Boosting Complete-Code Tool for Partial Program

Hao Zhong
Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
Email: zhonghao@sjtu.edu.cn

Xiaoyin Wang
Department of Computer Science
University of Texas at San Antonio, USA
Email: xiaoyin.wang@utsa.edu

***Abstract*—To improve software quality, researchers and practitioners have proposed static analysis tools for various purposes (*e.g.*, detecting bugs, anomalies, and vulnerabilities). Although many such tools are powerful, they typically need complete programs where all the code names (*e.g.*, class names, method names) are resolved. In many scenarios, researchers have to analyze partial programs in bug fixes (the revised source files can be viewed as a partial program), tutorials, and code search results. As a partial program is a subset of a complete program, many code names in partial programs are unknown. As a result, despite their syntactical correctness, existing complete-code tools cannot analyze partial programs, and existing partial-code tools are limited in both their number and analysis capability. Instead of proposing another tool for analyzing partial programs, we propose a general approach, called GRAPA, that boosts existing tools for complete programs to analyze partial programs. Our major insight is that after unknown code names are resolved, tools for complete programs can analyze partial programs with minor modifications. In particular, GRAPA locates Java archive files to resolve unknown code names, and resolves the remaining unknown code names from resolved code names. To illustrate GRAPA, we implement a tool that leverages the state-of-the-art tool, WALA, to analyze Java partial programs. We thus implemented the first tool that is able to build system dependency graphs for partial programs, complementing existing tools. We conduct an evaluation on 8,198 partial-code commits from four popular open source projects. Our results show that GRAPA fully resolved unknown code names for 98.5% bug fixes, with an accuracy of 96.1% in total. Furthermore, our results show the significance of GRAPA's internal techniques, which provides insights on how to integrate with more complete-code tools to analyze partial programs.**

***Index Terms*—Partial program, program analysis, boosting complete-code tool**

I. INTRODUCTION

A partial program is a subset of a complete program. Partial-code analysis [8] is necessary in many scenarios where only partial programs are available, such as mining bug fixes for automatic patching [16] and defect prediction [14], analyzing forum threads and software documents for code recommendation [44], and ranking code-search results [4]. In our paper, we follow the definition in Dagenais and Hendren [8]'s as below, which requires partial programs to be free of syntax errors.

Definition 1 (Partial Program): Given a complete program $< Src, Dep >$*, in which* Src *is the set of compilable source files, and* Dep *is the set of compiled dependency files, a partial program* par *is a subset of* Src*.*

This definition is consistent with the majority of application scenarios such as analyzing bug fixes, code search results, and samples in documents, where source files are incomplete but unlikely to have syntax errors.

Although the syntax of partial programs is often correct, it is infeasible to compile a partial program, since the code declarations it refers to may not be available. For various purposes, researchers (*e.g.*, [40], [30], [3]) have proposed approaches that analyze partial programs. However, these approaches have three limitations. First, most partial-code tools are not general. For example, Zhong *et al.* [43] propose MAPO that mines specifications for recommending code samples. Mishne *et al.* [25] criticize that MAPO cannot mine specifications for partial programs. To handle the problem, Mishne *et al.* [25] propose PRIME that compares unknown method calls with known method calls in other call sequences, when it mines specifications. Although PRIME thus is able to mine specifications for partial programs, its techniques cannot support partial-code analysis for other purposes (*e.g.*, [40]). Second, due to the difficulties of analyzing partial programs, partial-code analysis is typically imprecise. For example, Mishne *et al.* [25] admit that their approach is only *relatively precise*. Finally, existing partial-code tools cannot support complicated analyses. For example, although graphs are informative to compare code, Kim and Notkin [17] complain that CFG-based approaches (*e.g.*, [2]) cannot analyze partial programs.

Our insight. We notice that many complete-code tools (*e.g.*, WALA[1]) are built on mature compilers (*e.g.*, Eclipse JDT). As a partial program is incomplete, a compiler typically fails to resolve unknown code names. When complete-code tools encounters unknown code names, they will fail to produce meaningful results. If we fully resolve such unknown code names, it is feasible to boost some complete-code tools to analyze partial programs.

Definition 2 (Code Name): For a given partial program P, we define *code names of* P*, denoted as* $Names(P)$ as identifiers of code elements at all granularities (*e.g.*, classes/types, fields, methods, and variables) appearing in P. Also, we define *resolving code name* N as determining the variable type and the full name of N if N is a variable/field; determining the full name of N if N is a type; and determining the signature of N if N is a method.

The benefits and challenges. Our insight leads to a novel approach with the following benefits:

[1] http://wala.sourceforge.net

TABLE I: The inference strategies of PPA.

Strategy	Example
Assignment	B.field=''Hello World'';→{field,unknown, ⋗ java.lang.String}
Return	int m(){ return method()};→{method,unknown, ⋖ int}
Method	f1=m1(f2); D m1(E p);→{f1,unknown, ⋗ D} and {f2,unknown, ⋖ E}
Condition	if(f){...};→{f,unknown, boolean}
Binary and unary operators	int i = f-10;→{f,unknown, ⋖ int}
Array	f1 = f2[f3];→{f3,unknown, ⋖ int} and {f8,unknown,unknown[]}
Switch	switch(f){...};→{f,unknown, ⋖ int}
Conditional	int i=f1?1:f2;→{f1,unknown,boolean} and {f2,unknown, ⋖ int}

Benefit 1. This is a general approach that enables many complete-code tools for partial-code analysis. Unknown code names of partial programs have different impacts on these tools. Instead of proposing a solutions for specific purposes (*e.g.*, API specification mining), our approach works on all the complete-code tools that are built on mature compilers.

Benefit 2. This is a practical way to improve the precision of partial-code analysis. As partial programs contains unknown code elements, partial-code tools are typically imprecise. Our strategy obtains precise tools for partial-code analysis, since it preserves the preciseness of complete-code tools.

Despite the above benefits, to fulfill this strategy, we shall overcome the following challenges:

Challenge 1. It is challenging to resolve unknown code names for partial programs. PPA [8] is the state-of-the-art tool that resolves code names for unknown code elements, but our study shows that PPA fully resolves all code names for only 28.7% of partial-code snippets (Section IV-C).

Challenge 2. It is challenging to determine whether unknown code names are sufficiently resolved. Some complete-code tools (*e.g.*, [43]) do not implement complicated code analyses. As they never touch some code names, boosting such tools is insufficient to determine whether unknown code names are fully resolved. In contrast, although we can feed resolved partial programs to a compiler, it is a too strict criterion, since it is feasible to conduct many complicated analyses even without producing bytecode from partial programs.

