2017

Exploiting implicit belief to resolve sparse usage problem in usage-based specification mining

Samantha Syeda Khairunnesa
Iowa State University

Exploiting implicit belief to resolve sparse usage problem in usage-based specification mining

by

Samantha Syeda Khairunnesa

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Hridesh Rajan, Major Professor
Carl Chang
Wei Le

Iowa State University

Ames, Iowa

2018

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. I would like to thank Dr. Hridesh Rajan for his guidance, patience and support throughout this research and the writing of this thesis. Thanks are due to the US National Science Foundation for financially supporting this project under grants CCF-15-18897, CNS-15-13263, CCF-17-23215, CCF-17-23432, and CNS-17-23198.

I would like to thank my committee members Dr. Carl Chang and Dr. Wei Le for their efforts and contributions to this work. Also, I would like to thank the reviewers of 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2017). I would like to extend my thanks to all the members of Laboratory of Software Design for offering constructive criticism and timely suggestions during research. Majority of this draft is adopted from OOPSLA 2017 paper [18], which is written in collaboration with Dr. Hoan Anh Nguyen, Dr. Tien N. Nguyen and Dr. Hridesh Rajan.

I am very grateful to my parents and my husband for their moral support and encouragement throughout the duration of my studies.

ABSTRACT

Frameworks and libraries provide application programming interfaces (APIs) that serve as building blocks in modern software development. As APIs present the opportunity of increased productivity, it also calls for correct use to avoid buggy code. The usage-based specification mining technique has shown great promise in solving this problem through a data-driven approach. These techniques leverage the use of the API in large corpora to understand the recurring usages of the APIs and infer behavioral specifications (pre- and post-conditions) from such usages. A challenge for such technique is thus inference in the presence of insufficient usages, in terms of both frequency and richness. We refer to this as a “sparse usage problem." This thesis presents the first technique to solve the sparse usage problem in usage-based precondition mining. Our key insight is to leverage implicit beliefs to overcome sparse usage. An implicit belief (IB) is the knowledge implicitly derived from the fact about the code. An IB about a program is known implicitly to a programmer via the language’s constructs and semantics, and thus not explicitly written or specified in the code. The technical underpinnings of our new precondition mining approach include a technique to analyze the data and control flow in the program leading to API calls to infer preconditions that are implicitly present in the code corpus, a catalog of 35 code elements in total that can be used to derive implicit beliefs from a program, and empirical evaluation of all of these ideas. We have analyzed over 350 millions lines of code and 7 libraries that suffer from the sparse usage problem. Our approach realizes 6 implicit beliefs and we have observed that adding single-level context sensitivity can further improve the result of usage-based precondition mining. The result shows that we achieve overall 60% in precision and 69% in recall and the accuracy is relatively improved by 32% in precision and 78% in recall compared to base usage-based mining approach for these libraries.

CHAPTER 1. INTRODUCTION

Behavioral specifications (pre- and post-conditions) of application programming interfaces (APIs) could help developers effectively utilize the APIs [17]. However, currently the efforts needed to write behavioral specifications can be quite high [19, 32]. To reduce the costs of producing behavioral specifications of commonly used APIs, prior work has utilized large scale code corpus to infer preconditions [27]. Nguyen *et al.*'s *usage-based precondition mining* collects usage of the API methods within a large code corpus (call sites), at each such call site analyzes the context to identify explicitly written guard conditions that must be true for the API method to be invoked, and then obtains a consensus among all of these guard conditions to infer preconditions of the API method. The process of consensus building *eliminates project-specific guard conditions.* When applied to a large corpus containing hundreds of thousands of call sites, this technique has been shown to be very effective for widely used APIs such as in the Java Development Kit (JDK).

Not all important APIs are very widely-used, however. The API call sites may also not be rich to mean that few (potential) preconditions are present as explicit guard conditions. This in turn makes true preconditions indistinguishable from project-specific guard conditions. We observed this phenomenon when examining the precondition mining results for infrequently-used APIs in JDK from our prior work [27].

We call this the *sparse usage problem* in precondition mining. In this thesis, we propose a technique to solve the sparse usage problem by leveraging *implicit beliefs* (IBs). First, a fact about a program is what is directly derived from the code, e.g., an object is instantiated via a constructor. Second, an implicit belief is the knowledge about the program that can be implied from a fact(s). For example, the programmer might expect that the first parameter of the API call is non-null. An IB can be implied by the use of certain language constructs or semantics. Our intuition is that since these facts are implied,

```
1  public static double[] getOLSRegression(XYDataset data, int series) throws RegressionException {
2   int n = data.getItemCount(series);
3   if (n < 2) {
4     throw new RegressionException("Not enough data to calculate regression.");
5   }
6   ...                                          // explicit condition: n >= 2
7   for (int i = 0; i < n; i++) {
8     double x = data.getXValue(series, i);      // explicit condition: i < n, implicit belief: i >= 0
9                                                // implicit belief: condition n >= 2 is not required
10     ... }
11   ... }
```

Expected Precondition(s)	= { $i \geq 0, i < data.getItemCount(series)$ }
Explicit Conditions(s)	= { $data.getItemCount(series) \geq 2, i < data.getItemCount(series)$ }
Implicit Belief(s)	= { $i \geq 0, data.getItemCount(series) \geq 2$ not required }
Combined Precondition(s)	= { $i \geq 0, i < data.getItemCount(series)$ }

Figure 1.1: Control-related code elements to derive implicit beliefs, API of interest is highlighted in bold font.

developers don't check it explicitly, but they can still be leveraged for usage based precondition mining. Techniques such as symbolic execution [30, 29] and abstract interpretation [5] could also be used to expand the set of available invariants at program points, and help with the sparse usage problem. However, the cost of running these techniques on hundreds of thousands of call sites could be prohibitive. To solve this problem, we propose several kinds of implicit beliefs that can be identified by lightweight program analysis, realize a subset of these analyses, and propose an integrated usage based precondition miner leveraging implicit beliefs.

Implicit beliefs can help filter project-specific conditions. Figure 1.1 presents an example that uses *xy* package from the library *org.jfree.data* to measure the ordinary least square regression. At line 3, the condition $n < 2$ assures that no regression computation is needed if insufficient data points are provided. The intention of the loop at line 7 is to include data from each observation to compute regression coefficient.

The API method *getXValue()* in line 8 is our API of interest. Existing mining approaches extract the explicit guard conditions that must be satisfied before reaching the API call at line 8. In line 3, if the value of n is less than 2 then the *if* branch throws an exception. For the control flow to reach line 6, the condition $n \geq 2$ needs to be true. In line 7, the guard condition of the loop needs to be satisfied to reach

the statement at line 8 inside the loop. These are two explicit guard conditions that can be extracted from the call site at line 8 shown in Figure 1.1.

Now if we statically analyze the language constructs and semantics in the source code, we see that from lines 3-5 the programmer is throwing a regression exception, which is clearly project specific. It can be inferred automatically as this exception is neither an exception from API signature nor any runtime exception thrown by the language itself. Therefore, a technique based on implicit belief could ignore such false positive explicit conditions. In other words, for such client-specific exception, its guard condition is *less trustworthy* to lead to true preconditions than the conditions for API exceptions.

Implicit beliefs can help fill missing data. As discussed before, in line 7 of Figure 1.1, the loop initializes the counter i with a value of 0 and increases it by 1 for each iteration while the guard condition is still satisfied. Therefore, for any statement inside the loop, the belief $i \geq 0$ implicitly holds although it is not explicitly present as a condition in the code. If we consider these implicit beliefs and the explicit guard conditions together then we can extract the correct set of preconditions as expected (Figure 1.1).

The table below the listing in Figure 1.1 shows the expected preconditions, explicit conditions, implicit beliefs and the combined set of preconditions extracted for this API.

Figure 1.2 shows an example on creating monthly login chart, that uses *org.jfree.chart* library. In line 3, a *XYPlot* object *plot1* is instantiated. Then, in line 6, we have the API method of interest *CombinedDomainXYPlot.add()*.

```
1 private byte[] createMonthlyLoginChart (int width, int height) {
2  ...        // code for initializing dataset1, renderer1, axis1, domainAxis
3 XYPlot plot1 = new XYPlot(dataset1, null, axis1, renderer1);
4  ...                          // implicit belief : plot1 != null
5 CombinedDomainXYPlot cplot = new CombinedDomainXYPlot(domainAxis);
6 cplot.add(plot1, 3);          // implicit belief : ARG2 == 3 ==> ARG2 >= 1
7  ... }
```

Expected Precondition(s)	$= \{\ plot1 \neq null, ARG2 \geq 1\ \}$
Explicit Conditions(s)	$= \{\ \emptyset\ \}$
Implicit Belief(s)	$= \{\ plot1 \neq null, ARG2 == 3 \implies ARG2 \geq 1\ \}$
Combined Precondition(s)	$= \{\ plot1 \neq null, ARG2 \geq 1\ \}$

Figure 1.2: Data-related code elements to derive implicit beliefs, API of interest is highlighted in bold font.

In this source code, there are no explicit guard conditions present before the API call. Traditional mining approach will not be able to extract any condition at program point *cplot.add(plot1, 3)* in line 6. As mentioned before, in line 3, plot1 is instantiated and then the object is used as a parameter to the API method of interest. It is implicitly known if an object is instantiated, it can no longer be null. Using this implicit belief we can infer one precondition that the first argument of the API *add()* method is non-null. In case of the second argument, a constant value 3 is passed. We can infer that the value of this argument is equal to 3. If a constant is passed to the API, we refer to any implicit beliefs derived from its value as *constant propagation* implicit belief. To attain a precondition from this implicit belief we will need support from other call sites. Now assume that previously mined call sites provided a condition that the second argument is greater than or equal to 1 for the same API, then we verify this call site for implicit belief $ARG2 == 3 \implies ARG2 \geq 1$. As the statement is true for the call site shown in Figure 1.2, we would be able to strengthen the condition that the second argument is greater than or equal to 1 from this call site. We depend on mined conditions from other call sites for the same API in order to refrain from introducing too many false positives in case of constant propagation related implicit beliefs.

The expected preconditions, explicit conditions, implicit beliefs and the combined set of preconditions for this API is shown below the listing in Figure 1.2. The final row of the table in Figure 1.2 shows the combined set of preconditions that can be extracted for the API *add()* at line 13, that is same as the expected set of preconditions shown in the first row of the same table.

1.1 Contributions

The contribution of the thesis include:

- the notion of implicit beliefs and its usage for precondition mining,
- lightweight source code analyses to infer and propagate implicit beliefs, and
- an evaluation of our techniques on real-world programs consisting of 14,000+ projects (over 350 millions lines of code) and 7 libraries. The result shows that we achieve overall 60% in precision

and 69% in recall and such accuracy is relatively improved by 32% in precision and 78% in recall compared to base usage-based mining approach for these libraries.

This work is adopted from OOPSLA 2017 paper [18], which is written in collaboration with Dr. Hoan Anh Nguyen, Dr. Tien N. Nguyen and Dr. Hridesh Rajan.

The rest of this article is organized as follows. In Chapter 2, we present our approach. Then, Chapter 3 presents the empirical evaluation, Chapter 4 describes related work and Chapter 5 concludes the thesis.

CHAPTER 2. APPROACH TO EXPLOIT IMPLICIT BELIEFS TO IMPORVE USAGE-BASED SPECIFICATION MINING

This chapter describes our approach to detect implicit beliefs from programs and use such implicit beliefs to infer preconditions of APIs from API usages in a large corpus of source code. We first present an overview of our approach to derive the implicit beliefs from certain code elements, propagate them to the API call sites subsequently maintaining the context and infer API preconditions using these implicit beliefs. The implicit belief-related components in this chapter can be used for any code corpus to automatically mine preconditions to leverage the existing usage-based mining approach. We then present our systematic classification of 35 code elements containing the implicit beliefs. Finally, we provide detailed descriptions of algorithms for analyzing six representative code elements. We also present our technique using 1-CFA (1-level control flow analysis) to infer a richer set of preconditions from both implicit beliefs and explicitly checked conditions.

