Accurate Extraction of Bug Fix Pattern Occurrences using Abstract Syntax Tree Analysis

Matias Martinez, Laurence Duchien, Martin Monperrus

▶ To cite this version:

HAL Id: hal-01075938
https://hal.archives-ouvertes.fr/hal-01075938

Submitted on 20 Oct 2014

Accurate Extraction of Bug Fix Pattern Occurrences using Abstract Syntax Tree Analysis

Matias Martinez   Laurence Duchien   Martin Monperrus

Abstract

This manuscript presents an approach for studying the abundance of bug fix patterns. Bug fix patterns capture the knowledge on how to fix bugs, they are essential building blocks for research areas such as bug fix recommendation and automatic repair. In this paper, we focus on the problem of the accurate measurement of bug fix pattern abundance: how to reliably tell that one pattern is more common than another one? We propose an approach to formalizing bug fix patterns and an accurate instance pattern identification process that uses this formalization. Our technique is based on a tree differencing algorithm working with abstract syntax trees (AST). A comparative evaluation shows that our approach improves the accuracy of pattern instance identification by an order of magnitude.

1   Introduction

Bug fix patterns capture the knowledge on how to fix bugs. Bug fix patterns are essential building blocks of research areas such as automatic program repair [1, 2]. One such bug fix pattern called *Change of If Condition Expression* (IF-CC) has been identified by Pan et al. [3]. Figure 1 presents one instance of this pattern by showing two consecutive revisions of a source code file. Revision N (on the left-hand side) contains a bug inside the *if* condition, a wrong call to the boolean method *isEmpty* instead of a call to the method *isFull*. In revision $N+1$ (the right-hand side piece of code) a developer fixed the bug by modifying the *if* condition expression.

The notion of abundance of bug fix patterns refers to whether some bug fix patterns are more important than others. A measure of abundance is the number of commits in which one observes an instance of the pattern, we call it the *abundance* of the pattern. The abundance reflects to what extent those bug fix patterns are used in practice.

For instance, Pan et al. report [3] that in the history of Lucene[1], the bug fix pattern "Change of if condition expression" (IF-CC) is the most common

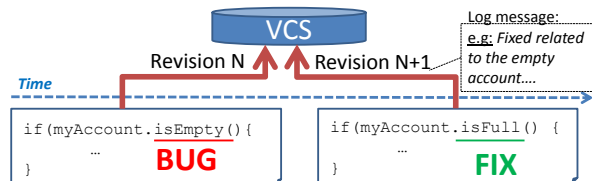---

[1]http://lucene.apache.org/core/

Figure 1: Example of a bug fix pattern called *Change of If Condition Expression (IF-CC)*[3], in two consecutive revisions of a source code file. The left-hand side revision contains a bug in the *if* condition: an incorrect method invocation. On the right-hand side, the developer fixed it by modifying the *if* condition, i.e. updating the method invocation.

pattern with 370 instances (accounting for 12% of all bug fix pattern instances identified). On the contrary, the pattern "Addition of operation in a operation sequence of field settings" (SQ-AFO) is the less abundant pattern, with only 5 instances being observed (0.2%).

The abundance of bug fix patterns is important for research on automatic software repair research [1, 2]. In our previous work [4], we have shown that knowing the abundance distribution of source code changes is key for repairing bugs faster. The contraposition holds, it takes longer to repair a bug with an inaccurate probability distribution than with no distribution at all. Since some automatic software repair approaches (e.g. [2]) use bug fix patterns to synthesize patches, one needs accurate abundance measures of bug fix patterns before embedding abundance in the repair process.

The *accuracy* of bug fix pattern instance identification refers to whether an approach yields the correct number of pattern instances. The threat to the accuracy of the abundance measurement of bug fix patterns is two-fold. First, one may over-estimate it by counting commits as instances of the pattern while there are actually not (false positives). Second, one may under estimate it by *not* counting commits, i.e. by missing instances (false negatives). *The challenge we address is to provide a mechanism to obtain an accurate measure of bug fix pattern abundance, by minimizing both the number of false positives and the number of false negatives.*

In this paper, we propose a new approach for specifying bug fix patterns and identifying pattern instances in commits. Our approach is based on the analysis of abstract syntax trees (AST) of revisions extracted from the software history (as recorded by version control systems). We measure and compare its accuracy against the one of Pan et al.'s pattern instance identifier, which is based on lines and tokens. This evaluation shows that our approach is more accurate: it identifies more correct bug fix pattern instances (78 vs 62), reduces the number of false positives (0 vs 74) and reduces the number of false negatives (11 vs 27). We explain why Pan et al.'s approach is sometimes inaccurate and discuss the limitations of our own approach.

To sum up, this paper makes the following contributions:

- A declarative approach for specifying bug fix patterns;

- An algorithm for accurately identifying pattern instances based on the analysis of abstract syntax trees;

- A comparative and manual evaluation of the accuracy of bug fix pattern instance identification of our AST-based approach against Pan et al.'s token-based approach;

- A formalization of 26 bug fix patterns using our AST-based specification approach, incl. 6 new patterns complementing those of the literature.

The remainder of this paper is as follows. Section 2 presents an approach to analyze software versioning history at the abstract syntax tree level. Section 4 presents the evaluation of the approach. Sections 5 and 6 respectively discuss the limitations of our approach and the related work. Sections 7 concludes the paper.