Our Contributions. In this paper, we propose a novel approach, called Graphs for Partial programs (GRAPA), that boosts complete-code tools to analyze partial programs. As it can take huge effort to analyse whole projects, researchers [7], [22], [10], [21] have explored analyzing only a subset of a whole program. Their basic idea is to extract an abstraction of other parts of a program. Even if the abstraction is not fully correct, it speeds up the analysis on the subset of a program significantly. Following their idea, GRAPA enables partial-code analysis by constructing the context code of a partial program. In particular, to handle the first challenge, we identify a compiled release of the complete program whose version is closest to the partial program to be analyzed, and extract information from the compiled release to fully resolve unknown code names. To handle the second challenge, we build System Dependency Graphs [12] (SDGs) for partial programs, which is a general abstraction of all code elements and relations, compared with lighter weight analyses in task-specific tools. It should be noted that a lot of useful analyses, such as change impact analysis [31], information flow analysis [37], and static slicing [15], can be simply performed on the SDG of the program to be analyzed.

This paper makes the following major contributions:

- A novel research idea for generally improving partial-code tools. Instead of another approach for specific partial-code analysis, our research idea has the potential to boost many existing complete-code tools for partial-code analysis, if such tools are built on compilers.
- A novel approach, called GRAPA, that boosts complete-code tools for partial-code analysis. It (1) includes a technique to locate context versions for a partial program; and (2) extends PPA with additional inference strategies.
- A tool that boosts WALA for partial-code analyses. The tool has enabled more in-depth empirical studies, and more advanced bug detection approaches.
- An evaluation of our tool on 8,198 partial-code bug fixes that are collected from four popular open source projects. Our results of the first evaluation show that GRAPA fully resolved unknown code names and thus built SDGs for 98.5% of the total bug fixes. Our results of the second evaluation show that in 96.1% of bug fixes, its code-name resolution results are identical with those generated by a Java compiler on their corresponding manually built complete programs. In summary, different from existing imprecise partial-code tools, our tool preserves the preciseness of WALA.

II. MOTIVATIONS

In this section, we use an example to illustrate the application scenarios of partial-code analysis and the limitation of existing tools. The example is from the committed files of ARIES-241[2]. Specifically, programmers modified the `TradeJdbc.java`, and added Line 4 as follow:

```
   public TradeJdbc{  ...
1.   private ... String getTSIAQuotesOrderByChangeSQL =...
2.   public MarketSummaryDataBean getMarketSummary() ... {
3.     Connection conn = null;
4.+    PreparedStatement stmt = getStatement(conn,
        getTSIAQuotesOrderByChangeSQL, ...);}
```

This bug fix may need to be analyzed for various reasons such as bug prediction and mining bug/repair patterns, and typically both the old and new version `TradeJdbc.java` needs to be fed into a static analysis tool. According to our definition in Section I, these two versions of `TradeJdbc.java` are two partial programs. PPA is the state-of-the-art tool to analyze

[2]https://issues.apache.org/jira/browse/ARIES-241

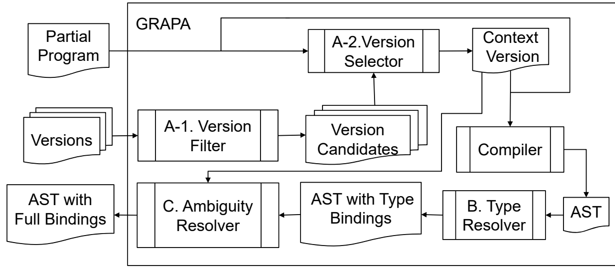


Fig. 1: The overview of GRAPA.

partial programs. It is built on existing compilers such as the Eclipse JDT and Polyglot, and uses its underlying compiler to resolve known code names, and iteratively infers unknown types based on known types. Table I shows the eight inference strategies of PPA. In this table, `t1` $\lessdot$ `t2` denotes that `t1` is a subtype of `t2`, and `t1` $\gtrdot$ `t2` denotes that `t1` is a super type of `t2`. The method strategy infers variable types based on known method signatures, and can reversely infer method signatures based known types, although it is not explicitly defined

As the types of the `getTSIAQuotesOrderByChangeSQL` and `conn` variables are known in Lines 1 and 3, it shall be feasible for PPA to infer the parameter types of the `getStatement` method in Line 4. However, PPA fails to resolve the parameter types as it depends on the underlying compiler (Eclipse JDT) to acquire the initial resolved code names, and Eclipse JDT (and most other compilers) stops compilation encountering the unknown type `MarketSummaryDataBean`. Although possible, revising existing compilers to bypass compilation stops raises extra burden and limitation. Furthermore, as `getStatement` is inherited, it is impossible for PPA to fully resolve the method signature as it does not know which super type of `TradeJdbc` defines the method.

As the above scenario is frequent, in Section IV-C, we find that PPA fully resolves only 28.7% bug fixes. In GRAPA, we acquire initial resolved code names from compiled releases and propose additional variable and field inference strategies (Section III-C). In this example, Aries release several versions[3]. GRAPA refers to the closest version for resolving more code names. As the closest version is introduced, it also reduces the crashing probability of the underlying compiler for other reasons (*e.g.*, incompatibility). As a result, in total, GRAPA fully resolves 96.1% bug fixes.

III. APPROACH

A. Approach Overview

Figure 1 shows the overview of GRAPA. The basic idea behind GRAPA is to extract context code of a given partial program, and to enable complete-code analysis on the partial programs with the information from the context code.

Definition 3 (Context Code): Given a partial program (*i.e.*, a set of source files) p which uses a set of code names $Names(p)$ as defined in Definition 2, the code context of p with depth 1 is defined as $Context^1(p) = p \cup \{v|\exists n \in Name(p), v\ declares\ n\}$, where v is a binary or source file. The full code context of p is then defined as $Context^*(p)$, indicating applying $Context^1$ function recursively on p until the result set no longer change (*i.e.*, the fixed point is reached).

With the definition of partial programs, and code context, we can see that for a partial program p, our goal is to find $Context^*(p)$ and feed it to a complete-code analysis tool (*e.g.*, WALA). The ideal source of context code would be the whole source code set, and all dependency jar files at the version the partial program is from. However, a recent study [36] shows that even if checking out whole set of source files, most commits are not compilable.

In GRAPA, for a partial program, we use the released versions of the software project it belongs to as its context code. As a released version is already successfully compiled, we do not need to fix compilation errors. A partial can refer to code names that do not appear in any released versions. For example, a code name may be added after a version is released, and deleted before the next version is released. In addition, a partial program can be from a code base after its latest release, so all the released versions are outdated. As a result, locating context code alone is insufficient, so we further propose inference strategies to resolve unknown code names.

As shown in Figure 1, GRAPA has the following major steps. For a partial program, GRAPA first searches for a compiled software versions that encloses the given code piece or is compatible with it (detailed in Section III-B). We refer to this version as the *context version* of the given partial program. After that, GRAPA extracts ASTs from the partial program, and resolves unknown type bindings of ASTs (detailed in Section III-C). To integrate with complete-code tools, GRAPA further resolves ambiguous types between the partial programs to be analyzed and context code (detailed in Section III-D).

B. Context Version Extraction

To efficiently find the context version of a partial program, GRAPA uses a two-stage strategy: (1) according to the code names in the partial program, GRAPA identifies a small set of context version candidates (detailed in Section III-B1), and (2) GRAPA tries the context version candidates one by one to compile the partial program (detailed in Section III-B2).