2.1 Approach Overview

Existing usage-based mining approaches extract only the explicit guard conditions present before an API call and considers the most frequent conditions of such kind as preconditions of those APIs. If an API is from a library that is not commonly used and does not have enough usages, then it becomes difficult to find explicit conditions for an API. Absence of such conditions will result in mining few or no specifications for an API. However, the language constructs and semantics can potentially serve as a means to detect conditions that are implicitly present in the code corpus before an API call. Facts that can be directly derived from the code are known as *beliefs* [11]. Some of these beliefs are explicitly checked in the code, while some are implicit.

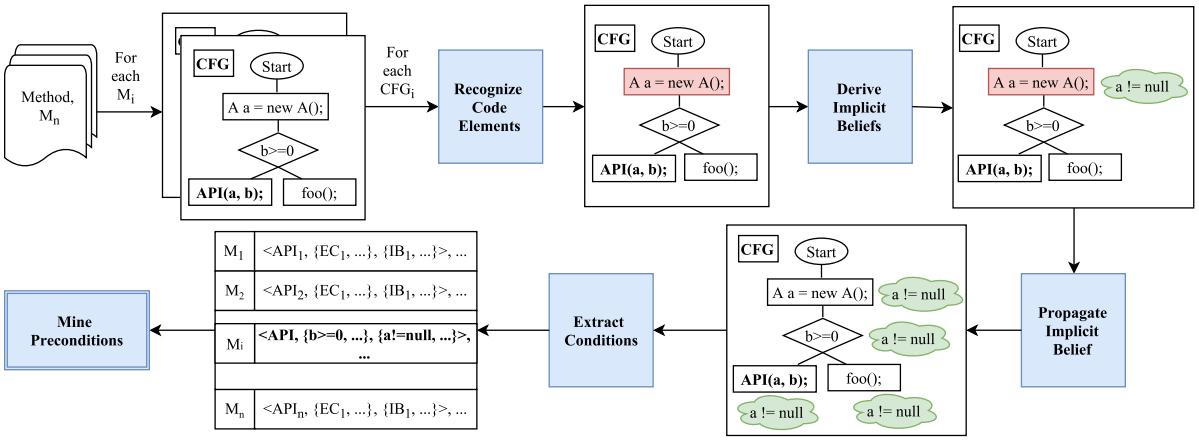


Figure 2.1: Approach overview: From the input code corpus, we build a control flow graph for each method. The implicit beliefs are derived by recognizing the corresponding code elements. Each implicit belief is propagated in subsequent paths. Preconditions are then inferred from explicit conditions and implicit beliefs guarding API calls.

Definition 1. An **Implicit Belief** is the knowledge about the program that can be implicitly derived from the language constructs or semantics through specific code elements.

An example of such implicit belief is the fact that an object must not be `null` after it is created with a class instance creation operation. The implicit belief does not depend on the frequency of occurrences in the code corpus. Recognizing a code element with a corresponding implicit belief confirms that it will hold in subsequent nodes given a control flow graph (CFG) unless the belief is invalidated. Each implicit belief can be invalidated and in result disallowing propagation to following CFG nodes. The notion regarding invalidation and propagation of implicit belief is further discussed in Section 2.3. An implicit belief can be used to mine preconditions given a usage corpus in addition to explicit guard conditions.

Figure 2.1 shows the overview of our approach to infer implicit beliefs and use them to mine preconditions in client code corpus of the APIs. Our approach makes use of both explicitly-checked conditions and implicit beliefs from control flow graphs (CFGs). For each method in the client code, the corresponding CFG is built. The CFG may contain one or more API method calls for each method. Therefore, the final result consists of explicit guard conditions and implicit beliefs for each API call inside a method enriching the precondition set. If an API is control-dependent on an explicit guard

conditions, then that condition will belong to the set of explicit guard conditions for the API. To infer a precondition from an implicit belief we need the following steps:

Recognition of Code Elements: Execution of certain code elements implies corresponding implicit belief directly at a program point of interest. To detect an implicit belief, the approach needs to identify these code elements and relate to the corresponding implicit belief. We refer to this as recognition of code elements to generate belief. In general, the approach looks for specific set of code elements in a node, n_i of the CFG that contains a belief according to the language constructs. For instance in the partial CFG shown in Figure 2.1, we observe the class/object instance creation node `A a = new A()` after the start node and it is recognized (highlighted in red) as an indicator of such a code element containing definition of `a`. In our implementation, we build CFGs at both statement and expression levels so that each CFG node corresponds to a distinct syntax construct. For the above code fragment `A a = new A()`, the CFG will contain one node for class instance creation and one node for assignment and a CFG node cannot be both instance creation and definition at the same time.

Derivation of Implicit Belief: Different code elements correspond to different implicit beliefs. Recognition of code elements of certain type helps us to pinpoint the associated implicit belief. In the previous step, our approach recognizes the object instance creation as a code element that can help in deriving an implicit belief. From the Java language semantics, we know that if an object is instantiated then it cannot be `null`. This implies the implicit belief that the object `a` in Figure 2.1 is `non-null`. Once we derive an implicit belief from a recognized code element, we annotate the node (highlighted in green) with the derived implicit belief, i.e., it is added as `GEN` of the corresponding node, n_i. Each node n_i also maintains a `KILL` set that contains the implicit beliefs invalidated at n_i. Therefore, in the example shown in Figure 2.1, `GEN` of the *class instance creation* node `A a=new A()` contains the implicit belief `a!=null` and `KILL` is empty.

Propagation of Implicit Belief: The next step after deriving an implicit belief from a code element is to propagate the implicit belief in appropriate subsequent path. The purpose of this step is to define a set of rules using `GEN` and `KILL` to propagate the implicit belief along the control flow path maintaining the validity of the implicit belief. Let the node `A a=new A()` from the Figure 2.1 be our current

node with `GEN` being `a!=null` and `KILL` is empty. To propagate the implicit belief we need to know incoming implicit belief IB_{in} for the current node. For any node n_i, IB_{in} contains set of implicit beliefs that any previous node in CFG may have. To propagate the implicit belief in any successor node of the CFG we make use of corresponding `GEN`, `KILL` and IB_{in} and refrain from propagating any invalidated belief. The subsequent nodes of the CFG shown in the Figure 2.1 are the decision node `b>=0` and action nodes `API(a, b)` and `foo()`. All of these nodes contain the same reaching implicit belief `a!=null` since none contains a re-assignment of `a` and resulting in changing the KILL set of the node where redefinition may occur. Therefore it is safe to propagate the implicit belief `a!=null` in all three nodes in Figure 2.1. However if any of these nodes contained a re-assignment of the variable `a` then computing the reaching implicit belief through `GEN` and `KILL` would help us invalidate the propagation of the implicit belief down that path. Note that along with the implicit belief `a!=null`, the guard condition `b>=0` is also present as API method invocation *API(a, b)* is control-dependent on this node. The mined preconditions will consist of both conditions.

2.2 Classification of Code Elements to Derive Implicit Beliefs

Implicit beliefs are present in the source code and can be recognized by the language syntax and semantics of certain code elements. The key challenge to use implicit belief in usage-based mining is to identify the precise beliefs to look for. In this section we describe our systematic classification of code elements containing the implicit beliefs. The classification is shown in Figure 2.2. At the top level, we classify code elements into 3 classes involving data, computation or control elements in a program. We then further classify each class into sub-classes until we reach code elements containing implicit beliefs.

2.2.1 Implicit Belief Derived from Data Code Elements in Programs

A data element could be a constant or a variable of primitive, array or reference type. We have the following implicit beliefs.

B1. Constant propagation: This code element looks for any constant data passed directly to an API. Recognizing this code element tells us about a concrete value for that API argument. To derive an

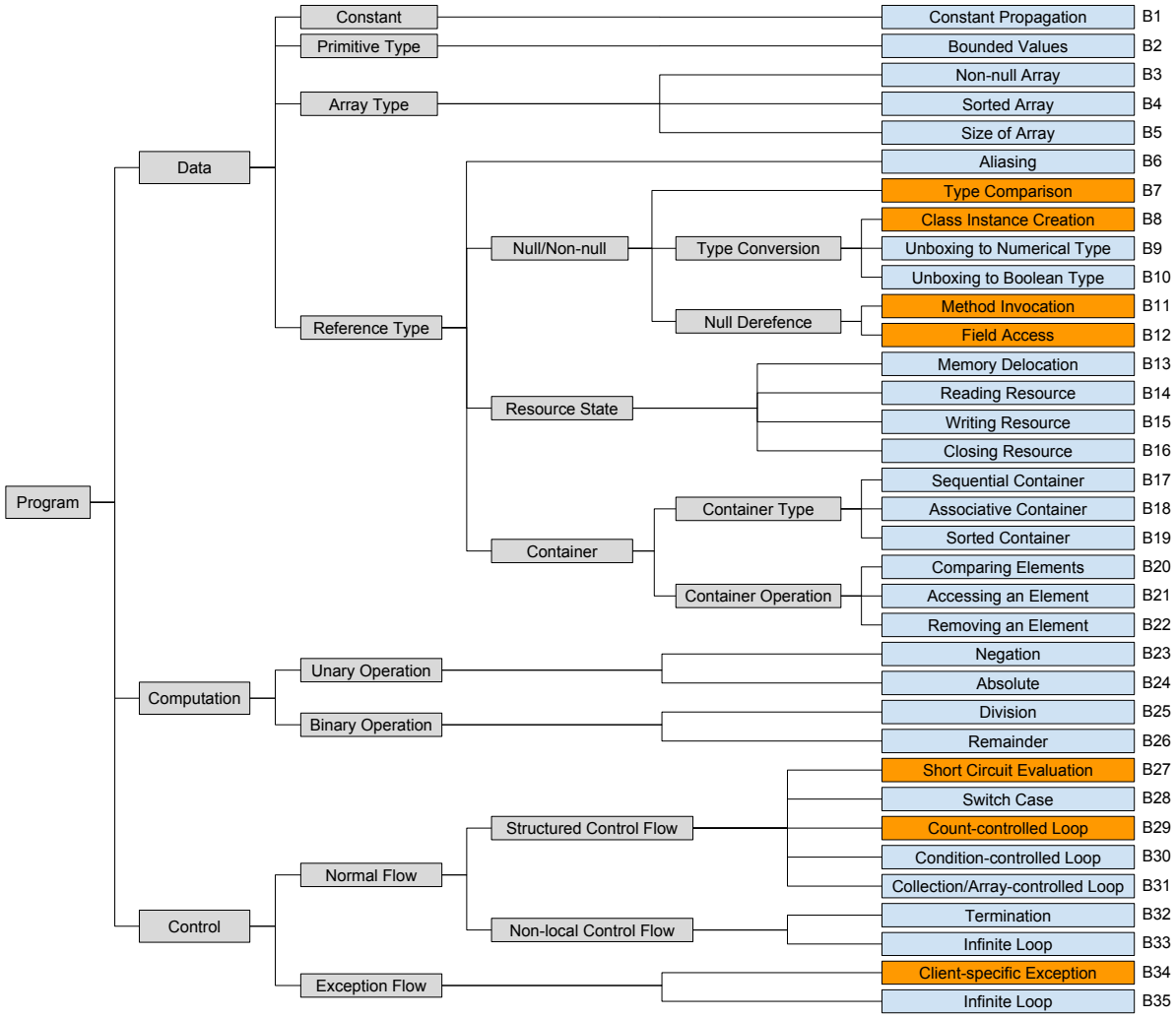


Figure 2.2: Classification of code elements to derive implicit beliefs. Code elements that are realized in this thesis are highlighted.

implicit belief from this code element we also make use of other call sites that provides candidate pre-conditions for this API. If we find a similar precondition then we use the current call site to strengthen the precondition. By similar condition we mean a condition related to same API component and the condition holds if current constant value is substituted with the API component found at other call sites.

B2. Bounded values: If an API uses a numeric type, e.g., `int`, variable then the initialization of the variable serves as the point to recognize the precise code element. The implicit belief we de-

rive is that the variable being used by the API must a hold a value that is in the range between `Integer.MINVALUE` and `Integer.MAXVALUE`.