2 A Novel Representation of Bug Fix Patterns based on AST changes.

In the literature, there are catalogs of bug fix patterns [3]. For example, Pan et al. presented a catalog of 27 bug fix patterns. They describe each bug fix pattern with a brief textual description and one listing that shows the changes corresponding to the pattern's instance (at the source code line level). For example, the pattern *Change of If Condition Expression (IF-CC)* from Pan et al. is described as follows:

Description: "This bug fix change fixes the bug by changing the condition expression of an *if condition*. The previous code has a bug in the if condition logic."

Listing 1: Pattern *Change of If Condition Expression* defined by Pan et al.

```
− if (getView().countSelected() == 0)
+ if (getView().countSelected() <= 1)
```

Then, they measure the abundance of each bug fix pattern by mining bug fix pattern instances from commits of version control systems. For that, the authors use a tool called SEP to automatically identify pattern instances. We observe that twos definitions co-exist for the same pattern: the natural language one and the one that is encoded in the tool.

Our motivation is to introduce a new mechanism to formalize bug fix patterns declaratively instead of having them hard-wired in tools. In particular, we aim this formalization be both: *a*) human comprehensible; and *b*) used as input of mining algorithms that search pattern instances.

In this section, we present a methodology to formalize bug fix patterns. The method is based on AST analysis and tree differencing. In subsection 2.1, we introduce the terminology used in this paper. Them, in subsection 2.2, we

present a formalization of source code changes at the AST level. Finally, in subsection 2.3, we present a formalization of bug fix patterns at the AST level.

2.1 Terminology

A Version Control System (VCS) records the history of software changes during the development and maintenance of a software system. *A bug fix change* is an actual change on source code to fix a bug. *A bug fix pattern* is a type of change done to fix a bug. *A bug fix pattern identifier* classifies a concrete bug-fix change as an instance of a bug-fix pattern. *A revision* is a set of source code changes done over one file and introduced in the SCM. The revision produces a new *version* of the modified field. *A bug fix commit* is a commit containing changes to fix buggy code. *A hunk* is a set of co-localized changes in a source file. At the level of lines, a hunk is composed of a consecutive sequence of added, removed and modified lines. *A hunk pair* is a pair of related hunks, one hunk being a section of code at version n and the other being the corresponding code in the fixed version $n + 1$. Hunks are paired through the process of differencing that computes them.

2.2 Representing Versioning Changes at the AST Level

Our method identifies bug fix pattern instances from version control system revisions. It works at the abstract syntax tree (AST) level. This means we represent a source code file revision with one AST. The advantage of this representation is it allows us to extract fine-grained changes between two consecutive revisions by applying an AST differencing algorithm. This involves representing source code revisions as changes at the AST level. We use ChangeDistiller as AST differencing algorithm.

Figure 2: A lined-based difference of two consecutive revisions. The bug hunk in revision N (the left one) contains an "else" branch. The fix hunk in revision N+1 is empty. The corresponding AST hunk (introduced in section 3.1) consists of two nodes removal i.e. the 'else' node and the method invocation.

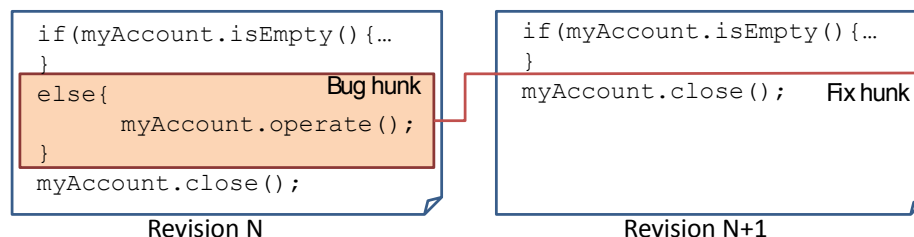Let us take as example the change presented in Figure 2. It shows a lined-based difference (syntactic) of two consecutive revisions. The bug hunk in revision N (the left one) contains an "else" branch. The fix hunk in revision N+1 is empty. The change consists of a removal of code: removal of "else" branch. At

4

the AST level, the AST differencing algorithm finds two AST changes: one representing the removal of an *else* node and another for the removal of a method invocation (i.e. myAccount.operate()) node surrounded by the *else* block.

ChangeDistiller handles a set of 41 source change types included in an object-oriented change taxonomy defined by Fluri and Gall [5]. For example, the taxonomy includes source code change types such as "Statement Insertion" or "Condition Expression Change". A code change type affects object-oriented elements such as "field addition". These elements are represented by 142 entity types.

Formally, ChangeDistiller produces a list of "AST source code changes". We formalize each change (scc) in a 7-value tuple:

$$scc = (ct, et, id\_e, pt, id\_p, opt, id\_op)$$

where ct is one of the 41 change types, et (for entity type) refers to the source code entity type related to the change. For example, a statement update may change a method call or an assignment. The field id_e is the identifier of the mentioned entity. As ChangeDistiller is an AST differencing, that field corresponds to the identifier of the AST node affected by the change. The field pt (for parent entity type) indicates the parent code entity type where the change takes place. For example, it corresponds to a top-level method body or to an "If" block. id_p is the identifier of the parent entity. For change type "Statement Parent Change", which represents source code movement, pt points to the new parent element. Moreover, opt and id_op indicate the parent entity type and the identifier for the old parent entity. Both fields specify the place the moved code was located before the change occurs, and they are omitted in tuples related to changes types different from "Statement Parent Change".

Let us present two examples of AST source code changes representation. As first example, a removal of an assignment statement located inside a "For" block is represented as: $scc1 =$("Statement delete" (ct), "Assignment" (et), node_id_23 (id_e), "For" (pt), node_id_14 (id_pt)).