1) Version Filter: GRAPA first extracts the declared types, methods, and fields from the binary code (*i.e.*, all `.class` files) of all available compiled versions, and constructs three lists of declared code names: L_t denotes the list of declared types; L_m denotes the list of declared methods; and L_f denotes the list of declared fields. In a list $L = \{l_1, \ldots, l_n\}$, each item, l, is in the format of $\langle c, V\rangle$, where c denotes a code name, and $V = \{v_1, \ldots, v_n\}$ denotes the set of compiled versions that declare c. For example, at the granularity of types, it extracts the following list for the released versions of Apache Derby[4]:

```
ClientSavepoint, {10.11.1.1}
QualifierUtil,   {10.1.2.1, 10.1.3.1}
...
```

For a partial program, we determine its context-version candidates based on the code names it uses. A context-version

[3]http://aries.apache.org/downloads/archived-releases.html

[4]https://db.apache.org/derby/

candidate should declare most (if not all) code names used by the partial program, if the code name is not declared in the partial program itself. For example, if a partial program declares a local variable whose type is , it uses the code name `ClientSavepoint`, and `10.11.1.1` will be considered a context-version candidate. Given partial program p, to extract the set of code names p uses (denoted as $Names(p)$), GRAPA applies the following rules.

1) For each `a.f` expression where `a` is a variable and `f` is a field, it adds both `a` and `f` to $Names(p)$.
2) For each `T.f` expression where `T` is a type and `f` is a field, it adds `T` and `f` to $Names(p)$.
3) For each `a.m(p, ...)` expression where `a` is a variable, `m` is a method, and `p` is a parameter, it adds `a`, `m`, and {`p, ...`} to $Names(p)$.
4) For each `T.m(p, ...)` expression where `T` is a type, `m` is a method, and `p` is a parameter, it adds `m` and {`p, ...`} to $Names(p)$.
5) For each `m(p, ...)` expression where `m` is a method and `p` is a parameter, it adds `m` and {`p, ...`} to $Names(p)$.
6) For each `(T)a` cast expression where `T` is a type and `a` is a variable, it adds both `T` and `a` to $Names(p)$.
7) For each `T a` declaration expression where `T` is a type and `a` is a variable, it adds `T` to $Names(p)$.
8) For each `T m(T1 p1,...) throws E1,...` expression where `T` is a return type, `m` is a method, `p1` is a parameter, `T1` is the type of `p1`, and `E1` is a thrown exception, it adds `T`, {`T1, ...`} and {`E1, ...`} to $Names(p)$.
9) For each `class|interface|enum T extends T1,... implements I1,...` expression where `T` is a declared class, interface, or enum, `T1` is a type, and `I1` is an interface, it adds {`T1, ...`} and {`I1, ...`} to $Names(p)$.

To determine the set of context-version candidates based on $Names(p)$ and a list of declared names L (L can be one of L_t, L_m, and L_f), we developed Algorithm 1. In the algorithm, Line 1 initializes V with the versions of l_i, where the declared code name of l_i appears in the list of called code names ($Names(p)$). Line 2 iterates used code names, if they appear at least in one version. For each iterated code name `c`, Line 3 compares whether $V \cap l_c.V$ is an empty set. If it is not an empty set, Line 4 updates V with $V \cap l_c.V$. If it is, Line 6 leaves the current V as the output. This line guarantees that V has at least one version.

Algorithm 1 Context Version Extraction Algorithm

Input:
  L is a list of declared code names
  $Names(p)$ is a set of called code names
Output:
  V is a set of context versions
1: $V \leftarrow l_i.V$ where $l_i.V \neq \emptyset \wedge l_i.c \in Names(p)$
2: **for all** $c \in Names(p) \wedge l_c.V \neq \emptyset$ **do**
3:   **if** $V \cap l_c.V \neq \emptyset$ **then**
4:     $V \leftarrow V \cap l_c.V$
5:   **else**
6:     break
7:   **end if**
8: **end for**

Given the set of used code names ($Names(p)$), GRAPA runs Algorithm 1 on each of the three lists of declared code names (L_t, L_m, and L_f) and produces three sets of context-version candidates as V_t, V_m, and V_f. As Line 6 of the algorithm guarantees that V has at least one version, V_t, V_m, and V_f are all nonempty sets. Shi *et al.* [32] show that types are more unlikely to change than methods and fields. For a partial program, GRAPA initiates its context versions V as V_t, and refines V as $V_t \cap V_m$ and $V_t \cap V_m \cap V_f$, if such intersection does not generate an empty set.

2) Version Selector: Our version filter uses an efficient algorithm to quickly find context version candidates from potentially many released versions. However, although the algorithm guarantees that all code names referred in a partial program are declared in a context version candidate, it cannot rule out incompatibilities such as type conflicts in transitive type inference. With fewer versions as context version candidates, we apply a compiler, which is a more heavy weight but more precise code name resolver, to the partial program and each of its context-version candidate to select the final context version. Specifically, as partial programs are syntactically correct, a compiler's syntax analysis can build the AST from a partial program, but as partial programs has unknown code names, the semantic analysis often fails to add name and type bindings to ASTs. As a result, a compiler will throw exceptions, when it encounters unknown or incompatible code names.

GRAPA uses the Eclipse JDT compiler as its underlying compiler, since many complete-code tools are built on the compiler. During parsing, a compiler can search its class path for code names. GRAPA adds context versions to the class path of a compiler, since it allows a compiler to search code names, and thus reduces the possibility of throwing exceptions. As introduced in Section III-B1, GRAPA can locate multiple context versions for a partial program. In this step, GRAPA tries context-version candidates one by one, until no exceptions are thrown. Although this strategy is simple, our evaluation results show that it is already sufficient to analyze real-world partial programs, such as bug fixes.

Algorithm 2 Inference Algorithm

Input:
  F is a list of facts
  T is a set of known types
Output:
  T is a set of inferred types
1: $T \leftarrow ppa'(F, T)$
2: **while** $\Delta T \neq \emptyset$ **do**
3:   $T \leftarrow variable(F, T)$
4:   $T \leftarrow field(F, T)$
5:   $T \leftarrow ppa'(F, T)$
6: **end while**

C. Context-aware Type Resolver

After the context version is determined, GRAPA further enhances the inference rules of PPA to take full advantage of the information from the context code. As shown in Table I, PPA follows eight strategies to resolve unknown type bindings. For example, its assignment strategy is as follow:

```
1 B.field=``Hello World'';
2 → field,unknown,⋗ java.lang.String
```

`t1` ⋖ `t2` denotes that `t1` is a subtype of `t2`, and `t1` ⋗ `t2` denotes that `t1` is an ancestor of `t2`. As shown in the above example, PPA considers only the partial program itself, its inference strategies are localized to statements, and do not fully reuse inferred results. In contrast, with context code available, GRAPA can link the definition and all usage locations of a variable in the code piece, and infer the variable type based on the global information from all usage locations. In particular, GRAPA has the following three additional inference strategies:

1. Variable inference strategy. For each method, the strategy updates bindings of variables until it is safe. Suppose that a method declares a variable v and uses the variable in locations $v_1, \ldots, v_n$. We use $dt(v)$ to denote the type of v, and the safe order of inferred types follows the definition of PPA, *i.e.*, `unknown` < `missing` < `super missing` < `full`. We consider the v variable to be safe, if $dt(v) = dt(v_1) = \ldots = dt(v_n)$. For partial programs, a variable can be unsafe, since a compile can fail to resolve its type at specific locations. For example, if v_i appears in a code line with a serious compilation error, a compiler can fail in resolving the bindings for v_i. As a result, although the bindings of other locations are resolved, the binding of v_i is still unknown. In addition, a variable can become unsafe during inference. For example, if a known variable is assigned to v_i, the assignment strategy of PPA can infer the binding of v_i. After the inference, v_i can be safer than its other locations. When this happens, our variable inference strategy locates the safest type $dt(v_i)$, and propagates $dt(v_i)$ to all the locations of the v variable to make it safe.