B3. Non-null array: The code element array access can be recognized to derive the implicit belief that the array being used is non-null. The code element of initialization of an array can be used to derive same implicit belief. **B4. Sorted array**: If the code element indicates that the array is sorted, then derived implicit belief is the value of each element in the array is less or equal compared to the next element stored in the array. **B5. Size of array**: The code element array access can be used to recognize and imply the belief that the size of array being accessed is greater than zero and the index is less than that size.

B6. Aliasing: When an alias of an object is present, recognizing non-null property for an object can imply that the aliased object is non-null as well.

B7. Type Comparison: If a reference type object is compared to check whether it is an instance of some class, then the code element can be recognized to derive the implicit belief that the object is non-null.

B8. Class Instance Creation: With this code element, the resulting object is guaranteed to be non-null.

Unboxing conversion: This type of conversion changes expressions of reference type to corresponding expressions of primitive type by invoking necessary method. Recognition of reference type in a conversion can help to derive that the object is not `null` as an implicit belief. The unboxing operation can happen to convert to the primitive types: **B9. Unboxing to numerical type** and **B10. Unboxing to boolean type**.

Null Dereference: A successful execution of dereferencing an object implies that the object is not `null`. **B11. Method Invocation** and **B12. Field Access** are two concrete code elements of null dereferences where the implicit belief is that the receiver object of the method invocation or the qualifier object of the field access is not `null`.

Resource State: Resources (e.g., memory, file, I/O etc.) have state-related code elements that can be recognized by knowing the implied preconditions. **B13. Memory deallocation**: To avoid resource leak once a memory allocation occurs, memory must be freed/delocated after use. Therefore, detection of memory deallocation code element can help derive the implicit belief that it has the memory allocation

preceding to it. **B14. Reading resource, B15. Writing resource and B16. Closing resource**: If we can detect a resource is being read, written/appended or closed, we can derive that the resource must exist or has been opened.

Container: Depending upon the container used in a program we can detect certain code elements to derive implicit beliefs. A container has various properties depending on its type. **B17. Sequential container**: Identifying a container as sequential indicates that values of the container can only be accessed sequentially, for example, through an iterator. Depending on different type of sequential container, e.g., `Set`, we can further derive that the values contained has no duplicate. **B18. Associative container**: Finding an associative container, e.g., `ArrayList`, implies that values can be accessed randomly through some index/key depending on the type of associative container. **B19. Sorted container**: Detection of a sorted container in a program implies that the values in the container are sorted. The code elements that perform different types of operation on a container could associate with certain implicit beliefs. **B20. Comparing elements**: This code element indicates the implicit belief that the container being processed is non-null. **B21. Accessing an element**: Identification of element access in a container, e.g., `next()`, implies that the container has more elements to be accessed because the program will throw `NoSuchElementException` otherwise. **B22. Removing an element**: The code element to modify an iterating container, e.g., `Iterator.remove()`, indicates that this call must be preceded by `Iterator.next()` to avoid `ConcurrentModificationException`.

2.2.2 Implicit Belief Derived from Computation Code Elements in Programs

Next, let us look at the code elements involving computation on data and how we can derive implicit belief by recognizing such code elements.

B23. Negation operation: Recognition of negation operation as a code element implies that if the result of the operation is not a number, then the operand can not be a number. **B24. Absolute operation**: Identifying absolute operation as a code element implies that if the result of the operation is not a number, then the operand can not be a number.

B25. Division operation: The code element division by a number implies that the denominator can not be zero. **B26. Remainder operation**: The computation of remainder implies that the denominator can not be zero.

2.2.3 Implicit Belief Derived from Control Code Elements in Programs

A program may contain different type of control flow and depending on the type of control flow found inside the program, it is possible to find code elements that lead to an implicit belief. A control flow in a program could be a normal or an exceptional control flow. A normal control flow leads to a normal exit point in the program while an exceptional control flow leads to an exception exit point. A normal control flow can be further classified as non-local or structured depending on if it contains statements, such as `break`, `continue` and `return`, that cause the flow of the execution to jump out of the given context or not.

In the structured control flow, we identify different kinds of code elements that could lead to implicit beliefs. One is the **short circuit evaluation** of the condition and one is the initializer and updater of the **count-controlled loop** among others.

B27. Short Circuit Evaluation: The branching node consisting of at least two operands joined by boolean operator can be detected as a potential code element based on the evaluation strategy used. If language considers short circuit evaluation technique, then we can derive the implicit belief that second argument will only be evaluated if evaluating the first argument is not enough to determine value of the whole expression. For example, evaluating the second operand of a logical conjunction implies that the first operand is evaluated to true. **B28. Switch Case**: Detection of switch case branches derives the implicit belief that the conditions in each case are exclusive. Switch case can also be realized in terms of if-else if branches. Hence we can detect same implicit belief in case of detection of this code element.

B29. Count-controlled Loop: In case of count-controlled loop, we observe the loop initializer and updater expressions of the counter. From these elements, we could derive the implicit belief that the counter is greater than or equal to the initialized value or the counter is less than or equal to the initialized value.

B30. Condition-controlled Loop: If a loop is condition controlled and we detect the code element to collect data before entering the loop, and loop body also collects subsequent data to continue the loop, then we can derive the implicit belief that any condition related to collecting data before entering the loop is also relevant inside the loop.

B31. Collection/Array-controlled Loop: If we detect a loop is collection/array controlled, then it can be derived that the collection/array is not null for any statement inside the loop.

Non-local Control Flow: In non-local control flow, the program behavior allows separation of cross-cutting concerns. In most languages, mainly transfer statements (e.g., break, continue, return, try-catch-finally) are used to achieve this purpose. **B32. Termination**: This code element recognizes sequential unreachable code after termination (e.g., return). In this case derived implicit belief is to ignore any expressions present in the unreachable code. **B33. Infinite Loop** in non-local control flow: If we detect infinite loop then simple implicit belief to derive is obvious true condition of the loop should not be considered. Identifying infinite loop with a transfer statement implies the belief that the guard condition of the transfer of flow should be considered if the condition is relevant to some API.

The last two code elements for deriving implicit beliefs are from exception control flow. Implicit beliefs derived from **B34. Client-specific Exception**s helps remove the guard conditions that are client-specific. Existing usage-based approaches consider conditions guarding the exceptional control flows as the preconditions of any API calls in the normal control flow counterpart. However, if the exception is specific to the project and cannot be thrown by the API, the guard condition should not be considered as an API precondition. We derive the implicit belief that any condition leading to throwing a project-specific exception is irrelevant to the API call. This implicit belief will not help to infer more preconditions, but it could help reduce the false positives.

B35. Infinite Loop in exception control flow: An interesting exceptional case that we observe in a program is running a process (e.g., server) infinitely in a loop until an exception occurs. Detecting this code element and presence of an API in the loop derives the implicit belief that the negation of the guard condition of the exception is the precondition to the API, if the condition is relevant.

Algorithm 1: Object Instance Creation

Input: CFG, API
Output: Implicit beliefs, IB_{API}

1 **foreach** *node n in topological order in CFG* **do**
2 $IB_{in}[n] \leftarrow \bigcap_{p \in n.Pred} IB_{out}[p]$
3 **if** *n is an object instance creation* **then**
4 $GEN \leftarrow \text{deriveIB}(n)$
5 $KILL \leftarrow \emptyset$
6 **else if** *n.isDef()* **then**
7 $GEN \leftarrow \emptyset$
8 $var \leftarrow \text{extractVariable}(n)$
9 $KILL \leftarrow \text{Defs}[var]$
10 **else**
11 $GEN \leftarrow \emptyset$
12 $KILL \leftarrow \emptyset$
13 $IB_{out}[n] \leftarrow GEN \bigcup (IB_{in}[n] \setminus KILL)$
14 **if** *n.isAPI()* **then**
15 $IB_{API} \leftarrow IB_{in}[API]$

Algorithm 2: Null Dereference

Input: CFG, API
Output: Implicit beliefs, IB_{API}

1 **foreach** *node n in topological order in CFG* **do**
2 $IB_{in}[n] \leftarrow \bigcap_{p \in n.Pred} IB_{out}[p]$
3 **if** *n is dereferencing a variable* **then**
4 $GEN \leftarrow \text{deriveIB}(n)$
5 $KILL \leftarrow \emptyset$
6 **else if** *n.isDef()* **then**
7 $GEN \leftarrow \emptyset$
8 $var \leftarrow \text{extractVariable}(n)$
9 $KILL \leftarrow \text{Defs}[var]$
10 **else**
11 $GEN \leftarrow \emptyset$
12 $KILL \leftarrow \emptyset$
13 $IB_{out}[n] \leftarrow GEN \bigcup (IB_{in}[n] \setminus KILL)$
14 **if** *n.isAPI()* **then**
15 $IB_{API} \leftarrow IB_{in}[API]$

2.3 Algorithms for Inferring and Propagating Implicit Beliefs

We have chosen six implicit beliefs that are most promising in terms of the type of preconditions the oracle holds to maximize the coverage. For each belief, we implemented the corresponding component to realize the implicit belief recognition and propagation.

2.3.1 Object Instance Creation (OIC)

The algorithm for this component is in Algorithm 1. The code element to recognize implicit belief is object instance creation and implicit belief to propagate is that the instance can not be `null`. Line 3 of the algorithm recognizes the code element, e.g., in line 3 of Figure 1.2, and line 4 of the algorithm derives the implicit belief `plot1!=null` through the code element. The set GEN of code instance creation node contains this belief and KILL is empty (Algorithm 1, lines 4–5) as there is no other redefinition present. To propagate this implicit belief to any subsequent node in CFG, the algorithm uses `GEN`, `KILL` and incoming implicit beliefs IB_{in}. Incoming implicit beliefs are the set of implicit belief propagated from the predecessor nodes (line 2). If a node contains a definition (lines 6–9) then the variable is extracted to kill any implicit belief related to the variable before calculating IB_{in} for that node. `GEN` remains empty in this case as no implicit belief is generated. In Figure 1.2 line 5, we detect

```
1  protected void drawSecondaryPass(...) {
2    ...
3    double tY1 = rangeAxis.valueToJava2D(...);
4    if (getItemShapeVisible(series, item)) {
5      Shape shape = getItemShape(series, item);
6       ... }
7    double xx = tX1;
8    ...
9    int index = plot.getRangeAxisIndex(rangeAxis);
10  ... }
```

Figure 2.3: Example for Null Dereference code element.

another object instance creation node, where `GEN` contains the new implicit belief `cplot!=null` and `KILL` is empty. In this case, we have IB_{in} containing `plot1!=null` and using the computation in line 13 of Algorithm 1, we get the outgoing implicit belief set IB_{out} to be `plot1!=null` and `cplot!=null`. Since the next node in CFG (Figure 1.2 line 6) is an API, we extract the relevant condition `plot1!=null` (lines 14–15).

2.3.2 Null Dereference (ND)

Once a variable is dereferenced to invoke a method or access a field, that code element is recognized by this component. Algorithm 2 derives the implicit belief that the variable can not be `null`. In Figure 2.3 line 3, the variable *rangeAxis* is dereferenced to invoke a method. Then this node is used to derive the implicit belief that *rangeAxis* is `non-null` following the rules from Algorithm 2 lines 3–5. To invalidate the propagation of this implicit belief a re-assignment of the same variable needs to occur (lines 6–9) in a successor node of the CFG. In the given example (Figure 2.3), this does not occur. Therefore, it is safe to propagate the implicit belief until we reach the API node `plot.getRangeAxisIndex(rangeAxis)` in line 9 of Figure 2.3. In lines 14–15, Algorithm 2 stores the implicit belief `rangeAxis!=null` because it is a guard condition for this API.