As a second example, a movement of an assignment located in a method body to inside an existing "Try" block located in the same method is represented as: $scc2 =$("Statement Parent Change" (ct), "Assignment" (et), node_id_24 (id_e), "Try" (pt), node_id_15 (id_p), "Method" (opt), node_id_10 (id_op)). As this change is a movement i.e. "Statement Parent Change", its tuple includes the identifiers and type from the location the code comes from (*opt* and id_op, both ignored in $scc1$) and the new location as well (*pt* and id_p).

The structure contains information necessary to describe pattern's changes. In particular, it includes information of the parent entity type (and eventually the new parent entity type for movement of code) to describe the entity type where the changed entity is located. Moreover, the structure includes the identifiers (*ids*) of each entity involved in the change. This allows us to link the entities (AST nodes) affected by the change. For instance, let us consider a pattern that adds an *if precondition* just before a statement. An instance of this pattern has two changes: one that corresponds to the addition of the *if*;

the other a movement of the assignment (now the parent entity is the added *if* statement). The *ids* fields are used to validate whether the new location (parent entity) of the moved assignment is the added *if precondition*. In the following section we go deeper in the formalization of change patterns.

2.3 AST-based Pattern Formalization

In this section, we present a structure to formalize a bug fix pattern. This structure is used to identify bug fix pattern instances from the AST-level representation of revisions presented in section 2.2.

We specify a bug fix pattern with a structure composed by three elements: a list of micro-patterns L, a relation map R, and a list of undesired changes U.

$$\text{pattern} = \{L, R, U\}$$

In the following subsection we describe each of those pattern elements.

2.3.1 List of Micro-patterns

A micro-pattern represents a change pattern over a single AST node. It is an abstraction over ChangeDistiller's AST changes [6]).

A micro-pattern is a 5-value tuple

$$mp = (ct, et, pt, opt, cardinality)$$

where *ct, et, pt* and *opt*[2] have the same meaning as the source code change formalization in section 2.2. The *ct* field is the only mandatory, while fields *et*, *pt* and *opt* can take a *wildcard* character "*", meaning they can take any value. The field *cardinality* takes a natural number that indicates the number of consecutive equivalent (with the same value in fields *ct, et* and *pt*) AST changes it represents. It also can take the value *wildcard*, meaning that the micro-pattern can represent undefined number of consecutive equivalent changes. By default (absence of explicit value in *mp* tuple), the cardinality is value one. For example, a micro-pattern ("Statement Insert",*,*) means that an insertion of one statement of any type (e.g., assignment) inside any kind of source code entity, e.g. "Method" (top-level method statement) or "If" block. This micro-pattern is an abstraction of all AST source code changes corresponding to the addition of one AST node, whatever the node type and place in the AST.

The list of micro-patterns L represents the changes done by the pattern. The list is ordered according to their position inside the source code file. It is not commutative: a pattern formed by micro-pattern *mp*1 followed by *mp*2 is not equivalent to another formed by *mp*2 followed by *mp*1. The former means that *mp*1 occurs before *mp*2, while the latter means the opposite.

As example, let us present the AST representation of pattern "Addition of Precondition Check with Jump" [3]. This pattern represents the addition of an *if* statement that encloses a jump statement like *return*. It is represented by two

---

[2]We omit to specify *opt* in the tuple for addition, updates and removes operations.

micro-patterns[3]: $mp1 = $ ("Statement Insert", "If", *) and $mp2 = $ ("Statement Insert", "Return", "If").

2.3.2 Relation Map

The *relation map* R is a set of relations between entities involved in micro-patterns of L and U. Each relation links two entities (*et*, *pt* or *opt*) of two different micro-patterns. The relation r is written as:

$$r = mp1.entity_1 \text{ comp } mp2.entity_2$$

A relation formalizes a *link* between two elements (AST) from a pattern instance. That means, the elements of a pattern instance must fulfill all the relations from the pattern's relation map.

Each relation has three elements: two operands and one operator. The operator *comp* is used to compare the related entities. In particular, we use two operators: *equal* (==) and *not equal* (!=). For example, the relation written as $mp1.pt == mp2.pt$ uses the former operator, and relation as $mp1.pt != mp2.pt$ the latter.

The operands $entity_1$ and $entity_2$ specify which entity field from each micro-pattern (*et*, *pt* or *opt*) is involved in the relation. For instance, relation $mp1.pt == mp2.pt$ defines a relation between entity *pt* from micro-pattern $mp1$ and entity *pt* from micro-pattern $mp2$. This relation expresses that two changes affect entities with *the same* type of parent. Contrary, $mp1.pt != mp2.pt$ expresses that two changes affect entities with a *different* type of parent.

Another case of entity relation is expressed as $mp2.pt == mp1.et$. It defines that a change (matched with $mp1$) is done in an entity whose parent entity is affected by the second change (matched $mp2$).

As we mentioned, a pattern instance (i.e., a set of AST changes) must fulfill all the relations defined by the pattern. For example, let us consider a pattern P composed by micro-patterns $mp1$ and $mp2$ and one relation $R1 = (mp1.pt == mp2.et)$. Then, let us suppose that a set of changes composed by changes $scc1$ and $scc2$, are instances of $mp1$ and $mp2$, respectively. Changes $scc1$ and $scc2$ form an instance of P iff $R1$ is fulfilled by them. To verify whether those changes fulfill relation $R1$, we compare the *identifiers* of the entities affected by the changes. The first term of R1, i.e. $mp1.pt$, corresponds to the parent of $scc1$, i.e. $scc1.id\_p$. The second term of R1 ($mp2.et$) corresponds to the entity of $scc2$ ($scc.id\_e$). As consequence, the relation $R1$ is fulfilled when $scc1.id\_p == scc2.id\_e$.