2. Field inference strategy. The field inference strategy is similar to the variable inference strategy, but with a different variable scope. The strategy updates bindings of fields until they coincide. For each field, the strategy combines solved types in all its usage locations.

3. Switch inference strategy. For a switch statement `switch(v)`, PPA defines a switch inference strategy that considers $dt(v)$ as a subtype of the `int` primitive. The strategy is not fully correct, since $dt(v)$ can be `byte`, `short`, `char`, `int`, `String` and `enum` values. GRAPA revises the strategy, and considers that a switch statement `switch(v){case c1:... case cn:...}` is safe, if $dt(v) = dt(c_1) \ldots = dt(c_n)$. If a switch statement is unsafe, GRAPA locates the safest type $dt(c_i)$ from $dt(v)$ and $dt(c_1) \ldots dt(c_n)$, and propagates $dt(c_i)$ to all the locations to make it safe.

PPA can infer the type for a variable at a location, but leaves its type unresolved at other locations. As a result, although the type of the variable can be useful to infer more unknown bindings, PPA fails to leverage such benefits. Algorithm 2 shows the process to solve the problem. Line 1 infers types with our modified PPA (ppa'). Here, GRAPA modifies the original switch inference strategy of PPA. For a variable or a field, Lines 3 and 4 search for its safest type, and updates all its locations with the safest type. After that, Line 5 further infers types with our modified PPA. If more unknown bindings are resolved, Line 2 repeats the follow-up lines until no new unknown bindings are thus resolved. Here, ΔT denotes newly resolved bindings in each iteration. In Algorithm 2, our variable and field inference strategies unveil the full potential of PPA, since they combine facts from different locations to infer the safest types for unknown code names.

We use the following partial program to illustrate the difference between GRAPA and PPA on type binding resolution.

```
1 b = A.f(1);
...
2 c = b + "xyz";
3 Object obj = A.f(2);
```

PPA resolves type facts for each variable location in the example, so for Line 2, it infers variables `b` and `c` as of type `java.lang.String`, and for Line 3, it infers `obj` and `A.f` as of type `java.lang.Object`. Finally, based on the type of `A.f`, PPA further resolves `b` at Line 1 as of type `java.lang.Object`. Here, PPA leaves the variable `b` at Line 1 and Line 2 with different types, because it does not know whether the two appearances refer to the same variable in partial-code analysis.

In contrast, GRAPA extracts information from the context version to confirm that the whole code piece comes from the same method and the two appearances of `b` refer to the same local variable. GRAPA further uses Algorithm 2 to infer the type of `b` across all appearances, and finally, the type of `b`, and `A.f` are both updated to `java.lang.String`.

D. Ambiguity Resolver

If a complete-code tool does not require inter-procedure analyses, the integration is straightforward. For example, as MAPO implements only intra-procedure analyses, it is feasible to boost MAPO for partial programs, after unknown bindings are resolved. However, if a complete-code tool requires inter-procedure analyses, involving the context version into partial-code analysis also results in a new challenge. As the context version can contain a copy of the partial programs piece, there may be two duplicate declarations of the same code name (*e.g.*, types, variables, and methods). In the partial-code analysis, we need to make sure that GRAPA analyzes the partial programs piece, instead of its copy in the context version.

For example, suppose that a partial program implements the $t_1, \ldots, t_n$ types and t_1 uses t_2. When a inter-procedure tool analyzes t_1, it needs to locate the declaration of t_2. When locating a type, a compiler typically searches the build path of a project before they search the source code of the project. GRAPA builds a temporary project for the partial program, and adds the context version to the build path of the project. The context version can include t'_2 whose name is identical with t_2. As a result, the t'_2 type in the context version is analyzed, instead of the t_2 type in the partial program. As the context

version is only an approximate of the partial program, the analysis results can be inaccurate.

We envisage that there are at least three techniques to handle the problem: (1) removing $t_1, \ldots, t_n$ from the build path of the project, (2) pre-analyzing $t_1, \ldots, t_n$, and merging their analysis results, and (3) changing the search sequence. The first technique is simple, but may not work on some complete-code tools. For example, if a compiler cannot find a type in the build path of a project, it may either search the source files of the project or throw exceptions, depending on its implementation details. The second technique can introduce extra analysis effort. The third technique requires modifying source files of complete-code tools, but it is flexible and does not introduce extra analysis effort. Therefore, in the implementation of GRAPA, we follow the third technique and we introduce how we modify the subject complete-code analysis tool in Section III-E.

E. Implementation

In the current implementation of GRAPA, we select WALA as the subject tool to illustrate the potential of our approach. WALA implements various analyses (*e.g.*, type hierarchy analysis, data flow analysis, slicing, and dependency analysis). WALA implements its advanced analyses on its unified intermediate representation called IR, whose format is like Java bytecode. When WALA analyzes Java source files, it uses the Eclipse JDT compiler to build ASTs from source code, and then translates ASTs into IRs. When the underlying compiler parses a partial program p, it is able to build an AST for p, since the syntax of p is correct. However, the unknown code names in p cause exceptions when WALA translates p to its IR. As a result, WALA cannot analyze partial programs.

Like most compilers, WALA searches the build path first when resolving types. As mentioned in Section III-D, we modify its code to reverse the search strategy. Here, we modify the classloader of WALA, instead of the classloader of JVM. As a result, our tool is not coupled to JVM. It should be noted that our modifications to WALA is minimal and such modifications are similar for all analysis tools with a specific compiler component. For analysis tools that depend on a compiler already supported by GRAPA (e.g., Eclipse JDT), GRAPA is able to work with the tools directly.

IV. EVALUATIONS

With the implementation of GRAPA, we conducted evaluations to explore the following research questions:

(RQ1) How effectively does GRAPA resolve unknown code names of partial programs (Section IV-A)?

(RQ2) What is the accuracy of GRAPA to resolve unknown code names (Section IV-B)?

(RQ3) What is the effectiveness of GRAPA's internal techniques (Section IV-C)?