Algorithm 3: Type Comparison

```
   Input: CFG, API
   Output: Implicit beliefs, IB_API
1  foreach node n in topological order in CFG do
2      IB_in[n] ← ⋂_{p∈n.Pred} IB_out[p]
3      if n is an instanceof check then
4          GEN ← deriveIB(n)
5          KILL ← ∅
6          foreach node m in True branch of n do
7              m.setTrueBranch()
8      else if n.isTrueBranch() then
9          if n.isDef() then
10             GEN ← ∅
11             var ← extractVariable(n)
12             KILL ← Defs[var]
13         else
14             GEN ← ∅
15             KILL ← ∅
16     else
17         IB_out[n] ← ∅ continue
18     IB_out[n] ← GEN ⋃ (IB_in[n] \ KILL)
19     if n.isAPI() then
20         IB_API ← IB_in[API]
```

Algorithm 4: Count Controlled Loop

```
   Input: CFG, API
   Output: Implicit beliefs, IB_API
1  foreach node n in topological order in CFG do
2      IB_in[n] ← ⋂_{p∈n.Pred} IB_out[p]
3      if n is a count-control loop then
4          init ← n.Initializer
5          incr ← n.Increment
6          GEN ← deriveIB(init, incr)
7          KILL ← ∅
8          foreach node m in body of loop do
9              m.setTrueBranch()
10     else if n.isTrueBranch() then
11         if n.isDef() then
12             GEN ← ∅
13             var ← extractVariable(n)
14             KILL ← Defs[var]
15         else
16             GEN ← ∅
17             KILL ← ∅
18     else
19         IB_out[n] ← ∅ continue
20     IB_out[n] ← GEN ⋃ (IB_in[n] \ KILL)
21     if n.isAPI() then
22         IB_API ← IB_in[API]
```

```
1 public String getChartViewer(...) {
2   ...
3   ValueAxis axis = plot.getRangeAxis();
4   if (axis instanceof NumberAxis) {
5     int i = plot.getRangeAxisIndex(axis);
6     ... }
7  ... }
```

Figure 2.4: Example for Type Comparison.

2.3.3 Type Comparison (TC)

The algorithm for this component is shown in Algorithm 3. If an object is checked whether it is an instance of some class, it indicates the code element needed for this component. In Figure 2.4 line 4, we observe such a check. In our current solution, we only look for `instanceof` checks as part of condition expressions of `if` statements. More sophisticated static analysis could be used to track the flow from any `instanceof` expressions into the conditions guarding API calls. According to

Algorithm 3 line 3, this node will return true for the CFG built from the method `getChartViewer()` in Figure 2.4. The node is used to derive the implicit belief that `axis` is not `null`. In Algorithm 3 line 4, we see the derivation of this implicit belief in `ib` that sets `GEN` for the node. `KILL` is empty as this does not invalidate any implicit belief. To invalidate this implicit belief, we need a redefinition of the variable that is present in true branch (Algorithm 3 lines 8–12). The implicit belief does not hold outside the true branch (Algorithm 3 lines 16–17). If the node is in true branch and it does not involve a re-assignment of the same variable, e.g., line 5 of Figure 2.4, then the implicit belief is safe to propagate as shown in Algorithm 3 line 18. In this example (Figure 2.4 line 5), the next node in CFG being an API method call node `plot.getRangeAxisIndex(axis)`, receives the implicit belief that `axis` is `non-null`, hence it is extracted as a condition of the API (lines 19–20).

2.3.4 Count Controlled Loop (CCL)

The count control loop initializes the counter with an initial value and increases/decreases it after each iteration, while the guard condition is still satisfied. The initialization and updater of the loop counter together are the code elements necessary for this component. Depending on the fact whether the loop counter is increased/decreased, the algorithm derives the implicit belief that loop counter is greater/less than or equal to the initial value (line 6 of Algorithm 4). In method `getOLSRegression()` in Figure 1.1, the code element for this component is recognized in line 7. The initializer sets counter `i` to `0` and the updater increments it by 1. Therefore, our algorithm derives the implicit belief that $i \geq 0$ (Algorithm 4 lines 4–6). In this case, the belief should not be propagated to any path that is outside the scope of the loop (Algorithm 4 lines 18–19). Another factor to invalidate the implicit belief is the re-assignment of the loop variant. It is checked in lines 10–14 of Algorithm 4, and implicit belief is not propagated if the decision holds. In the example shown in Figure 1.1, the next node of the CFG is within the scope of the count controlled loop and no re-assignment is present, therefore the implicit belief $i \geq 0$ is propagated to the method call node at line 8 of the figure. Since the node is an API call, Algorithm 4 stores the belief as an extracted condition for the API (lines 21–22).

Algorithm 5: Short Circuit Evaluation

Input: CFG, API
Output: Implicit beliefs, IB_{API}

1 **foreach** *node n in topological order in CFG* **do**
2 **if** *n is conjunction or disjunction* **then**
3 $op \leftarrow n$.Operator
4 $l \leftarrow n$.LeftOperand
5 $r \leftarrow n$.RightOperand
6 $GEN \leftarrow$ deriveIB(op, l)
7 $IB_{in}[r] \leftarrow \bigcap_{p \in r.Pred} IB_{out}[p] \bigcup GEN$
8 **if** *r.isAPI()* **then**
9 $IB_{API} \leftarrow IB_{in}[API]$

Algorithm 6: Local Exception

Input: CFG, API
Output: Implicit beliefs, IB_{API}

1 **foreach** *node n in topological order in CFG* **do**
2 **if** *n throws an exception* **then**
3 $e \leftarrow n$.Exception
4 **if** *e is a local exception* **then**
5 $gc \leftarrow$ getGuardCondition(n)
6 **if** *API call is in same branch as n* **then**
7 $IB_{API} \leftarrow \{remove(gc)\}$
8 **else**
9 $IB_{API} \leftarrow \{remove(!gc)\}$

```
1 private void preProcess(...) throws IOException {
2   ...
3   if (reader.getEventType() == PROCESSING_INSTRUCTION && reader.getPITarget() != null) {
4     ... }
5 ... }
```

Figure 2.5: Example for learning Implicit Belief in Conjunctions and Disjunctions.

2.3.5 Short Circuit Evaluation (SCE)

This component focuses on a decision node consisting of at least two operands joined by a boolean operator. The semantics of these boolean operators indicates that the second argument is evaluated only if the first argument does not suffice to determine the value of the expression and so on. That is, in 'e1 && e2' or 'e1 || e2', where e2 calls an API and there is no short-circuit, e1 or !e1 will be the guard condition of the API call, respectively. Line 2 of Algorithm 5 detects such code element, e.g., in lines 3–4 of Figure 2.5. It extracts the operator and operands from the boolean expression. As these code elements are recognized, the derived implicit belief refers to the fact that the left operand being true/false depending on the boolean operator, is the precondition of the expressions in the right operand. In the example in Figure 2.5, the boolean operator is `conditional AND`, which means in line 6 of Algorithm 5, the derived implicit belief is that `reader.getEventType()` must be equal to `PROCESSING_INSTRUCTION` in order to evaluate the right operand. Since the derived implicit belief is only effective in the right operand of the boolean expression, it is not prop-

agated to IB_{out}. In our current implementation, we assume that the expressions in the boolean expression of the node are side-effect free. Thus, it is not necessary to maintain `KILL`. If the right operand contains an API method invocation, the implicit beliefs are extracted as preconditions (Algorithm 5 lines 8–9). In Figure 5 line 4, the right operand `reader.getPItarget()!=null` contains an API call, therefore the implicit belief that we derived is that the left operand is true, i.e., `reader.getEventType()==PROCESSING_INSTRUCTION` is a guard condition of the API call.

2.3.6 Local Exception (LE)

In this component, the code element to look for is an explicit guard condition check followed by throwing local exception. Local exception can be identified automatically if the exception is not a runtime exception thrown by the language, nor an API-specific exception. From a throw statement we extract the exception (Algorithm 6 lines 2–3) and then perform a check to confirm if it is a local exception. The check is performed automatically in two steps. First, we check if the exception is a runtime exception thrown by the underlying language. If not, we further check the API signature to confirm the exception is not an API-specific one. Recognition of both these elements enables us to derive the implicit belief that `gc` is project-specific, therefore should not be used to extract a guard condition as shown in Algorithm 6 lines 4–6. In Figure 1.1, line 3 contains a condition `n<2` that guards a non-runtime exception, `RegressionException`, thrown in line 4. In line 8, the API call `getXValue()` of our interest does not throw the exception `RegressionException` according to its signature. Therefore, we derive our implicit belief that the exception is local regarding the call `getXValue()` and the guard condition is irrelevant to the API. An API call can be present in the true branch (e.g., printing the location of the exception) or the false branch of the guard condition (e.g., subsequent nodes in CFG if the exception is not thrown). Depending on this fact, we consider the guard-condition itself (lines 6–7), or the negation of the condition (lines 8–9) respectively as irrelevant, if our algorithm identifies a local exception.

2.4 1-Level Control Flow Analysis (1-CFA)

To scale to mine preconditions from a large code corpus, usage-based mining techniques, such as [27], extract the conditions within the same procedure calling the API. If the conditions appear in the calling context of the procedure, i.e., its callers, they are not available for those intra-procedural techniques to mine. This can be another challenge while handling sparse data usage. To consider multilevel context in case of usage-based approaches could be very expensive, if large source code corpus is involved. However, addition of single level context sensitivity can allow usage-based approach to consider calling context while processing the target procedure to mine preconditions and still maintain feasibility to process big code.

```
1 public void computeRegression(XYDataset data) {
2   int series = data.getSeriesCount();
3   ...
4   for (int i = 0; i < series; i++) {
5     regression = getOLSRegression(data, i);
6     ... }
7 ... }
```

Figure 2.6: Example of Caller computeRegression() containing context sensitive Callee getOLSRegression().

To find out the calling context in an object-oriented language, the target of the function call object.method() depends on the value that flows to the expression object. The value problem "To which values may the expression `m()` evaluates?" is an undecidable problem. In this circumstance, we use the class of algorithms known as k-Level Control Flow Analysis (k-CFA) to solve this value problem in a conservative way [37]. The CFA algorithms compute conservative over-approximations, i.e., if a CFA says that the procedure `m()` is invoked at some call site, then it may be invoked at that call site. If `m()` can't actually be invoked, then it is a false positive. We use similar concept to build a dictionary consisting of callee–caller mapping, where map stores callers corresponding to a callee. We consider only single level context sensitivity, which is similar to 1-CFAs. However, our 1-CFA algorithm by being conservative in static analysis, may lead to overestimation of preconditions due to infeasible paths, path-sensitivity, etc. In Algorithm 7, lines 1–2 build this dictionary for each project p given a dataset P. Next, in lines 3–4 of the algorithm, a control flow graph is built for each method m.

Algorithm 7: 1-Level Control Flow Analysis

```
   Input: Collection of projects P
   Output: Extracted conditions C
1  foreach project p ∈ P do
2      D ← buildDictionaryOfCallerCallee(p)
3      foreach method m in p do
4          G[m] ← buildCFG(m)
5      foreach method m in p, m calls API do
6          g = G[m]
7          EC_API ← extractEC(g, API)
8          IB_API ← deriveIB(g, API)
9          foreach c ∈ D.getCallers(m) do
10             EC_API ← EC_API ⋃ extractEC(G[c], m)
11             IB_API ← IB_API ⋃ deriveIB(G[c], m)
12         C ← removeIrrelevant(EC_API ⋃ IB_API)
```

In Figure 1.1, for the API of interest `getXValue(int, int)` in line 8, we observe that the first parameter of the API can be found in caller procedure. Therefore usage-based mining technique can not mine any precondition related to this API component from the procedure `getOLSRegression (XYDataset, int)` defined in lines 1–11 of the figure. In this component we consider only direct callers of the callee reflecting the notion of single level context sensitivity. We consider both implicit beliefs and explicit conditions available in the calling context to mine preconditions for the callee that contains the API call.