2.3.3 Undesired Changes

Our bug fix pattern formalization also comprises a second list of micro-patterns. The list of *undesired changes* U represents micro-patterns that must not be present in the pattern instance. For example, the bug fix pattern "Removal of

---

[3]to simplify the example, we exclude jump statements 'break' and 'continue'.

an Else Branch" [3] requires only the "else" branch being removed, keeping its related "if" branch in the source code. In other word, the related "if" must not be removed.

As example, let us formalize this pattern. L contains one micro-pattern $mp1$ = ("Statement delete", "else","If"), U contains one *undesired change* u_mp1= ("Statement delete", "If",*) and R contains the relation $u\_mp1.et$!= $mp1.opt$. Generally, relations associated to micro-patterns from U have an operator "!=" and relate a micro-pattern of U with another from L. Hence, the relation restricts that no undesired change be related to changes associated to micro-patterns from L. In the example, the formalization of the pattern specifies that: *a)* there is a deletion of a "else" ($mp1$); *b)* it does not exist a deletion of an "if" entity that, in turn, is the parent entity of the deleted "else" (u_mp1)).

3 Defining the Importance of Bug Fix Patterns

The notion of abundance of bug fix patterns refers to whether some bug fix patterns are more important than others.

> The "abundance of a bug fix pattern" is the number of commits in which one observes an instance of the pattern.

The abundance reflects to what extent those bug fix patterns are used in practice. For instance, Pan et al. report [3] that in the history of Lucene[4], the bug fix pattern "Change of if condition expression" (IF-CC) is the most common pattern with 370 instances (12% of all bug fix pattern instances identified). On the contrary, the pattern "Addition of operation in an operation sequence of field settings" (SQ-AFO) is the less abundant pattern, with only 5 instances being observed (0.2%).

To measure the abundance of one bug fix pattern we need to *identify* instances of that pattern. The *accuracy* of bug fix pattern instance identification refers to whether an approach yields the correct number of pattern instances. The threat to the accuracy of the abundance measurement of bug fix patterns is two-fold. First, one may over-estimate it by counting commits as instances of the pattern while they are actually not (false positives). Second, one may under estimate it by *not* counting commits, i.e. by missing instances (false negatives). The challenge we address is to provide a mechanism to obtain an accurate measure of bug fix pattern abundance, by minimizing both the number of false positives and the number of false negatives.

Before to present an accurate pattern instance identifier in Section 3.2, we present the notion of AST hunk.

---

[4]http://lucene.apache.org/core/

8

3.1 Defining "Hunk" at the AST level

Previous work has set up the "localized change assumption" [3]. This states that the pattern instances lie in the same source file and even within a single hunk i.e., within a sequence of consecutive changed lines. For example, Figure 2 shows an example of two consecutive revisions of a Java file and a hunk pair representing the changes between the two revisions. The differences between the two files are grouped in consecutive changed lines which are called "hunk". In Pan et al.'s work [3], the authors identify pattern instances that necessarily belongs to only one hunk pair and, by transition, to one revision file.

From our experience, the "localized change assumption" is relevant in the process of identification of bug fix pattern instances. Since we work at the level of AST and hunk are the level lines (syntactic level), we define the notion of "hunk" at the AST level. AST hunks are *co-localized source code changes*, i.e., changes that are near one from another inside the source code.

The notion of hunk is important for searching pattern instances. Our identifier aims at identifying pattern instances from AST source code changes that are in the same hunk. That means, a pattern instance never contains AST instances from different AST hunks.

> For us, an AST hunk is composed of those AST changes that meet one of the following conditions: *a*) they refer to the same syntactic line-based hunk; or *b*) they are moves within the same parent element.

For instance, the two AST changes from the example of Figure 2 are in the same AST hunk (both changes occur in the same syntactic hunk). By construction, there is no AST hunk for changes related to comments or formatting, while, at the syntactic, line based level, those hunks show up.

3.2 An Novel Algorithm to Identify Instances of Commit Patterns from Versioning Transactions

This section presents an algorithm to identify bug fix pattern instances inside an AST hunk (see Section 3.1). The pattern instance identifier algorithm is composed of three consecutive phases: *a*) change mapping (Section 3.2.1); *b*) exclusion of AST hunks containing undesired changes (Section 3.2.2); and *c*) identification of change relations (Section 3.2.3). Let us explain each phase in the remain of the section.

3.2.1 Mapping Phase

The pattern instance identification algorithm first processes the phase named *Mapping phase*. The goal of the phase is to map each micro-pattern mp_j of L (list of micro-patterns, see section 2.3) with one AST change scc_i of the hunk. The output of the phase is a map of micro-patterns and AST changes. The

result of the mapping phase is successful if all micro-patterns of the bug fix pattern appear in the AST hunk i.e., they have at least one mapping with AST changes of the hunk. If this condition is not satisfied, the outcome phase is a *fail*, stopping the execution of the following phases. In other words, a pattern instance could not be identified in the hunk.

The mapping algorithm is explained in Section 10. Before, in Section 3.2.1 we detail the algorithm to match AST changes with micro patterns.