RQ1 concerns the effectiveness of GRAPA. As our primary research goal is to boost existing complete-code tools, in Section IV-A, we select the state-of-the-art tool WALA as the subject tool for the first research question, and use the SDG generation for bug fixes as our application scenario. From four widely used open source projects, we collected 8,198 bug fixes. Our results show that WALA built SDGs successfully for 98.5% of the bug fixes after GRAPA resolved their unknown code names. This is a significantly improvement, since as a complete-code tool, WALA cannot build any system dependency graphs from such bug fixes. We understand that it is possible that some code names are wrongly fixed. To explore this issue, we introduce more evaluations in RQ2.

TABLE II: Overall Result.

Name	V	Fix	Success	Failure			%
				U	F	D	
Aries	4	547	533	1	10	0	97.4%
Cassandra	116	3,444	3,405	11	16	12	98.9%
Derby	20	2,560	2,538	8	12	0	99.1%
Mahout	13	560	494	3	51	3	88.2%
Total		7,111	6,970	23	89	15	98.0%

U: unsupported fragment; F: not fully resolved; D: defect.

RQ2 concerns the accuracy of GRAPA. In Section IV-B, we select 91 bug fixes, and manually build the corresponding buggy and fixed versions, and fed the complete built programs to WALA, so that WALA can correctly resolve all the code names and build SDGs correctly. Collecting these SDGs as golden standard, we compare with them SDGs generated by GRAPA for the same buggy and fixed programs (partial programs). Our results show in 96.1% of bug fixes, GRAPA-based SDGs are identical to the SDGs in the golden standard.

RQ3 concerns the internal techniques of GRAPA. In Section IV-C, we turn off different internal techniques of GRAPA, and our results show the significance of individual techniques. The results also reflect how much code-name resolution is needed to leverage a complete-code tool such as WALA.

A. RQ1: The Effectiveness

1) Subject: As we introduced in Section II, bug fixes can be viewed as partial programs, and we use bug fixes as the subjects of our evaluation. We use the following two criteria to determine a bug fix:

1. Issue number. Open source projects typically have issue trackers to track various issues (*e.g.*, bugs, improvements, new features, tasks, and sub-tasks). When programmers commit their modified files, they often write the corresponding issue number in the message of the commit. For example, in Cassandra, a commit's message says "implement multiple index expressions. patch by jbellis; reviewed by Nate McCall for CASSANDRA-1157". In the issue tracker, the page of the issue says that CASSANDRA-1157 is a bug. We thus determine that the corresponding commit is a bug fix.

2. Keyword. Programmers may detour issue trackers, especially when they believe that a change is trivial. When they commit such a change, programmers may write a message to describe the fix. For example, in Aries, the message of a commit says "Fix broken service registration listener". We thus determine the commit as a bug fix, since its message contains words such as "bug" or "fix". The heuristic is simple, and a

number of previous studies (*e.g.*, [19]) used the same technique to extract bug fixes.

In Table II, Column "Name" shows project names. All the projects are from the Apache software foundation. Column "Version" lists released versions of the projects. Column "Bug Fix" lists number of analyzed bug fixes. We select these projects, since they are under active maintenance and their bug fixes are already collected and confirmed by previous studies [41]. Here, we filter bug fixes that do not modify source files or modify only test code. Typically, a released version does not include test code, so it becomes infeasible to locate context versions for test code. As the input of this evaluation, a bug fix is defined as follow:

Definition 4 (Bug Fix): A bug fix between a buggy version bV and a fixed version fV is a pair of two partial programs $< bp, fp >$, where bp belonging to bV and includes all file removed or modified, fp belonging to fV and includes all source files added or modified during the bug fix.

Tufano *et al.* [36] analyzed 219,395 snapshots from 100 Apache projects. They found that only 38% code commits are automatically compilable, and 58% are caused by unresolvable references. Mockus *et al.* [26] introduce that the Apache foundation has a strict management over its hosted projects. Without such a management system, the problem may be worse in other open source communities (*e.g.*, Github). As a result, both bV and fV are typically partial programs. Zhong and Su [41] show that such partial programs typically contain fewer than three source files.

2) Evaluation Scenario: We use the SDG generation for bug fixes as our evaluation scenario, due to two considerations. First, analysis of bug fixes is the basis of many downstream research topics such as bug prediction and mining of bug/repair patterns. Automating in-depth analysis of bug fixes will allow all above research to be applied on a much larger data set. Second, SDGs are general abstractions of all code elements and relations in the program, and many general code analysis tools such as WALA and CodeSurfer [1] use SDGs as their default output. A lot of useful analysis results, such as change impact analysis [31], information flow analysis [37], static slicing [15], can be direct extracted from the SDGs of the program to be analyzed. As the output of GRAPA, an SDG is defined as follow (the same as its definition in WALA):

Definition 5: A system dependency graph is defined as $g = \langle V, E_1, E_2 \rangle$, where V is a set of nodes corresponding to variables/expressions, and $E_1, E_2 \subseteq V \times V$ are two sets of edges. A $\langle s_1, s_2 \rangle \in E_1$ edge denotes a data dependency from s_1 to s_2, and a $\langle s_1, s_2 \rangle \in E_2$ edge denotes a control dependency from s_1 to s_2.

As an example, Figure 2 shows the built system dependency graph for the buggy code of DERBY-5396[5], which swallows an exception.

```
1: private static void closeStream(...){
2:   try{
3:     if(stream!=null)
4:        stream.close();
5:   }catch(IOException e){
6:     Util.javaException(e); } }
```

[5]https://issues.apache.org/jira/browse/DERBY-5396.

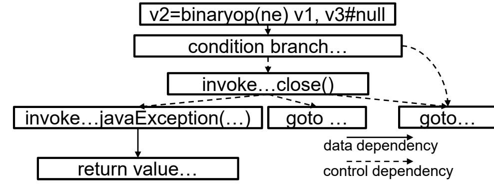


Fig. 2: Built graphs for partial programs

Under this application scenario, with GRAPA, bug prediction and pattern mining tools can easily acquire SDGs and other down-stream analysis results (*e.g.*, slicing) from a large number of bug fixes. While without GRAPA, there must be a lot of people manually build all the relevant code versions, and analyzing the whole code base is also much slower than analyzing only the partial programs involved in the bug fixes.

3) Criterion: As we collected thousands of bug fixes and each fix can contain thousands of unknown code names, it is infeasible to manually examine whether GRAPA fully resolved unknown bindings of a bug fix or not. Instead of a manual examination, we use the underlying complete-code tool, WALA, as an automatic measure. A simple tool is insufficient for the automated measure, since it may not touch many resolved code bindings. In contrast, WALA is the state-of-the-art tool for Java analysis, and it implements many advanced analysis techniques that require various code bindings. Our criterion illustrates our contributions to the state of the art, since it complements existing tools.