In Algorithm 7, while traversing each CFG g of a method (lines 5–12) to mine preconditions using implicit beliefs and explicit conditions, it first extracts explicit conditions and implicit beliefs from the callee method itself (lines 7–8). Then our algorithm further does an optimization to confirm if context information is necessary. In example shown in Figure 1.1, we see that we need the context information for the API method call `getXValue()`, as the first parameter `series` of the API corresponds to the second formal parameter by the same name of callee method `getOLSRegression()`. In such cases, Algorithm 7 uses the dictionary to get the set of callers for this callee in line 9. Say we have a caller `computeRegression(XYDataset)` shown in Figure 2.6 that contains the callee method `getOLSRegression()` in line 5. We are interested in the context of the parameter `i` from the caller as it is passed to the formal parameter `series` of the callee method. For this caller, our algorithm first extracts the explicit guard condition, $i < data.getSeriesCount()$, and adds it to the set of explicit conditions mined for the API (Algorithm 7 line 7). Next we observe that the callee

`getOLSRegression()` is called inside a count controlled loop (Figure 2.6 lines 4–6), where the loop variant `i` is initialized to zero and increases until the guard condition is true. Therefore, in line 11 the algorithm derives the implicit belief $i \geq 0$. Finally, both these conditions coming from explicit conditions and implicit beliefs ($EC_{API} \bigcup IB_{API}$) are retained, as they are relevant to the API of interest (line 12). Any conditions that are not related to API call's receiver or parameters are considered irrelevant and, thus, are removed.

CHAPTER 3. EMPIRICAL EVALUATION

This chapter presents our experiments and results on evaluating the precision and recall of the mined preconditions using our approach (Section 3.2) and the characteristics of the mined preconditions (Section 3.3). This chapter also shows the contributions of each component in our approach in improving the mining results (Section 3.4).

3.1 Data Collection

3.1.1 Code Corpus

Like any data driven approach, the experimental result of mined preconditions is dependent upon the quality of source code corpus that is used. We have used a large code corpus [2] consisting of 14,785 projects. The large code corpus is curated using Github's social fork system in a way to isolate the low quality projects that are rarely forked. The authors [2] have also mentioned that among the collected projects they have manually excluded projects that share common commit SHAs. This ensures excluding projects that are likely to be forked from other original projects. The corpus contains over 350 million lines of source code where only files written in Java language are considered. Figure 3.1a shows the complete statistics of the used datasets. The dataset includes in total 1,212,124 API methods calls (Figure 3.1b) from 7 different libraries of interest.

3.1.2 Libraries of Interest

There are some JDK libraries (e.g., javax.xml) that are used less frequently than some other common JDK (e.g., java.io) libraries. Besides such libraries, non-JDK libraries are also less frequently used and less studied compared to popular JDK libraries. These non-JDK libraries can directly depend on JDK libraries and other non-JDK libraries. Let us use the term *leaf library* to denote a library that de-

Projects	14785
Total Source Files	1938865
Total Classes	2442288
Total Methods	17378637
Total SLOCs	352312696
Total Method calls	69374374

(a) Dataset statistics.

Library	Calls	Preconditions
org.jfree.chart (CHART)	35033	169
org.jfree.data (DATA)	16671	125
org.apache.commons.math (MATH)	34010	20
javax.swing (SWING)	479373	20
org.eclipse.swt.widget (SWT)	328174	56
weka.core (WEKA)	44047	28
javax.xml (XML)	274816	48

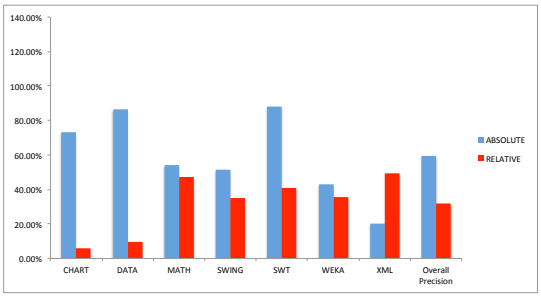
(b) Library API usages and ground truth.

Figure 3.1: Dataset, library API usages and ground truth.

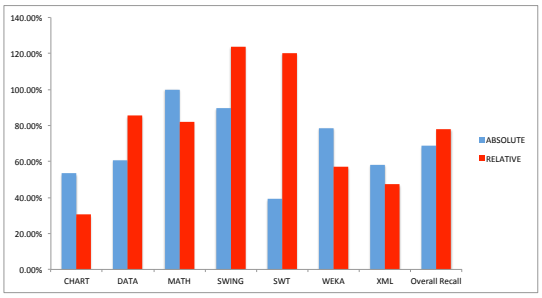
pends only on JDK libraries and itself. To determine the specifications for leaf libraries, it is sufficient to look at the classes of the library and JDK classes it may use. The non-leaf libraries can depend on any JDK or non-JDK, leaf/non-leaf libraries making those more complex in nature and in terms of building specifications. Therefore we concentrated on building the oracle for only leaf libraries in comparison to other non-JDK, non-leaf libraries. Another criterion of choosing the libraries from this list is that the chosen libraries achieve different purposes in general such as chart management, mathematical computation, graphical interface, machine learning computation, etc. All the chosen libraries are open-source, which is another necessity for us to build the ground truth consisting of preconditions for these libraries. In this experiment, we have chosen 2 JDK and 5 non-JDK leaf libraries. These libraries are: org.jfree.chart (CHART), org.jfree.data (DATA), org.apache.commons.math (MATH), javax.swing (SWING), eclipse.swt.widget (SWT), weka.core (WEKA), and javax.xml (XML). We use the short form of the library names throughout rest of the thesis.

3.1.3 The Ground-truth

To determine the accuracy of the mined preconditions from our approach, we built the ground-truth of preconditions for the most frequently-used API methods from the 7 chosen libraries. The top APIs for each library is determined by the number of times they are called within the dataset. We examined the documentation and implementations of the APIs of interest and any related APIs in the same class or project if needed to come up with their preconditions. In case of interface or abstract class, we use the idea of supertype abstraction [20] to deduce the preconditions of APIs. We examined the available classes implementing those interfaces or abstract classes instead to determine the required preconditions. The implementing classes provide the means to statically determine



(a) Precision and improvement.



(b) Recall and improvement.

Figure 3.2: Precision and recall of our approach using implicit beliefs on 7 libraries. The blue bars show the absolute precision/recall and the red bars show the relative improvement over the base approach.

all possible preconditions for each API method call. The idea is that if a precondition for an API is true for all implementing classes, then that precondition can be considered as the precondition of the API in the interface or abstract class. The ground-truth and supplemental materials are available here: https://samanthasyeda.github.io/oopsla2017/.

3.2 Accuracy

We ran our approach on the dataset described in Section 3.1. Mined preconditions are compared with the ground-truth to determine the accuracy of the result in terms of precision and recall. Precision is the ratio between the number of true positive preconditions and the number of total mined preconditions. Recall is the ratio between the number of true positive preconditions and the number of total expected ones. The usage-based mining approach without using implicit beliefs [27] is used as the base case to show the absolute and relative improvement we achieve. The components for inferring implicit beliefs added to traditional usage based mining are the following: object instance creation, type comparison, null dereference, count-controlled loop, short circuit evaluation and local exception, and 1-level control flow analysis.

Figure 3.2 shows the absolute values and relative improvements in precision and recall for 7 libraries. Our approach achieved precision from 21%–88% and recall from 39%–100%. Overall, the precision and recall of our approach are high: 60% and 69%, respectively. The accuracy was improved on all libraries and overall by 32% in precision and 78% in recall.

3.2.1 Performance by Popularity of Library

Next we observed the 7 libraries of interest to analyze the performance of the APIs of these libraries. Among these 7 libraries, we consider SWING, SWT, XML as *more popular* libraries and CHART, DATA, MATH, WEKA as *less popular* libraries as per their usages in Figure 3.1b. The libraries are categorized this way to compare the performance between the more and less popular libraries (Figure 3.2). We observe that for the popular libraries such as SWING, SWT, XML, we achieve a higher relative improvement in precision that ranges from 35%–50%. Comparatively, for less popular libraries (CHART, DATA, MATH, CORE), we also achieve a good relative improvement that ranges from 6%–47%. In terms of recall, for more popular libraries, the relative improvement is from 47%–124% and for less popular libraries, relative improvement is within the range of 30%–85%. We observe that for more popular libraries SWING and SWT, we have significant relative improvement compared to their less popular library counterparts. For SWING and SWT, we achieve 35%–41% relative improvement in terms of precision and 120%–124% relative improvement in terms of recall. Although for the remaining more popular library XML, with a high relative improvement in precision (50%), we achieve comparatively low relative improvement (47%) in terms of recall.

3.2.2 Performance by Data Size

As any data-driven approach, our technique might be affected by the size of the data used in mining. Thus, we performed an experiment to analyze our approach's accuracy with respect to the increasing sizes of mining data. We also compared with the usage-based mining approach. To accomplish this process, we have created several datasets of size S by randomly dividing the projects from full dataset of 14,785 projects. To enable a fair division of projects, the dataset are divided into bins having the same number of S projects where $S = 2^i$ ($i = 9..0$). For each value of i, we ran our approach on all the bins of data and measured the performance via the average precision and recall for all the bins. Then, by decreasing i, we have increased the dataset until reaching the full size of the dataset.

Table 3.1 and Table 3.2 show the accuracy of our approach running on CHART and SWING, respectively. Due to space limit, we have chosen these two libraries as a representative of sparse and popular libraries in our dataset. As expected in Table 3.1, the recall of our approach increases from 4%

Table 3.1: Accuracy comparison on **org.jfree.chart** library with progressive dataset.

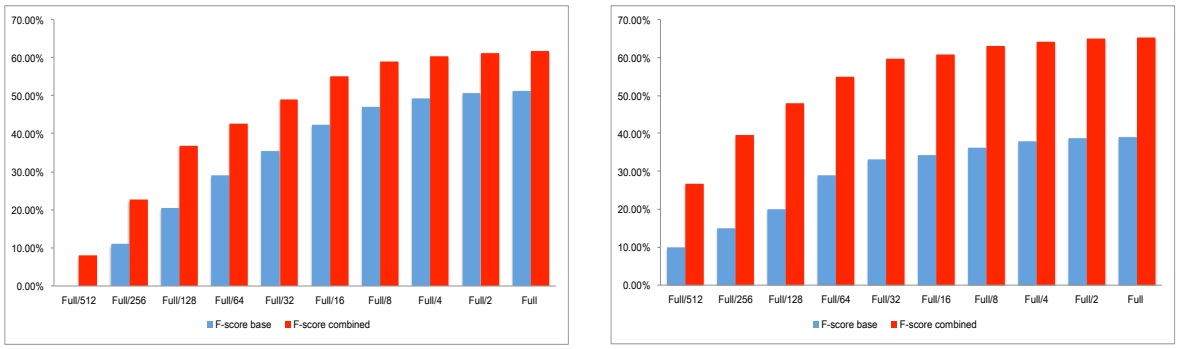
		Data Size									
		Full/512	Full/256	Full/128	Full/64	Full/32	Full/16	Full/8	Full/4	Full/2	Full
Precision	Base	100%	94%	85%	79%	77%	75%	73%	72%	70%	69%
	Ours	100%	98%	92%	87%	84%	81%	80%	77%	74%	73%
Recall	Base	0%	5%	11%	17%	23%	29%	34%	37%	39%	40%
	Ours	4%	12%	22%	28%	34%	41%	46%	49%	51%	53%

Table 3.2: Accuracy comparison on **javax.swing** library with progressive dataset.