Algorithm 1: Algorithm to verify the matching between a micro-pattern and an AST change

Input: micro_pattern   ▷ Micro-pattern
Input: change   ▷ AST change (scc)
Output: boolean value: true if the AST change change matches with the micro-pattern micro_pattern

1 begin
  /* First, comparison of change types   */
2   **if** micro_pattern.ct == change.ct **then**
    /* Then, comparison of entity types   */
3   **if** micro_pattern.et != "*" *and* micro_pattern.et != change.et **then**
4   return $false$;
    /* Finally, comparison of parent entity types   */
5   **if** micro_pattern.pt != "*" *and* micro_pattern.pt != change.pt **then**
6   return $false$;
7   **else**
8   return $true$;
9   **else**
10   return $false$;

Mapping Creation Criterion   A change *scc* is mapped to the micro-pattern *mp* if *scc* is an instance of the change described by the *mp*. This relation is verified by matching the structures *scc* and *mp*. Algorithm 1 shows the matching algorithm pseudo-code. Both *match* (the matching is true) if their change types (line 2), entity types (line 3) and parent types (line 5) are the same. Note that if one wildcard (see Section 2.3) is specified, the field comparison is ignored (lines 3 and 5).

Mapping Algorithm Overview   Let us first explain the mapping procedure and then we detail the algorithm step by step in Section 28.

First, the mapping algorithm tries to find a mapping between list of micro patterns and a sequence of AST changes of the hunk. The algorithm first searches all possible beginnings of the pattern in the hunk. A beginning is an

AST change from the hunk and candidate to be the first element of the pattern inside the hunk. In other words, it must match with the first micro pattern.

Then, the algorithm tries to search a pattern instance from each of those beginnings. For each beginning, it proceeds to map the changes that follow the beginning with the list of micro-patterns. It iterates both the sequence of changes and the list of micro-patterns at the same time. A matching between a change and a micro-pattern is done in each iteration (as we explain in Section 3.2.1). If both match, the algorithm continues with the iteration, otherwise it stops analyzing the sequence and continues with the following beginning. Once the algorithm maps all micro-patterns with a sequence of changes, it returns that mapping. This sequence of AST changes is candidate to be a pattern instance.

It is important to note that the mapping phase defines two restrictions for the mappings between AST changes in a hunk and the micro-patterns. We call the first restriction *mapping total order*. It defines that the mapped AST changes must satisfy the order defined by L. Let us consider the list of micro patterns $L= \{mp1, mp2\}$ and an AST hunk $H = \{scc1, scc2\}$. The mapping $scc1$ with $mp1$ and $scc2$ with $mp2$ is valid. Let us explain why. The mapped AST changes respect the order imposed by the pattern i.e., through L, the first AST element of the hunk mapped with the first micro pattern, and so on. However, the mapping $scc1$ with $mp2$ and $scc2$ with $mp1$ is not valid. As $mp1$ appears before $mp2$ in L, then $scc2$ (mapped to $mp1$) must appear before $scc1$ in the hunk, and this is not the case. As consequence, this last mapping is not valid.

We call *consecutive mapping* to the second restrictions. It defines the mapped AST changes must be consecutive inside the hunk. In other words, it cannot exist one no-mapped change between two mapped changes. For instance, given a pattern formalization with 2 micro-patterns $mp1$ and $mp2$, and an AST hunk composed by 3 AST changes $scc1$, $scc2$ and $scc3$. Then, the mapping $mp1$, $scc1$ and $mp2$, $scc3$ is invalid due $scc2$ is not mapped and it is located between $scc1$ and $scc3$, both mapped changes.

Mapping Algorithm Pseudo-code Now, let us analyze the algorithm in detail. Algorithm 2 shows the pseudo-code of this mapping phase. The input of the algorithm is the list of micro-patterns L that represents the pattern, and a list of changes $Changes$ that represents one AST hunk.

The algorithm starts by searching a list *initial_changes* of AST changes. The list contains "candidates beginning" of the pattern inside the hunk i.e., in list $Changes$ (line 3). Each change of *initial_changes matches* with the first micro-pattern (see Algorithm 1 and explanation in Section 10).

Then, for each AST change *initial* of the mentioned list *initial_changes*, the algorithm tries to map all micro-patterns of L with the sequence S of consecutive AST changes that follow *initial*. The algorithm defines two cursors *change_i* and *micro_pattern_i* to iterate the sequence S and L, respectively. In each iteration (line 6), the algorithm matches the head of both cursors (line 12)

using Algorithm 1. If both match, the algorithm maps them and saves the association (line 13). After that, the cursors are updated (line 14 to 18 and from 22 to 23). The micro-pattern cursor is only updated once the algorithm has analyzed as many AST changes as the micro-pattern's cardinality indicates (line 16). When the cardinality is "$*$" (wildcard), the cursor $micro\_pattern\_i$ is updated (line 22) if at least one change from S is mapped to the current micro pattern (line 20).

The algorithm finishes successfully when all micro-patterns are mapped to consecutive AST changes (line 23 and 24).

3.2.2   Undesired Changes Validation Phase

The second phase verifies that no change of the *undesired changes U* list is present in the hunk. The algorithm of this phase maps changes for U with AST changes from the hunk. So it is similar to that one corresponding to the *Mapping phase.*

As opposed to the previous phase, an empty set of mappings is a good signal: no undesired change is present in the hunk. Contrary, in case that the micro-patterns of U are mapped to changes of the hunk, the relations over them must be fulfilled by the phase defined in section 3.2.3.

3.2.3   Relation Validation Phase

The change relation validation phase verifies that the relations defined by the pattern's *relation map* are satisfied by the mapped AST changes of the hunk. For the validation, the maps calculated in the two previous phases (3.2.1 and 3.2.2) are used.

Algorithm 3 shows the corresponding pseudo-code. First, for each relation the algorithm retrieves the two micro-patterns it relates (lines 3 and 4). Then, it retrieves the AST changes mapped to those micro-patterns (lines 5 and 6). After that, the algorithm retrieves the identifiers of the entities related to the changes. For that, function $getIdFromEntityType$ first determines which kind of entities (et, pt, or opt) the relation pinpoints. Then, it returns the identifier of the corresponding entity (lines 9 and 10). Finally, the two entity identifiers are compared (line 12) according to the operator defined by the relation (line 11). The comparison involves comparing *ids* of the entities i.e., AST nodes affected by the changes. As this phase is the last one from the AST change pattern identification, a successful validation of all relations means the presence of a pattern inside the analyzed hunk.