For each bug fix, we extract its buggy files (revised or deleted files of the bug-fixing commit in the pre-fix version) and modified files (revised or added files of the bug-fixing commit in the post-fix version). Both the set of the buggy files and modified files are partial program, since we did not check out their corresponding whole projects. We consider that bindings in the buggy or modified files are *fully resolved*, if WALA is able to build system dependency graphs from the code. For a bug fix, we consider that it is successfully resolved, if GRAPA fully resolved all its source files. Here, we concede that WALA can build a system dependency graph for a partial program, even if it is incorrectly resolved. We further investigate this issue in Section IV-B.

4) Result: GRAPA uses only tens of released versions to approximate contexts of thousands of commits, and thus avoid effort of (1) analyzing the whole project for each commit, and (2) building each commit, which often needs nontrivial manual effort. Our results lead to the following findings:

1. GRAPA fully resolved unknown code names for most bug fixes. In Table II, Column "Success" lists number of successes, and Column "%" lists success rates. In total, we find that GRAPA fully resolves 98.5% bug fixes, since their dependency graphs can be built.

2. WALA failed to analyze 0.3% fixes, even after GRAPA fully resolved their bindings. Column "Failure" lists number of failures. Based on our inspection, we further put them into three categories. In particular, subcolumn "U" lists fixes that WALA failed to analyze. For example, the "f0be890" commit

TABLE III: The results of comparing with the golden standard.

Name	Version	Fix	File	Method	Same	%
Aries	1.0	24	38	456	452	99.1%
Cassandra	3.0.0	14	22	585	558	95.4%
Derby	10.11.1.1	37	88	2,088	1,995	95.5%
Mahout	0.10.0	16	25	661	636	96.2%
Total		91	173	3,790	3,641	96.1%

of Cassandra modifies `SchemaKeyspace.java`, a file with lambda expressions:

```
ALL.forEach(table->getSchemaCFS(table).truncateBlocking());
```

The underlying tool, WALA, cannot analyze lambda expressions. As another example, we notice that GRAPA fails on the following code:

```
final StandardMBean standardMBean =
   new StandardMBean(bean, beanInterface){...};
```

We inspect the relevant code of WALA, and find an assertion:

```
assert superCtor!=null:"couldn't find constructor for
                          anonymous class";
```

Based on the error message, WALA seems to have limitations on analyzing anonymous classes.

3. GRAPA failed to fully resolve bindings in 1.1% bug fixes. Subcolumn "F" lists cases where bindings are not fully resolved. We find that the failed cases fall into three major categories:

1. Insufficient facts to infer bindings. For example, we find a code snippet as follows:

```
ReplicationMessage message = ...;
switch (message.getType()){
   case ReplicationMessage.TYPE_LOG:...
   case ReplicationMessage.TYPE_FAILOVER:...
   case ReplicationMessage.TYPE_STOP:...
   default:...
}
```

In this code snippet, `ReplicationMessage` is an unknown code name. Based on the `switch` statement, GRAPA infers that `TYPE_LOG`, `TYPE_FAILOVER`, `TYPE_STOP`, and the return value of the `message. getType()` method shall be the same. The type does not appear in any released versions, but between two versions. As the code snippet does not provide adequate information, GRAPA fails to determine their types.

2. Insufficient type inference strategies. For example, we find a code snippet as follows:

```
public class EditAuthorForm extends FormServlet{
  private static final long serialVersionUID = ...;
```

In the above code, `FormServlet` is an unknown code name, but the inference strategies of PPA and our extension cannot infer the binding for the ancestor type.

3. Conflict facts. For example, we find the following snippet:

```
public void setup(String aggregateName){
  super.setup(aggregateName); ...;
```

The above code calls the `super.setup(...)` method, but in the located version, the signature of the method is as follow:

```
public void setup(ClassFactory classfactory, String s,
    DataTypeDescriptor datatypedescriptor){...;}
```

The above code and release have different definition on the signature of the `super.setup(...)` method. We find that when this type of conflicts happens, PPA tends to reject the information from source code. However, as our located releases are approximate, the information from the build path shall be rejected, so the strategy of PPA leads to incorrect binding resolution in this example.

4. The Eclipse JDT compiler failed to parse 0.2% fixes. Subcolumn "D" lists cases where the JVM machine crashes:

```
# A fatal error has been detected by the Java Runtime
Environment:...
```

We notice that the following method calls appear in crash logs:

```
ciMethod org/eclipse/jdt/core/compiler/CharOperation...
.../jdt/internal/compiler/ast/TypeReference<init>...
.../jdt/internal/compiler/ast/Expression<init>...
```

We suspect that the underlying compiler, Eclipse JDT, is not designed for parsing partial program, so it reports the above problem when parsing partial program.

In summary, our results show that GRAPA effectively resolved bindings for more than 90% of bug fixes. Only 1.1% of bug fixes are not fully resolved, which may be resolved in our future work.

B. RQ2: The Accuracy

1) Subject: Table III shows our subjects. We choose the latest version, since it is easier to fix possible compilation errors. For example, an old version may require a third-party library that is no longer available. Column "Bug fix" lists selected bug fixes. Tufano *et al.* [36] show that recent snapshots are almost 3.76 times more likely to compile than early snapshots. To reduce the manual effort to prepare the golden standard, we select bug fixes whose detected version is exactly the latest version. We filter out bug fixes whose graphs are not built, and bug fixes that do not have compilation problems. Column "File" and Column "Method" list total source files and total methods of selected bug fixes, respectively.

2) Criterion: For each bug fix, we checked out its buggy and fixed versions, and repaired all compilation errors. After that, we used WALA to build SDGs (G_{WALA}) for the buggy files of the bug fix as our golden standard. Meanwhile, we use GRAPA to build SDGs (G_{GRAPA}) from the buggy files as a partial program. We implemented a tool to compare G_{WALA} with G_{GRAPA} for their differences. The tool uses the Hungarian algorithm [20] to match nodes, and we define the distance between two nodes (m and n) as follows:

$$dis(m,n) = \frac{|i(m)-i(n)|}{i(m)+i(n)} + \frac{|o(m)-o(n)|}{o(m)+o(n)} + d(l(m),l(n)) \quad (1)$$

$i(m)$ denotes the indegree of m; $o(m)$ denotes the outdegree of m; $l(m)$ returns the label of m; and $d(l_1,l_2)$ returns the Levenshtein edit distance between the l_1 label and the l_2 label.

3) Result: **1. GRAPA correctly resolved bindings in most fixes.** Table III shows the overall results. Column "Same" lists methods whose built graphs are identical. Column "%" lists corresponding percents. In total, 96.1% bindings resolved by GRAPA are identical with the golden standard.

TABLE IV: The significance of internal techniques.