		Data Size									
		Full/512	Full/256	Full/128	Full/64	Full/32	Full/16	Full/8	Full/4	Full/2	Full
Precision	Base	100%	87%	77%	67%	58%	54%	50%	45%	41%	38%
	Ours	96%	86%	78%	71%	66%	62%	59%	55%	53%	51%
Recall	Base	5%	8%	11%	18%	23%	25%	28%	32%	36%	40%
	Ours	15%	25%	34%	44%	54%	59%	67%	76%	83%	89%

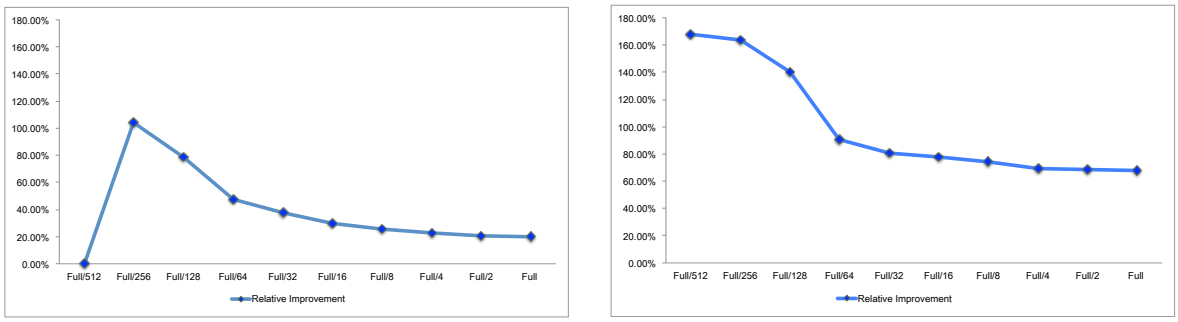
to 53% as more source code is added, because the approach encountered more API usages and was able to derive preconditions of the APIs from implicit beliefs. The precision of our approach decreases from 100% to 73% as data's size is increased. Importantly, despite that the accuracies of our approach and the base approach have the same trend, the recall and precision of our approach are always better than that of the base one. In case of SWING in Table 3.2, we observe a trend that almost resembles to the case of CHART. That is, our approach gains more in terms of precision and recall compared to the base approach. Interestingly, only for SWING library, the approach with Full/512 dataset achieves only 96% in precision. We observed that although our approach extracts more true preconditions initially with a smaller dataset compared to base approach, it also mines some stronger conditions within the bins. The type of stronger conditions our approach mines is explained in details in Section 3.3.2. In the base approach, the smaller bins contain less false positives than what our approach mined.

We have considered `F-score` to aggregate precision and recall for both usage-based and our combined approach. `F-score` is defined as the harmonic mean of precision and recall. It is computed as $F\text{-}score = (2 \times Precision \times Recall)/(Precision + Recall)$. Figure 3.3a and 3.3b shows the result of `F-score` of usage-based and our approach respectively for CHART and SWING libraries. The harmonic mean for base approach increases from 0% to 51% and for our approach 8% to 61% for CHART library. In case of SWING library harmonic mean for base approach gained from 10% to 39% where as our approach again starts from a higher value 26% and increases up to 65%. As seen in Figure 3.3a and 3.3b, our approach gains significantly in `F-score` when dataset size is increased.



(a) Chart (b) Swing

Figure 3.3: Mining performance with progressive data size.



(a) Chart (b) Swing

Figure 3.4: Mining relative improvement with progressive data size.

Relative Improvement. We have also analyzed the relative improvement of accuracy when more data was added. As seen in Figure 3.4a and Figure 3.4b, the relative improvement overall ranges from 20% to 105% for the CHART library while that for the SWING library ranges from 68% to 168%. When the dataset is smaller, the relative improvement in accuracy for all the libraries is higher. The reason is that when the data size starts increasing, much more implicit beliefs are derived, leading to high relative improvement. However, the relative improvement slowly decreases as the dataset size progressively increases further. The reason is that after certain data's size, adding more data, the additional implicit beliefs did not add much more knowledge than the beliefs gained from the smaller dataset.

3.3 Analysis on the Characteristics of Mined Preconditions

We studied the characteristics of the preconditions that were correctly-mined, incorrectly-mined and missing from our approach.

3.3.1 Correctly Mined Preconditions

For the APIs in our dataset, correctly mined preconditions falls into three categories: `null` comparison, primitive comparison and method invocation. The first one contains simple preconditions of comparing arguments with `null`: `ARG!=null`. The preconditions in the second category contains arguments of primitive types being compared to constants or other primitive arguments such as `ARG1>=0` and `ARG1<=ARG2`. The last one containing method calls, on receivers such as `!Receiver.hasNext()` or on arguments such as `!ARG.isEmpty()` or on both. Table 3.3 shows the numbers of preconditions for each category correctly mined from the base approach and the additional ones from our approach using implicit beliefs. The numbers in parentheses are the percentages over all expected preconditions in the corresponding category.

Table 3.3: Categories of correctly-mined preconditions.

Library	**Preconditions from Base Approach**			**New Preconditions from Implicit Beliefs**		
	Non-null Comparison	Primitive Comparison	Method Invocation	Non-null Comparison	Primitive Comparison	Method Invocation
CHART	57 (34%)	9 (5%)	3 (2%)	13 (8%)	6 (4%)	2 (1%)
DATA	19 (15%)	7 (6%)	14 (11%)	7 (6%)	23 (18%)	5 (4%)
MATH	0 (0%)	2 (10%)	9 (45%)	0 (0%)	9 (45%)	0 (0%)
SWING	0 (0%)	0 (0%)	8 (40%)	1 (5%)	9 (45%)	0 (0%)
SWT	5 (9%)	0 (0%)	5 (9%)	7 (13%)	2 (4%)	3 (5%)
WEKA	2 (7%)	0 (0%)	12 (43%)	1 (4%)	4 (14%)	3 (11%)
XML	0 (0%)	0 (0%)	19 (40%)	5 (10%)	0 (0%)	4 (8%)
Overall	83 (18%)	18 (4%)	70 (15%)	34 (7%)	53 (11%)	17 (4%)

The result shows that our approach is able to mine additional preconditions that were missed by the base one in all three categories. However, the improvement was mainly in simpler preconditions which involve comparing arguments against `null` or comparing between primitive values. This is due to the fact that the code elements mostly contain implicit beliefs for inferring those kinds of preconditions.

Three in six of the implemented components, Object Instance Creation (OIC), Type Comparison (TC) and Null Dereference (ND), look for `non-null` property of API components. Our approach mined these preconditions for the chosen libraries (Table 3.3) when related implicit beliefs are present.

Count-controlled Loop (CCL) component only infers the conditions checking the index/counter of the loop against its bounds. Local Exception (LE) could only remove incorrectly-mined preconditions but not add correctly-mined ones. Short Circuit Evaluation (SCE) is the only component that could

Table 3.4: Categories of incorrectly-mined preconditions.

Library	Total	Stronger	Weaker	Specific
CHART	33	16	4	13
DATA	12	6	2	4
MATH	17	2	0	15
SWING	16	2	2	12
SWT	3	2	0	1
WEKA	29	15	2	12
XML	109	19	0	90
Overall	219	62	10	147

infer new complex preconditions involving multiple API components. We will show the details of improvements from each component in Section 3.4.

3.3.2 Incorrectly Mined Preconditions

We have examined the incorrectly mined preconditions to find out the reason behind the occurrences of such conditions. Table 3.4 shows these categories of incorrectly mined preconditions for different libraries.

The first category contains incorrectly-mined preconditions which are **stronger** than required. For example, API `parse(InputSource, DefaultHandler)` in class `javax.xml.parsers.-SAXParser` parses the `InputSource` parameter using the specified DefaultHandler parameter. It makes sense from usage point of view to not pass an `null` default handler. Through using implicit belief, the approach will also mine the precondition that the second parameter cannot be `null`. However, that condition is not required by API and, thus, introduces the incorrectly mined precondition. These cases increases the number of stronger preconditions compared to the base usage-based mining approach and, therefore, increases the number of preconditions incorrectly mined by our approach. For the libraries CHART, DATA and SWING, we have results with additional stronger conditions compared to the usage-based mining approach.

We also observed incorrectly-mined preconditions which are **weaker** than required. For example, the mined preconditions is `ARG!=-1` while the required one is `ARG>=0`. However, all of these weaker preconditions came from the explicit guard conditions present in the code instead of from implicit beliefs.

Another major type of incorrect conditions in our result is the **project-specific** conditions. To some extent, they are also stronger than needed. However, we put them in a separate category because these mined preconditions are only frequent in certain projects. As some APIs do not have rich use cases across multiple projects, many project-specific conditions were considered as preconditions for these APIs. Result on XML library suffers from this the most. We examined and found that most of the call sites in the dataset calls the APIs for a document, until reaching the end of the document. Hence conditions such as `hasnext()` or `!isEndDocument()` is not part of specification for most of the APIs of this library, but are frequently present before the calls of the APIs due to the access pattern. Despite that, our implicit belief from local exception component successfully removed the conditions guarding project-specific exceptions, thus lowered the number of incorrect preconditions from project-specific conditions.

3.3.3 Missing Preconditions

Although our approach was able to detect more preconditions through implicit belief compared to existing usage-based mining approach, we still miss preconditions. We have analyzed the missing classes of preconditions that our approach was unable to find given the dataset we have used for 7 libraries of interest. We found that we did not miss any preconditions that the base approach could mine. We have mainly observed three categories of missing preconditions shown in Table 3.5.

Table 3.5: Categories of missing preconditions.

Library	Total	No Check				Infrequent	Private
		Exception Handling	Sem Guarantee No Exception	Intentional Throw	Unintentional Throw		
CHART	79	14	40	11	6	8	0
DATA	49	3	31	9	2	4	0
MATH	0	0	0	0	0	0	0
SWING	2	2	0	0	0	0	0
SWT	34	7	22	3	0	2	0
WEKA	6	0	3	0	0	1	2
XML	20	5	12	0	0	3	0
Overall	190	31	108	23	8	18	2

No Check. The first category of missing cases is those preconditions that do not appear explicitly, or implicitly at all in the code corpus before calling the APIs. The absence of such preconditions are mainly caused by four factors:

- *Exception Handling*: Sometimes, programmers might be unsure about the preconditions of the APIs, thus, they would surround API calls with try statements and catch clauses to handle the exceptions. This is often the reason why code corpus does not have the necessary preconditions for APIs before calling.

- *Semantics Guarantee No Exception*: Sometimes, developers are well aware that the domain semantics of their project guarantees that certain preconditions of an API always hold. For example, if they know the files are always non-empty then, the list of lines read from the files are always non-empty too. Therefore, accessing first element of the list is always possible. In such cases, checking whether precondition is met before calling the API becomes extraneous, resulting in the absence of such preconditions.

- *Intentional Throw Exception*: Programmers often use test cases to ensure code correctness. In such scenario, programmers may intentionally provide cases where the API will surely throw exception. The exceptional cases will confirm the programmer the code behavior is correct. These types of cases usually do not contain any preconditions before calling the API.

- *Unintentional Throw Exception*: The other subcategory that we observed in our code corpus is that programmers incorrectly used the APIs without checking the required preconditions. Incorrect usage of any API results in throwing exceptions. This type of buggy code does not contain the required preconditions.

Infrequent usages. Those preconditions are present in the client code but not frequent enough to be considered as correct preconditions by the mining technique.

Private members. The final category belongs to the type of preconditions involving private/internal members of the APIs, which cannot be accessed outside the implementation of the APIs. They are, therefore, not possible to be observed in the client before calling the API.

3.4 Effectiveness of Single Components

The previous experiments have shown that, our approach with six implemented implicit beliefs and 1-level control flow analysis (1-CFA) overall improves the precondition inference result in terms

Table 3.6: Relative improvement in precision and recall for single components for 7 libraries.

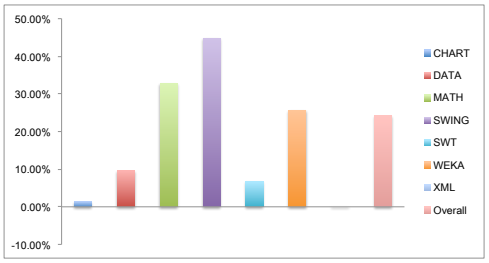
Component	Precision							Recall						
	CHART	DATA	MATH	SWING	SWT	WEKA	XML	CHART	DATA	MATH	SWING	SWT	WEKA	XML
Type Comparison (TC)	3%	0%	0%	-9%	12%	0%	34%	9%	0%	0%	5%	40%	0%	42%
Object Instance Creation (OIC)	2%	-1%	0%	-9%	10%	5%	34%	17%	10%	0%	5%	30%	7%	42%
Null Dereference (ND)	3%	1%	0%	-9%	7%	5%	29%	9%	2%	0%	5%	20%	7%	37%
Short Circuit Evaluation (SCE)	3%	1%	16%	9%	12%	5%	5%	3%	1%	27%	9%	25%	25%	5%
Count-Controlled Loop (CCL)	2%	10%	32%	45%	7%	0%	26%	2%	71%	64%	111%	20%	43%	0%
Local Exception (LE)	2%	2%	7%	0%	23%	2%	11%	0%	0%	0%	0%	0%	0%	0%

of precision and recall. In this section, we analyze the effectiveness of each component individually. Table 3.6 shows the improvements of each component in precision and recall, respectively. In general, every component helps improve the recall of the usage-based approach. The improvement could be as high as 110% with count-controlled loop on SWING. Most of the times, every component also helps improve the precision except for the 4 cases of three components on SWING and OIC component on DATA. Over the six libraries, XML and SWT benefit the most in both precision and recall. Now, let us present detailed results of two components CCL and OIC which are representative for two trends: improving in both precision and recall, and improving in recall with some decrease in precision, respectively. Detailed results for other components and analysis is present in Appendix for interested readers.