3.2.4   Identification Outcome

Once all phases were executed, the pattern instance identification algorithm determines the presence of a pattern instance inside the analyzed hunk if the following conditions are valid: *a*) all micro-patterns of L are mapped and the mapped AST changes from L fulfill relations of R; and *b*) no micro-pattern of U is mapped or every mapped AST change from U fulfill relation of R.

3.2.5 Conclusion

In this subsection, we have presented an algorithm to identify bug fix pattern instance in versioning transactions. This algorithm allows us to measure the abundance of bug fix patterns. Then, the abundance can be used in the automatic software repair field, for example, to define probabilistic repair models.

In the remain of the section we present two evaluations. In section 4.1, we evaluate the genericity of our bug fix pattern formalization approach. In section 4.2 we evaluate the accuracy of our approach with a manual analysis of a random sample of bug fix pattern instances found in commits of an open-source project.

4 Evaluation

In this section, we present a evaluation of our AST-based bug fix pattern identifier presented in Section 2. First, in section 4.1, we evaluate the genericity of our bug fix pattern formalization approach. Then, in section 4.2 we evaluate the accuracy of our approach with a manual analysis of a random sample of bug fix pattern instances found in 86 commits of an open-source project.

4.1 Evaluating the Genericity of the Pattern Specification Mechanism

In this section, we focus on the bug fix formalization we presented in section 2 and present an evaluation of its *genericity*. That means, we evaluate whether it is possible to specify known and meaningful bug fix patterns using our formalism.

> Can our specification format represent known and meaningful bug fix patterns?

To answer this research question, we first present bug fix patterns from the literature in section 4.1.1 and a set of new bug fix patterns in section 4.1.2. We eventually formalize these patterns in section 4.1.3.

4.1.1 Source #1: Bug Fix Patterns from the Literature

Pan et al. [3] have defined a catalog of 27 bug fix patterns divided in 9 categories. The categories are: If-related, Method Calls, Sequence, Loop, Assignment, Switch, Try, Method Declaration and Class Field. According to the number of citations, this is one of the most important papers on bug fix patterns.

Furthermore, we also consider three additional new bug fix patterns proposed by Nath et al. [7]. The patterns are named as: "method return value changes", "scope changes" and "string literals".

The existing definitions of bug fix patterns are written in natural language and are sometimes ambiguous. Before formalizing them, we have to clarify four

bug fix patterns from Pan et al. to facilitate their comprehension and formalization. We split four patterns whose definition mixes adding and removing code. For instance, "Add/removal of catch block" becomes "Add of catch block" and "Removal of catch block". Within the 27 original patterns, four of them were split, this results in a restructured catalog of 31 patterns $(27 + 4)$.

4.1.2 Source #2: New Bug Fix Patterns

In this section, we present new meaningful bug fix patterns that we found ourselves. We identified them while browsing many commits that were done to repair bugs [4].

Pattern DEC-RM: *Deletion of variable declaration statement* This bug fix pattern consists of the removal of a variable declaration inside the buggy method body (e.g. after a refactoring to transform a variable as field). This pattern is a sibling of Pan's patterns related to Class Field (i.e. Removal of a Class Field) but at method level.

Pattern THR-UP: *Update of Throw Statement* This bug fix pattern corresponds to the update of a *throw* statement. It includes changing the type of exception that is thrown, or modifying the exception's parameter.

Pattern MC-UP-CH: *Update of Method Invocation in Catch Blocks* This bug fix pattern consists in modifying the source code inside a catch body. This bug fix pattern hints that some bug fixes change the error handling code of *catch* blocks.

Pattern CONS-UP: *Update of Super Constructor Invocation* This bug fix pattern refers to the modification of *super* statement invocation, e.g., to change the parameter values. This bug related to incorrect calls to *super* were so far not discussed. "Super" is, according to our teaching experience, a hard concept of object-oriented design.

Pattern IF-MC-ADD: *Addition of Conditional Method Invocation* This bug fix pattern adds an *if* whose block contains one method invocation. This change could correspond to the addition of a guarded invocation, typically done in a bug fix to add missing logic in a limit cases. In Pan et al.'s catalog, there is a pattern "Addition of Precondition Check", that only adds the guard around an existing block. In contrast, our pattern also specifies the addition of both the precondition and the code of the "if block". Consequently, both patterns are related, they share the same motivation, but they are conceptually disjoint.

Pattern IF-AS-ADD: *Addition of Conditional Assignment* This bug fix pattern represents the case of adding an *if* statement and an assignment inside its block. It corresponds to a modification of a variable value under a specific condition defined by the *if*.

4.1.3 Results

In this section, we present a formalization of bug fix patterns, using the formalization presented in section 2.3. We formalize: *a*) 18 bug fix patterns from Pan

14

et al., belonging to the categories If, Loops, Try, Switch, Method Declaration and Assignment; *b*) 2 patterns proposed by Nath et al. [7]; *c*) 6 new bug fix pattern presented in section 4.1.2.

Table 1 shows the result of the formalization of those bug fix patterns. The table groups the formalization according the source of the patterns i.e., Pan, Nath, and the new bug fix patterns presented in section 4.1.2. Column *Name* shows the bug fix pattern identifier. The remaining three columns correspond to the formalization itself: *L (Micro-Patterns)* the list of micro-patterns, *U (Undesired Micro-Patterns)* the list of undesired changes and *R (Relational Map)* relations between micro-patterns.

