKNOWLEDGMENTS

The research was partially funded by Research Grants Council (General Research Fund 611813) of Hong Kong and MSRA Collaborative Research Grant FY16-RES-THEME-010. We would like to thank Jeff Burkhartsmeyer for proofreading the paper.

11. REFERENCES

[1] Apache Maven Central Repository http://search.maven.org/ accessed September 2015.

[2] Baker Web Service http://gadget.cs.uwaterloo.ca:2145/snippet/index.html accessed December 2015.

[3] CSNIPPEX: http://sccpu2.cse.ust.hk/csnippex/.

[4] Maven Central Terms of Service http://repo1.maven.org/terms.html accessed September 2015.

[5] Quantcast - www.quantcast.com/stackoverflow.com accessed January 2016.

[6] Stackexchange http://stackexchange.com/sites?view=list#traffic accessed January 2016.

[7] StackOverflow Data Dump https://archive.org/download/stackexchange accessed September 2015.

[8] G. Ammons, R. Bodík, and J. R. Larus. Mining Specifications. In *POPL*, pages 4–16, 2002.

[9] K. Arnold, J. Gosling, D. Holmes, and D. Holmes. *The Java Programming Language*. Addison-wesley Reading, 1996.

[10] A. Bacchelli, M. Lanza, and R. Robbes. Linking E-mails and Source Code Artifacts. In *ICSE*, pages 375–384, 2010.

[11] A. Bacchelli, L. Ponzanelli, and M. Lanza. Harnessing Stack Overflow for the Ide. In *RSSE*, pages 26–30, 2012.

[12] I. Balaban, F. Tip, and R. Fuhrer. Refactoring Support for Class Library Migration. In *OOPSLA*, pages 265–279, 2005.

[13] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information processing letters*, 1987.

[14] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric Programming: Integrating Web Search Into The Development Environment. In *SIGCHI*, pages 513–522, 2010.

[15] F. Chen and S. Kim. Crowd Debugging. In *FSE*, pages 320–332, 2015.

[16] V. Chvatal. A Greedy Heuristic for the Set-covering Problem. *Mathematics of Operations Research*, 1979.

[17] B. Dagenais and L. Hendren. Enabling Static Analysis for Partial Java Programs. In *OOPSLA*, pages 313–328, 2008.

[18] B. Dagenais and M. Robillard. Recovering Traceability Links Between an API and Its Learning Resources. In *ICSE*, pages 47–57, 2012.

[19] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *ICSE*, pages 422–431, 2013.

[20] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing Recurring Crash Bugs via Analyzing Q&A sites. In *ASE*, pages 307–318, 2015.

[21] J. Gosling. *The Java Language Specification.* Addison-Wesley Professional, 2000.

[22] S. C. Johnson. Hierarchical Clustering Schemes. *Psychometrika*, 1967.

[23] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet physics doklady*, 1966.

[24] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann. Design Lessons from the Fastest Q&A Site in the West. In *SIGCHI*, pages 2857–2866, 2011.

[25] K. Mao, L. Capra, M. Harman, and Y. Jia. A Survey of the Use of Crowdsourcing in Software Engineering. *RN*, 15:01, 2015.

[26] A. Mishne, S. Shoham, and E. Yahav. Typestate-based Semantic Code Search Over Partial Programs. In *OOPSLA*, pages 997–1016, 2012.

[27] M. Monperrus and A. Maia. Debugging with the Crowd: a Debug Recommendation. *Technical report hal-00987395 INRIA*, 2014.

[28] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative Analysis of Integrated Development Environment Recommendations. In *OOPSLA*, pages 669–682, 2012.

[29] S. Nasehi, J. Sillito, F. Maurer, and C. Burns. What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow. In *ICSM*, pages 25–34, 2012.

[30] J. Ossher, S. Bajracharya, and C. Lopes. Automated Dependency Resolution for Open Source Software. In *MSR*, pages 130–140, 2010.

[31] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora. Mining Source Code Descriptions from Developer Communications. In *ICPC*, pages 63–72, 2012.

[32] C. Parnin and C. Treude. Measuring API Documentation on the Web. In *Web2SE*, pages 25–30, 2011.

[33] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey. Crowd Documentation: Exploring the Coverage and the Dynamics of API Discussions on Stack Overflow. *Technical Report: CrowdDoc-GIT-CS-12-05*, 2012.

[34] K. Philip, M. Umarji, M. Agarwala, S. E. Sim, R. Gallardo-Valencia, C. V. Lopes, and S. Ratanotayanon. Software Reuse Through Methodical Component Reuse and Amethodical Snippet Remixing. In *CSCW*, pages 1361–1370.

[35] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack Overflow in the Ide. In *ICSE*, pages 1295–1298, 2013.

[36] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining Stackoverflow to Turn the IDE into a Self-confident Programming Prompter. In *MSR*, pages 102–111, 2014.

[37] A. Reddy et al. Java Coding Style Guide. *Sun MicroSystems - Technical report*, 2000.

[38] P. C. Rigby and M. P. Robillard. Discovering Essential Code Elements in Informal Documentation. In *ICSE*, pages 832–841, 2013.

[39] J. Riordan. *Introduction to Combinatorial Analysis*. Courier Corporation, 2012.

[40] C. Sadowski, K. T. Stolee, and S. Elbaum. How Developers Search for Code: A Case Study. In *FSE*, pages 191–201, 2015.

[41] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes. How Well Do Search Engines Support Code Retrieval on the Web? *TOSEM*, 21(1):4, 2011.

[42] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the Search for Source Code. *ACM TOSEM*, 23(3):26, 2014.

[43] K. T. Stolee, S. Elbaum, and M. B. Dwyer. Code Search with Input/Output Queries: Generalizing, Ranking, and Assessment. *Journal of Systems and Software*, 21:35–48, 2015.

[44] M.-A. Storey, L. Singer, B. Cleary, F. Figueira Filho, and A. Zagalsky. The (R)Evolution of Social Media in Software Engineering. In *FOSE*, pages 100–116, 2014.

[45] S. Subramanian and R. Holmes. Making Sense of Online Code Snippets. In *MSR*, pages 85–88, 2013.

[46] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API Documentation. In *ICSE*, pages 643–652, 2014.