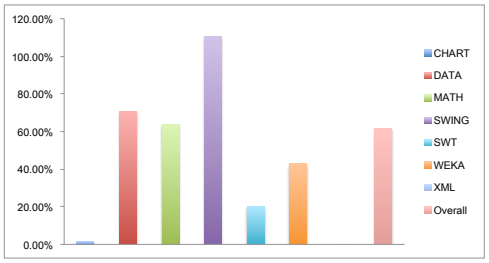
3.4.1 Count Controlled Loop (CCL)

This result shows the trend that we observed by some of the components when our approach gains in both precision and recall. This component observes the loop variant to infer the loop invariant and propagate this belief as a precondition. The result in Figure 3.5 shows that in terms of the precision, our approach achieves 2%–45% relative improvement and in terms of recall 2%–111% relative increase for CHART, DATA, MATH, SWING, SWT, WEKA libraries. The count-controlled loop component deduces implicit beliefs related to the upper or lower bounds of the loop. This helps to infer loop invariant preconditions and increases the recall as seen in the result. Our approach also achieves the improvement in terms of precision because the approach only mines true positive conditions and did not introduce new false positives. The result on XML was not improved compared to the base approach, as the APIs of interest for this library were not called inside a count-controlled loop in our dataset. We

have observed that most APIs from this library were called inside the condition-controlled loop if the loop is involved, which is out of context for this component.



(a) Relative improvement in precision.

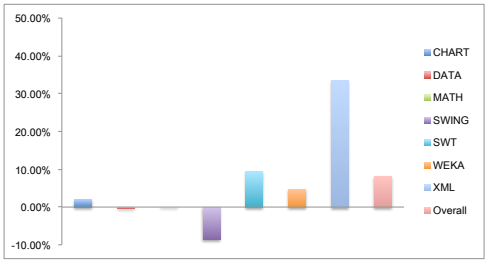


(b) Relative improvement in recall.

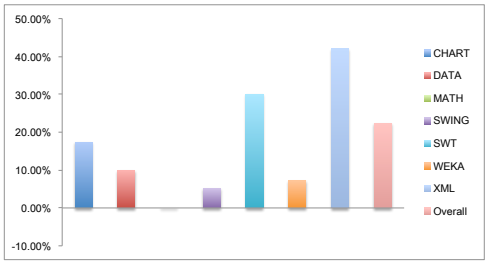
Figure 3.5: Accuracy of Count Controlled Loop (CCL).

3.4.2 Object Instance Creation (OIC)

Once an object is instantiated with a constructor, it becomes a `non-null` entity. These type of implicit belief generates `non-null` related preconditions. The expectation is to get more preconditions indicating that a component of the API could be `non-null`. Figure 3.6a shows that precision is increased by 2%–34% whereas Figure 3.6b shows recall increased by 7%–42% for CHART, SWT, WEKA and XML libraries. The addition of more true positive preconditions gives better recall and it helps increase the precision as well. In the cases of the libraries DATA and SWING, we see a decrease in precision by 1% and 9% respectively. The reason behind this is the mining of false positive conditions related to some APIs. We have examined such false positives and came to the conclusion that these false positives are stronger conditions than necessary, e.g., the API `Document.insertString(int, String, AttributeSet)` from the `text` package of SWING library permits insertion of a `null` string. However, from a programmer's point of view, it is logical to pass a `non-null` string to write into a document. In terms of recall, for both libraries DATA and SWING, our approach has an increase of 10% and 5%, respectively. In the case of the MATH library, we do not see any improvement for this component. This is because the APIs from this library do not have any `null`-related preconditions.



(a) Relative improvement in precision.

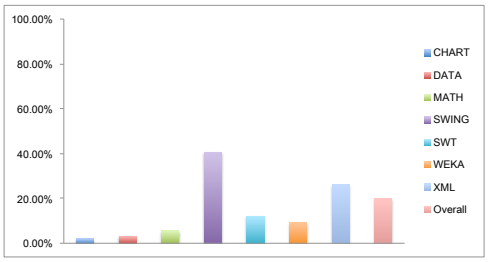


(b) Relative improvement in recall.

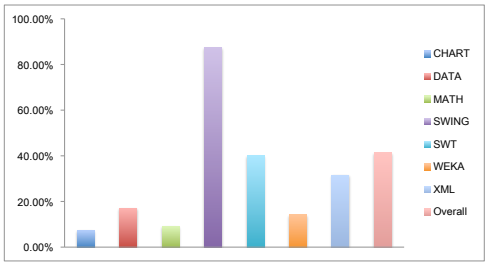
Figure 3.6: Accuracy of Object Instance Creation (OIC).

3.4.3 1-Level Control Flow Analysis (1-CFA)

If a parameter of a procedure is directly passed to an API, the existing mining approach cannot the mine precondition for that API due to the absence of the calling context. Performing single level context sensitive analysis to infer preconditions for such cases can benefit in mining preconditions. This component takes advantage of both implicit-belief-related conditions and explicit conditions from the caller methods if such conditions are within context. For all libraries of interest, our approach is able to improve mining accuracy in terms of both precision and recall. The detailed result is shown in Figure 3.7.



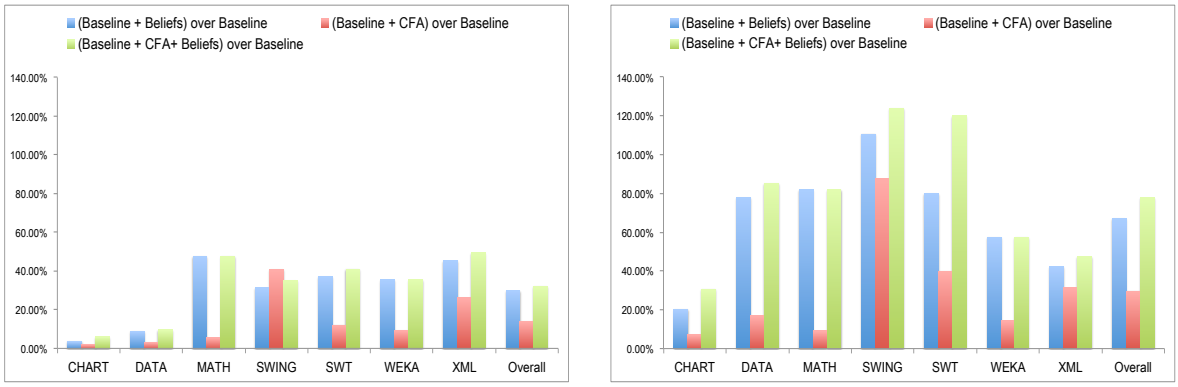
(a) Relative improvement in precision.



(b) Relative improvement in recall.

Figure 3.7: Accuracy of 1-Level Control Flow Analysis (1-CFA).

We have also studied whether the CFA component can retrofit with the base approach and if yes to what extent. We built the models under study including `(Baseline + Beliefs)`, `(Baseline + CFA)`, `(Baseline + CFA + Beliefs)`, and ran them on all libraries to compare the accuracies against the base approach. Figure 3.8 shows the result of this comparison. The base approach combined with implicit beliefs can increase precision by 4%–30% and recall by 20%–67%. Afterwards, in place of implicit-belief-related addition, we ran the experiment only with the CFA component on top of the



(a) Comparison in precision.

(b) Comparison in recall.

Figure 3.8: Comparison of 1-Level Control Flow Analysis (1-CFA), Beliefs with respect to baseline approach in terms of precision and recall for 7 different libraries of interest.

base model to demonstrate the improvement achieved by only this component. The retrofitted approach achieves improvement by 2%–14% in precision, and by 7%–30% in recall. It is evident from these results that CFA helps compared to only using the baseline approach, but the gain in accuracy is lesser than what we achieve when only implicit belief is used. Finally, we ran an experiment with the addition of both implicit belief components and CFA to differentiate the gains that our approach can achieve with both components. In this case, precision is increased by 6%–32% and recall by 30%–78%. This helps us conclude the fact that using both types of components further increases the accuracy. However, the preconditions mined by these two different types of components incorporate some overlapping preconditions.

3.5 Threats to Validity

The chosen dataset and libraries could not be representative. We mitigated this by using a large dataset of 14,785 projects which were carefully selected and widely used by previous work and selecting seven libraries from different domains. To verify the accuracy of the mined preconditions, we have manually built the ground-truth, which is prone to human error. We have implemented and shown the effects of each single component related to an implicit belief can have and how we can exploit these implicit beliefs to mine preconditions from code corpus. These single components that we have chosen might not be standard for all implicit beliefs discussed in our detailed classification. We have identified

the missing classes of preconditions and incorrect cases by studying the call sites of our used source code corpus.

CHAPTER 4. RELATED WORK

Our work is most related to the error inferring approach by Engler *et al.* [11]. The authors introduce a technique to collect sets of programmers' beliefs that are used to check for contradictions in source code. They define *beliefs* as the facts implied in the source code. They also classify beliefs into two types. MUST beliefs are directly implied in the code and MAY beliefs are where code features can be mined to suggest a belief from programmers but "may instead be a coincidence". For example, a call to *m* followed by a call to *n* implies a belief that they must be paired, but it could be a coincidence. Their technique first mines the MAY beliefs in the source code and considers a behavior that deviates from the MAY beliefs as a potential bug. In comparison, in our work, our implicit beliefs belong to the type of MUST beliefs. However, they may not be directly implied from the code. That is, they might not be directly exposed in the guard conditions. In those cases, the preconditions do not occur frequently enough for effective mining. This is the key limitation of the approach by Nguyen *et al.* [27], where the authors mine the preconditions of the APIs as the MAY beliefs by examining the guard conditions in the client code of the libraries of interest. By analyzing the implicit beliefs, we could complement to the techniques such as Nguyen et al. [27], which derives the MAY beliefs in those cases.

There has been rich literature on specification mining and inference [15, 13, 26, 8, 42, 1, 35, 36, 34]. Regarding the context of the code considered during specification mining, the existing specification mining approaches are broadly grouped into two kinds of approaches: usage-based approaches and implementation-based approaches.

Typical examples of usage-based approaches include [33, 27, 16, 28, 38, 22, 41, 24]. Ramanathan *et al.* [33] analyze call sites of a method and use path-sensitive, inter-procedural program analysis to mine predicates from these points. They infer preconditions by collecting the sets of predicates along each distinct path to the call sites. Then, the predicate sets at the points where the paths merge to

capture both control- and data-flow information are intersected. Preconditions are derived from frequent itemset mining on the data-flow results and sub-sequence mining on the control-flow results. Other approaches rely on data mining techniques while using more light-weight static analysis. Gruska *et al.* [16] mines temporal properties regarding pairs of called methods using a notion of consensus from 6k Linux projects. GrouMiner [28], JADET [38], Dynamine [22], Williams and Hollingsworth [41], and CodeWeb [24] also mine patterns of pairs of method invocations and graphs of API elements.