The table presents bug fix patterns that are formalized by two or more sub-patterns. For example, pattern IF-APCJ (Addition of If PreCondition and Jump statement) is formalized by three sub-patterns. Each of these sub-patterns identifies pattern instances with a concrete jump statement. One corresponds to "break" jump statement, the other to "continue" jump statement and the last one to "return" statement.

The table also shows that the size of the micro-pattern list *L* varies between one and three. For those that *L* has two or three micro-patterns such as TY-ARTC, it exists a relation in *R* that defines a relation between micro-patterns (See Section 2.3.2). For those that *U* is not empty, a relation from *U* links a micro-pattern from *R* with another *U* (See Section 2.3.3).

In section 5 we discuss the limitations of our approach to formalize the remaining patterns from Pan et al. bug fix catalog.

4.1.4   Summary

In this section, we have shown that our approach is able to formalize 26 bug fix patterns. This answers our research question: our approach is flexible enough to formalize bug fix patterns from the literature and can also be used to specify new bug fix patterns.

15

Algorithm 2: Algorithm to map micro patterns to AST changes

Input: L   ▷ List of micro-patterns
Input: Changes   ▷ List of AST changes of a hunk
Output: boolean value: true if all micro-patterns of the pattern are
            mapped to AST changes of the hunk, false otherwise
Output: Mapping of Micro-Patterns and Changes from Changes

1 **begin**
```
   /* Retrieves the first micro-pattern                       */
```
2   micro_pattern_i ← getMicropattern(L,0) ;
```
   /* Search AST changes of the hunk that matches with
      micro_pattern_i                                         */
```
3   initial_changes ← getFirstMatchingChanges(Changes, micro_pattern_i) ;
4   **if** initial_changes *is null* **then**
5   return *false*, ∅

6   **foreach** *change* initial *of the list* initial_changes **do**
7   change_i ← initial; M ← ∅ ;
8   partialMapping ← *true*;
9   cardinality_iter ← 0 ;
10   **while** partialMapping *and* micro_pattern_i *is not null and* change_i
          is not null **do**
```
          /* cardinality receives a natural number or ∗
             (wildcard)                                       */
```
11   cardinality ← cardinality(micro_pattern_i);
```
          /* Comparison of AST change and micro-pattern       */
```
12   **if** *match(*micro_pattern_i, change_i*)* **then**
13   saveMapping(M, micro_pattern_i, change_i) ;
14   change_i ← getNextASTChange(Changes, change_i);
15   cardinality_iter ←cardinality_iter +1 ;
16   **if** cardinality *!= "∗" and* cardinality_iter *==* cardinality
                  then
17   micro_pattern_i ← getNextMicropattern(L,
                      micro_pattern_i);
18   cardinality_iter ← 0 ;
19   **else**
```
              /* if current micro_pattern_i could be mapped,
                 analyze next micro-pattern                   */
```
20   **if** cardinality *== "∗" and isMapped(* M, micro_pattern_i*);
21   **then**
22   micro_pattern_i ← getNextMicropattern(L,
                      micro_pattern_i);
23   cardinality_iter ← 0 ;
24   **else**
25   partialMapping ← *false*;

```
      /* Return true if all analyzed changes are mapped and
         all micro-patterns from L are mapped to changes from
         Changes                                              */
```
16
26   **if** partialMapping *and allMapped(* M, L, Changes*)* **then**
27   return *true*, M

28   return *false*, ∅

Algorithm 3: Algorithm to verify the relation between AST changes

Input: R   ▷ List of relations of a pattern
Input: M   ▷ Mapping of Micro-Patterns and Changes
Output: boolean value: true if the mapped AST changes respect the
             relations defined by the pattern, false otherwise

1 **begin**
2   **foreach** *relation* relation *of the list* R **do**
3   micro_pattern_1 ← getFirstMicropattern(relation);
4   micro_pattern_2 ← getSecondMicropattern(relation);
5   changes_1 ← getMappedChanges(micro_pattern_1, M);
6   changes_2 ← getMappedChanges(micro_pattern_2, M);
7   **foreach** *change* change_1 *of the list* changes_1 **do**
8   **foreach** *change* change_2 *of the list* changes_2 **do**
9   id_entity_1 ← getIdFromEntityType(relation.entity$_1$,
             change_1);
10   id_entity_2 ← getIdFromEntityType(relation.entity$_2$,
             change_2);
          /* operator receives values ''=='' or ''!=''   */
11   operator ← relation.comp;
          /* Applies the comparison operator operator to
           id_entity_1 and id_entity_2   */
12   comparison ← evaluate(id_entity_1, id_entity_2, operator) ;
          /* If the relation is not valid, the phase
           returns false   */
13   **if** comparison == *false* **then** return *false*;
14   ;

     /* All relations were valid   */
15   return *true*;

    /* Return an entity identifier according to the field (*et*, *pt*
     and *opt*) that a relation links   */
16 Function(*getIdFromEntityType(* relation, change*) : id*)
17 **begin**
18   **if** relation.*entity is a et field* **then**
19   return change.id_et;
20   **else**
21   **if** relation.*entity is a pt field* **then**
22   return change.id_pt;
23   **else**
24   return change.id_opt;