Name	No Version		No PPA		No Extension	
	Suc.	%	Suc.	%	Suc.	%
Aries	339	61.9%	353	64.6%	435	79.6%
Cassandra	899	26.1%	1,546	44.9%	2,332	67.7%
Derby	637	24.9%	1,088	42.5%	1,882	73.5%
Mahout	189	33.8%	351	62.6%	442	79.0%
Total	2,041	28.7%	3,307	46.5%	5,113	71.9%

2. GRAPA failed to correctly resolve bindings in only 3.9% fixes. After manual inspection, we identified two types of wrong resolutions:

1. Not fully resolved bindings. For example, CASSANDRA-6181 includes the following code:

```
public void serialize(DataOutput out...)...{
  ByteBufferUtil.
    writeWithShortLength(tombstones.starts[i], out);...}
```

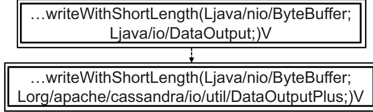
In this code, the `writeWithShortLength` method is unknown. PPA infers its signature based on its parameters:

```
writeWithShortLength(Ljava/nio/ByteBuffer;
                       Ljava/io/DataOutput;)V
```

When we manually recover the code, we find that the declaration of the method is as follow:

```
public static void writeWithShortLength
    (ByteBuffer buffer, DataOutputPlus out)...{...}
```

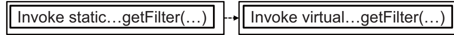
The above code has no compilation errors, since `DataOutputPlus` is a subtype of `DataOutput`. Our tool detects the following difference:



2. Wrong static and instance code names. For example, CASSANDRA-8413 modifies the following code:

```
bf = FilterFactory.getFilter(...);
```

Here, the `FilterFactory.getFilter(...)` method is unknown. PPA wrongly resolves the static method to an instance method, although it correctly resolves all the parameters and the return value. We identify the wrong resolution, since our tool detects the following difference:



In the above two situations, WALA can still build graphs based on incorrect binding resolutions. In Table II, the results on these methods are counted as successes according to the criterion in Section IV-A. Our tool detected the incorrect resolutions, but as shown in Table III, 96.1% of the total bindings are correctly resolved.

C. RQ3: Internal Techniques

1) Setup: Our approach has two major techniques to resolve unknown bindings: (1) it detects the context versions for a partial program (Section III-B1), and (2) it extends PPA with additional inference strategies (Section III-C). In this section, we use GRAPA to analyze the bug fixes in Table II, without the above techniques to show their significance.

2) Result: In Table IV, Column "No Version" shows the results when GRAPA does not detect context versions. Column "No PPA" shows the results when GRAPA does not resolve unknown bindings with PPA. Column "No Extension" shows the results when GRAPA uses the standard PPA, instead of our extended PPA as described in Section III-C. Subcolumn "Suc." lists number of successes. Subcolumn "%" lists success rate. The results lead to our findings:

1. The technique of locating context versions is the most effective internal technique. Without locating context versions, only 28.7% of the total fixes are fully resolved. Without the standard PPA and our extended PPA, 46.5% and 71.9% of the total fixes are fully resolved, respectively. The results highlight the importance of locating context versions, and the improvement over PPA.

2. Our extended PPA shows its significance when partial programs have critical compilation errors. Comparing Column "No PPA" with Column "No Extension", we find that our extension of PPA improves the success rates. We notice that like other compilers, the Eclipse JDT compiler can stop resolving bindings after it encounters a critical compilation error. For example, in ARIES-241, a method is as follows:

```
private ... String getTSIAQuotesOrderByChangeSQL =...
public MarketSummaryDataBean getMarketSummary() ... {
 PreparedStatement stmt = getStatement(
   getTSIAQuotesOrderByChangeSQL, ...);...}
```

In the method, the code name `MarketSummaryDataBean` is unknown, and causes a critical compilation error. The underlying compiler, Eclipse JDT, does not resolve any bindings inside the method, after encountering the error. For example, although `getTSIAQuotesOrderByChangeSQL` is declared in the same type, its type is resolved as `null`. Consequently, PPA fails to resolve the signature of the `getStatement` method. Our variable and field strategies links resolved bindings before the error line and those unresolved bindings after the error line. In this example, our field strategy links the declaration of `getTSIAQuotesOrderByChangeSQL` to the parameter of the `getStatement` method. As a result, it becomes feasible to resolve the signature of the method. In this way, our additional inference strategies make the improvements in Table IV.

D. Threats to Validity

The threats to internal validity include our criterion to determine incorrect resolutions in RQ2. To reduce the bias, we implement a tool to compare dependency graphs, but even a tool can introduce errors. The threat could be reduced by introducing more manual inspections. The threats to external validity include that we focus on only a type of partial programs from only Apache and a single tool, so our effectiveness may vary on other subjects. To reduce the threat, we select thousands of commits and the state-of-the-art tool. The threat could be reduced by analyzing more subject tools.

V. APPLICATIONS OF GRAPA

As a support tool, the benefits of GRAPA are indirect, and our evaluations did not show its applications. However, we

believe that GRAPA can be useful in many real applications, since it enables more accurate analysis on partial programs.

1. More in-depth empirical studies. Researchers have conducted various empirical studies to explore open questions that involve partial programs (*e.g.*, bug fixes [38], [41]). As their underlying partial-code tools are limited and imprecise, their results can be superficial or even biased. GRAPA enables more in-depth analysis on partial programs. Indeed, Zhong and Meng [39] have conducted an empirical study on using past fixes to construct new patches. As they compare SDGs of bug fixes, their study is infeasible without GRAPA,

2. More advanced bug detection/repair approaches. Many researchers believe that it is feasible to propose more advanced bug detection/repair approaches, based on the knowledge that is extracted from past fixes. However, as existing partial-code tools are limited, it is quite challenging to automate the extraction. For example, although Kim *et al.* [16] implemented tools to reduce their inspection effort, their analysis is largely manual, which is tedious and error-prone. GRAPA enables more in-depth empirical studies on bug fixes [39] and more advanced bug detection/repair techniques. Indeed, Zhong and Wang [42] have detected previously unknown bugs for all the four projects, and their tool is built on GRAPA.

3. Analyzing partial programs in more scenarios. The applications of GRAPA are not limited to analyzing bug fixes. Besides this scenario, GRAPA can be applied to many other scenarios where partial programs and some compiled releases are available. Typical scenarios include code samples in API documents [40], development emails [30], and code search engines [24]. It is feasible to extend GRAPA for the above scenarios after minor changes. For example, although many code samples in API documents do not have method/class bodies, Zhong and Su [40] show that it is feasible to construct fake bodies, which enables follow-up analysis. As another example, Terragni *et al.* [34] propose an approach that extracts code samples from forums. From such samples, it is feasible to locate context versions by querying such types on existing code search engines (*e.g.*, GrepCode or Github). Despite of that Algorithm 1 is suboptimal, it achieves more than 90% accuracy, leaving adequate space for further improvements.

VI. RELATED WORK

Partial-code analysis. Mishne *et al.* [25] propose PRIME that mines specifications from partial programs. Zhong and Su [40] detect errors in code samples of API documentations, and such samples are partial programs. Subramanian and Holmes [33] extract API calls from code samples in StackOverflow. Our approach presents a general way to leverage tools for complete code, so they can analyze partial programs, complementing existing approaches. Dagenais and Hendren [8] propose PPA that fixes unknown bindings for partial programs. We are the first to provide the insight that extending PPA can potentially leverage complete-code tools to analyze partial programs. Ossher *et al.* [28] resolve missing dependencies for complete programs, but most partial programs are not compilable, even if their dependencies are resolved. Carvalho Gomes *et al.* [9] analyze incomplete Java bytecode. During compilation, complete code information is encoded into bytecode, so JVM can execute it correctly. When analyzing partial code, we do not have such information, so the problem we target is more challenging.