The second kind of approaches is implementation-based. They use either static or dynamic analysis on the implementation of the API of interest. Cousot *et al.* [6] use abstract interpretation to automatically infer preconditions. Buse *et al.* [4] use symbolic execution and inter-procedural dataflow analysis to automatically infer conditions leading to exceptions. There are also *dynamic* approaches to mining specifications [3, 7, 9, 12, 14, 21, 23, 31, 39, 43]. Most notable is Daikon [12] that detects program invariants by running test cases. Wei *et al.* [39] use programmer-specified contracts in code to infer more complex post-conditions. Weimer *et al.* [40] identify temporal safety rules by looking at exceptional control paths.

CHAPTER 5. CONCLUSION ANS FUTURE WORK

5.1 Conclusion

Although usage-based mining approach shows promising result to automatically mine specifications, the fact that it depends heavily on explicit guard conditions creates the sparse usage problem, if the call sites do not contain rich and frequent API usage. In this thesis, we have proposed usage of implicit beliefs to leverage such existing automated behavioral specification mining technique to resolve sparse usage problem. We have discussed capturing language construct and semantics related facts present in code to enable detection of implicit beliefs. We extract the relevant implicit beliefs as preconditions of an API in this approach. We have experimented using 2 JDK and 5 non-JDK leaf libraries that suffer from sparse usage problem in our dataset containing over 350 million lines of code and over 1 million API method calls from the chosen libraries. Compared to the results achieved by base usage-based mining approach, our approach has a relative raise in precision by 32% and in recall by 78% and reached a precision of 60% and recall of 69%.

5.2 Future Work

Currently, our data-driven approach mines atomic preconditions. We have observed that often these preconditions are connected by logical operators, that can be considered as richer/complex version of atomic preconditions. In future, we plan to investigate how effective usage-based and implicit belief related minings are in regard to inferring these richer set of preconditions. We can mine the code corpora to find the regularities present between the atomic precondition to achieve this purpose.
Another interesting direction to explore, could be creating a collection of specified code from applying our specification inference approach on quality code corpora. There have been examples of specified

code. For example, [10] that has focused on specified code in practice but not on the correctness of the specification, [25] that has focused on bug detection in implementation, but not on correctness of the specification, etc. These type of exploratory study can benefit, if a corpus of specified code is readily available.

BIBLIOGRAPHY

[1] F. Aleen and N. Clark. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, XIV, pages 241–252, New York, NY, USA, 2009. ACM.

[2] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Working Conference on Mining Software Repositories*, MSR'13, pages 207–216, May 2013.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16. ACM, 2002.

[4] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 273–282, New York, NY, USA, 2008. ACM.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[6] P. Cousot, R. Cousot, M. Fahndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *in Proceedings of the 14th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'13)*. Springer Verlag, January 2013.

[7] P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 150–168. Springer-Verlag, 2011.

[8] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 85–96, New York, NY, USA, 2010. ACM.

[9] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 528–550. Springer-Verlag, 2005.

[10] J. Dietrich, D. J. Pearce, K. Jezek, and P. Brada. Contracts in the Wild: A Study of Java Programs. In P. Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:29, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, Oct. 2001.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE'99, pages 213–224. ACM, 1999.

[13] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 364–374, New York, NY, USA, 2011. ACM.

[14] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 339–349. ACM, 2008.

[15] M. Gabel and Z. Su. Testing mined specifications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 4:1–4:11, New York, NY, USA, 2012. ACM.

[16] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 119–130. ACM, 2010.

[17] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[18] S. S. Khairunnesa, H. A. Nguyen, T. N. Nguyen, and H. Rajan. Exploiting implicit beliefs to resolve sparse usage problem in usage-based specification mining. *Proc. ACM Program. Lang.*, 1(OOPSLA):83:1–83:29, Oct. 2017.

[19] G. T. Leavens and C. Clifton. Lessons from the JML project. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Zurich, Switzerland*, volume 4171, pages 134–143, 2008.

[20] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32:705–778, August 1995.

[21] C. Liu, E. Ye, and D. J. Richardson. Software library usage pattern extraction using a software model checker. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 301–304. IEEE Computer Society, 2006.

[22] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 296–305, New York, NY, USA, 2005. ACM.

[23] D. Lo and S. Maoz. Mining hierarchical scenario-based specifications. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 359–370. IEEE Computer Society, 2009.

[24] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE'00, pages 167–176. ACM, 2000.

[25] V. Murali, S. Chaudhuri, and C. Jermaine. Bayesian specification learning for finding api usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 151–162, New York, NY, USA, 2017. ACM.

[26] A. C. Nguyen and S.-C. Khoo. Extracting significant specifications from mining through mutation testing. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering*, ICFEM'11, pages 472–488, Berlin, Heidelberg, 2011. Springer-Verlag.

[27] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 166–177, New York, NY, USA, 2014. ACM.

[28] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392. ACM, 2009.

[29] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[30] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[31] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382. IEEE Computer Society, 2009.

[32] H. Rajan, T. N. Nguyen, G. T. Leavens, and R. Dyer. Inferring behavioral specifications from large-scale repositories by leveraging collective intelligence. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 579–582, Piscataway, NJ, USA, 2015. IEEE Press.

[33] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 123–134. ACM, 2007.

[34] M. Renieris, S. Chan-Tin, and S. P. Reiss. Elided conditionals. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 52–57, New York, NY, USA, 2004. ACM.

[35] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, Nov. 1997.

[36] J. R. Ruthruff, S. Elbaum, and G. Rothermel. Experimental program analysis: A new program analysis paradigm. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 49–60, New York, NY, USA, 2006. ACM.

[37] O. G. Shivers. Control-flow analysis of higher-order languages of taming lambda. 1991.

[38] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44. ACM, 2007.

[39] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 191–200. ACM, 2011.

[40] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 461–476. Springer-Verlag, 2005.

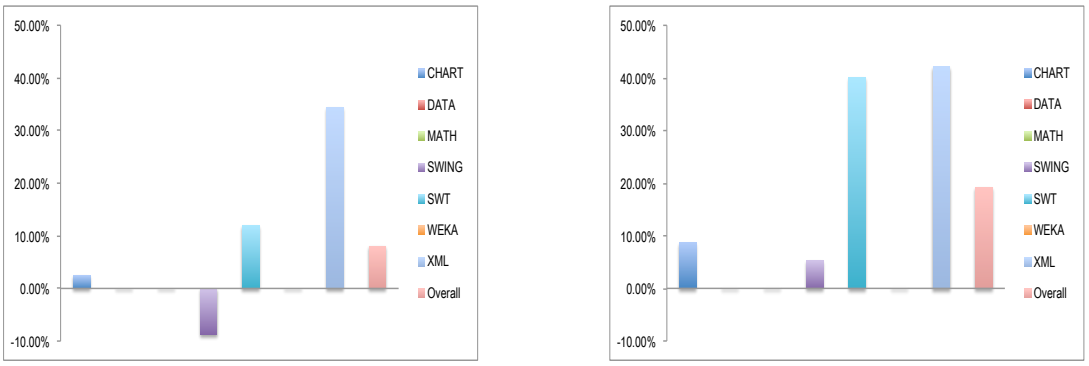
[41] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480, 2005.

[42] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *Formal Approaches to Software Testing: Third International Workshop on Formal Approaches to Testing of Software*, FATES '03, pages 60–69, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[43] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291. ACM, 2006.

APPENDIX. DETAILED RESULTS AND ANALYSIS OF SINGLE COMPONENTS

A Type Comparison (TC)



(a) Relative improvement in precision.

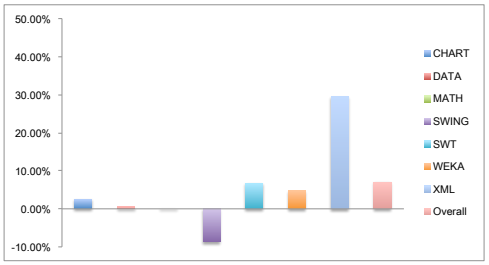
(b) Relative improvement in recall.

Figure A.1: Accuracy of Type Comparison (TC).

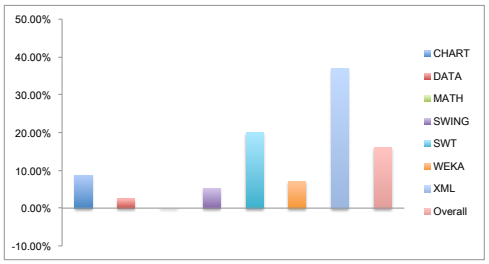
If an object is checked whether it is instance of some class and returns `true`, then that object is `non-null`. Using this implicit belief to mine more preconditions, we see a similar trend in terms of accuracy that we observed in object instance creation (OIC) component. For some libraries the precision is decreased due to mining stronger conditions than required. Recall improves for all libraries if the libraries themselves are expecting any `non-null` related precondition. The accuracy of this component is shown in Table 3.6 and the bar charts depicting the accuracy for this component are shown in Figure A.1.

B Null Dereference (ND)

If an object is dereferenced and then used by an API, that object must be `non-null`. The preconditions that stems from this implicit belief increased recall for the chosen libraries if the library anticipates `non-null` preconditions. In terms of precision we see a slight drop for two libraries as the



(a) Relative improvement in precision.



(b) Relative improvement in recall.

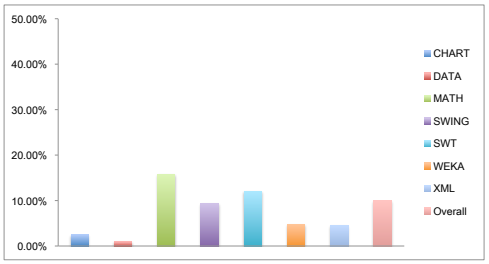
Figure B.1: Accuracy of Null Dereference (ND).

approach mines stronger `non-null` related preconditions. Again we see a similar trend as in other `non-null` implicit belief related component in the result in terms of accuracy. The accuracy for this component is reported in Table 3.6 and the bar charts of this component are present in Figure B.1.

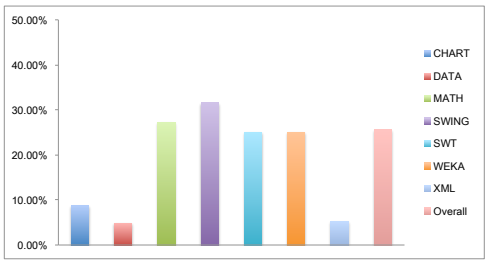
C Short Circuit Evaluation (SCE)

In short circuit evaluation, the evaluation of the second operands implies a specific value of the first operand.

Using this to mine preconditions for APIs that act as an operand in such expressions we achieved increase in precision and recall for all libraries. The result follows the trend of previous component and shown in the accuracy Table 3.6.



(a) Relative improvement in precision.



(b) Relative improvement in recall.

Figure C.1: Accuracy of Short Circuit Evaluation (SCE).

D Local Exception (LE)

If a client code throws a client-specific exception before calling an API, then the explicit guard condition is irrelevant to the API. This implicit belief can help in removing false positive conditions

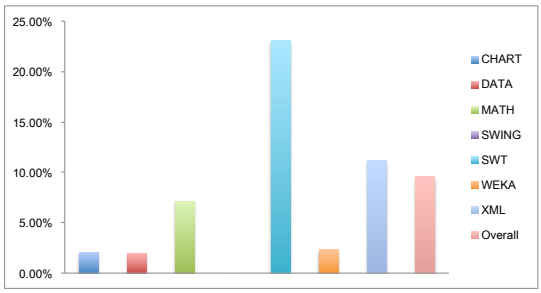


Figure D.1: Improvement in precision of Local Exception (LE).

from the existing mining based approach, thus can increase the precision. In Figure D.1, we see the relative peak in precision for 6 libraries by 2%–23%. SWING achieves the same result as the base approach, as our approach could not detect any client-specific exceptions from the usages. Inspecting the call sites, we have seen that although client-specific conditions were present in the code corpus we have used, those conditions do not guard any client specific exceptions. As a result, the approach was not able to detect such noise for SWING library. Table 3.6 shows that there is no increase in recall which is expected as this implicit belief aims to remove false positives only. The result also confirms that we have not removed any true condition, as there is no decrease in recall.