Code comparison. Fluri *et al.* [13] compare ASTs of source files for their differences. Maletic and Collard [23] translates files into srcML and compare the translated format for differences. Kim and David [18] denote code with a set of predicates and compare these predicates for their differences. Buse and Weimer [5] generate documents to describe code changes. Apiwattanapong *et al.* [2] compare CFGs for their delta. Although graphs are more informative, Kim and Notkin [17] complain that CFG-based approaches cannot analyze partial programs, which is complemented by our approach.

Empirical study on bug fixes. Researchers have conducted various empirical studies to understand the nature of bug fixes. Yin *et al.* [38] show that bug fixes can introduce new bugs. Nguyen *et al.* [27] show that repetitiveness is common in small size bug fixes. Eyolfson *et al.* [11] show that the bugginess of a commit is correlated with the commit time. Thung *et al.* [35] show that faults are not localized. Zhong and Su [41] analyze the nature of fixing bugs. Canfora *et al.* [6] analyze the relations between fixed code elements and the time to fix bugs. With our approach, researchers can integrate more advanced tools for analyzing bug fixes, reducing the analysis effort.

VII. CONCLUSION AND FUTURE WORK

In this paper, we propose an approach that locates approximate Java archive files and resolves remaining unknown bindings. After integration, we can boost existing complete-code tools to analyze partial programs. Based on our approach, we implement GRAPA, and conduct evaluations on thousands of partial-code bug fixes. Our results show that our tool correctly fully resolved unknown bindings for 98.5% of bug fixes, and for 96.1% of bug fixes, our binding resolutions are identical with our golden standard.

In future work, we plan to explore two directions as follows. First, applying GRAPA to different application scenarios may require more advanced technical adaptations. For example, symbolic analysis often needs a driver function to initiate objects of partial programs. Person *et al.* [29] propose differential symbolic execution that reuses analysis results between program versions, and test code can provide hints on initiating objects. Analyses of forum threads calls for more advanced searching techniques for context versions. Second, although our tool works on Java, our approach is general and can be adapted for other programming languages.

ACKNOWLEDGMENTS

We appreciate the reviewers for constructive comments, and Na Meng for repairing bugs in GRAPA. Hao Zhong is sponsored by the 973 Program in China (No. 2015CB352203), the National Nature Science Foundation of China No. 61572313, and the grant of Science and Technology Commission of Shanghai Municipality No. 15DZ1100305. Xiaoyin Wang's work is supported in part by NSF grant CCF-1464425 and DHS grant DHS-14-ST-062-001.

References

[1] P. Anderson and M. Zarins. The CodeSurfer software understanding platform. In *Proc. IWPC*, pages 147–148, 2005.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. ASE*, pages 2–13, 2004.

[3] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. In *Proc. ICSE*, pages 375–385, 2012.

[4] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Companion to OOPSLA*, pages 681–682, 2006.

[5] R. Buse and W. Weimer. Automatically documenting program changes. In *Proc. ASE*, pages 33–42, 2010.

[6] G. Canfora, M. Ceccarelli, L. Cerulo, and M. D. Penta. How long does a bug survive? an empirical study. In *Proc. WCRE*, pages 191–200, 2011.

[7] P. Cousot and R. Cousot. Modular static program analysis. In *Proc. CC*, pages 159–179, 2002.

[8] B. Dagenais and L. J. Hendren. Enabling static analysis for partial Java programs. In *Proc. OOPSLA*, pages 313–328, 2008.

[9] P. de Carvalho Gomes, A. Picoco, and D. Gurov. Sound control flow graph extraction from incomplete Java bytecode programs. In *Proc. FASE*, pages 215–229. Springer, 2014.

[10] T. W. Dillig. *A modular and symbolic approach to static program analysis*. PhD thesis, Stanford University, 2011.

[11] J. Eyolfson, L. Tan, and P. Lam. Correlations between bugginess and time-based commit characteristics. *Empirical Software Engineering*, 19(4):1009–1039, 2014.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[13] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[14] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[15] S. Horwitz, T. Reps, and D. Binkley. *Interprocedural slicing using dependence graphs*, volume 23. ACM, 1988.

[16] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. ICSE*, pages 802–811, 2013.

[17] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proc. MSR*, pages 58–64, 2006.

[18] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proc. ICSE*, pages 309–319, 2009.

[19] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proc. ICSE*, pages 489–498, 2007.

[20] H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[21] S. K. Lahiri, K. Vaswani, and C. A. Hoare. Differential static analysis: opportunities, applications, and challenges. In *Proc. FOSE*, pages 201–204, 2010.

[22] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear. Verification modulo versions: Towards usable verification. In *Proc. PLDI*, pages 294–304, 2014.

[23] J. Maletic and M. Collard. Supporting source code difference analysis. In *Proc. ICSM*, pages 210–219, 2004.

[24] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2012.

[25] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *Proc. OOPSLA*, pages 997–1016, 2012.

[26] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[27] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proc. ASE*, pages 180–190, 2013.

[28] J. Ossher, S. Bajracharya, and C. Lopes. Automated dependency resolution for open source software. In *Proc. MSR*, pages 130–140, 2010.

[29] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Psreanu. Differential symbolic execution. In *Proc. FSE*, pages 226–237, 2008.

[30] G. Petrosyan, M. P. Robillard, and R. D. Mori. Discovering information explaining API types using text classification. In *Proc. ICSE*, pages 869–879, 2015.

[31] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proc. OOPSLA*, pages 432–448, 2004.

[32] L. Shi, H. Zhong, T. Xie, and M. Li. An empirical study on evolution of API documentation. In *Proc. FASE*, pages 416–431, 2011.

[33] S. Subramanian and R. Holmes. Making sense of online code snippets. In *Proc. MSR*, pages 85–88, 2013.

[34] V. Terragni, Y. Liu, and S. Cheung. CSNIPPEX: automated synthesis of compilable code snippets from Q&A sites. In *Proc. ISSTA*, pages 118–129, 2016.

[35] F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *Proc. MSR*, pages 74–77, 2012.

[36] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 2016.

[37] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proc. CCS*, pages 116–127, 2007.

[38] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proc. ESEC/FSE*, pages 26–36, 2011.

[39] H. Zhong and N. Meng. Poster: An empirical study on using hints from past fixes. In *Proc. ICSE*, 2017.

[40] H. Zhong and Z. Su. Detecting API documentation errors. In *Proc. OOPSLA*, pages 803–816, 2013.

[41] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proc. ICSE*, pages 913–923, 2015.

[42] H. Zhong and X. Wang. Detecing bugs with past fixes. In *Proc. OOPSLA*, page submitted, 2017.

[43] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. ECOOP*, pages 318–343, 2009.

[44] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. ASE*, pages 307–318, 2009.