Name	L (Micro-Patterns)	U (Undesired Micro-Patterns)	R (Relational Map)	
New bug fix pattens				
DEC-RM	Delete of Variable declaration statement	mp1 = (Statement delete, Variable declaration, Method)		
THR-UP	Update of Throw statement	mp1 = (Statement update, Throw, Method)		
MC-UP-CH	Update of Method invocation in Catch clause	mp1 = (Statement update, Method invocation, Catch clause)		
CONS-UP	Update of Super constructor invocation	mp1 = (Statement update, Super constructor invocation, Method)		
IF-MC-ADD	Addition of precondition method invocation	mp1 = (Statement insert,If, Method) mp2 = (Statement insert, Method invocation, If)		mp2.pt == mp1.et
IF-AS-ADD	Addition of precondition assignment	mp1 = (Statement insert, If, Method) mp2 = (Statement insert, Assignment, If)		mp2.pt == mp1.et
Pan et al. bug fix pattens				
IF-APC	Addition of Precondition Check	$mp1$ = (Statement Insert, If,*) $mp2$ = (Statement Parent Change,*, If)		mp2.pt == mp1.et
IF-APCJ	Add Precondition Check with Jump (3 subcases)	$mp1$ = (Statement Insert, If,*) $mp2$ = (Statement Insert, Break, If) $mp1$ = (Statement Insert, If,*) $mp2$ = (Statement Insert, Continue, If) $mp1$ = (Statement Insert, If,*) $mp2$ = (Statement Insert, Return, If)		mp2.pt == mp1.et
IF-RMV	Removal of an If Predicate	$mp1$ = (Statement Delete, If,*) $mp2$ = (Statement Parent Change,*, If)		mp2.pt == mp1.pt
IF-ABR	Addition of an Else Branch	$mp1$ = (Alternative Part Insert, Else Statement,*)	u_mp1 = (Statement Insert, If,*)	u_mp1.pt != mp1.pt
IF-RBR	Removal of an Else Branch	$mp1$ = (Alternative Part Delete, Else Statement,*) $mp2$ = (Statement Parent Change,*, Else Statement)	u_mp1 = (Statement Delete, If,*)	u_mp1.opt != mp1.et
IF-CC	Change of If Condition Expression	$mp1$ = (Condition Expression Change, If,*)		
LP-CC	Change of Loop Predicate(3 subcases)	$mp1$ = (Condition Expression Change, While,*) $mp1$ = (Condition Expression Change, For,*) $mp1$ = (Condition Expression Change, Do,*)		

SW-ARSB	Addition of Switch Branch	$mp1$ = (Statement Insert, Switch Case,*)	u_mp1 = (Statement Insert, Switch Statement,*)	u_mp1.et != mp1.pt
SW-ARSB	Removal of Switch Branch	$mp1$ = (Statement Delete, Switch Case,*)	u_mp1 = ((Statement Delete, Switch Statement,*)	u_mp1.et != mp1.pt
TY-ARTC	Addition of Try Statement	$mp1$ = (Statement Insert, Try,*) $mp2$ = (Statement Parent Change , *, Try) $mp3$ = (Statement Insert, Catch Clause,*)		$mp1.et == mp2.pt$ and $mp1.et == mp3.pt$
TY-ARTC	Removal of Try Statement	$mp1$ = (Statement Delete, Try,*) $mp2$ = (Statement Parent Change, *, Try) $mp3$ = (Statement Delete, Catch Clause,*)		$mp1.et == mp2.pt$ and $mp1.et == mp3.pt$
TY-ARCB	Addition of a Catch Block	$mp1$ = (Statement Insert, Catch Clause,*)	u_mp1 = (Statement Insert, Try,*)	u_mp1.et != mp1.pt
TY-ARCB	Removal of a Catch Block	$mp1$ = (Statement Delete, Catch Clause,*)	u_mp1 = (Statement Delete, Try,*)	u_mp1.et != mp1.pt
MD-CHG	Change of Method Declaration (7 subcases)	$mp1$ = (Parameter Insert, Single Variable Declaration,*) $mp1$ = (Parameter Delete, Single Variable Declaration,*) $mp1$ = (Parameter Type Change, Simple Type,*) $mp1$ = (Parameter Type Change, Primitive Type,*) $mp1$ = (Parameter Ordering Change, Single Variable Declaration,*) $mp1$ = (Return Type Change, Simple Type,*) $mp1$ = (Return Type Change, Primitive Type,*)		
MD-ADD	Addition of a Method Declaration	$mp1$ = (Additional Functionality, Method,*)		
MD-RMV	Removal of a Method Declaration	$mp1$ = (Removed Functionality, Method,*)		
CF-ADD	Addition of a Class Field	$mp1$ = (Additional Object State, Attribute,*)		
CF-RMV	Removal of a Class Field	$mp1$ = (Removed Object State, Attribute,*)		
Nath et al. bug fix pattens				
Nath-1	Scope changes (2 subcases)	$mp1$ = (Statement Parent Change, *,*,*)	u_mp1 = (Statement Delete, *,*)	u_mp1.et != mp1.opt
		$mp1$ = (Statement Parent Change, *,*,*)	u_mp1 = (Statement Insert, *,*)	u_mp1.et != mp1.pt
Nath-3	Method return value changes	$mp1$ = (Statement Update, Return,*)		

Table 1: Formalization of bug fix patterns.

4.2 Evaluating the Accuracy of AST-based Pattern Instance Identifier

In this section, we present an experiment to measure the accuracy of our bug fix pattern instance identifier presented in section 3.2 . This evaluation is built on the following research question: *Is our AST-based pattern instance identifier more accurate than the state-of-the-art pattern identifier presented by Pan et al. [3]?*

The experiment consists of a manual inspection and validation of bug fix pattern instances identified in 86 commits of an open-source project. Given a bug fix commit and an instance of pattern P_i identified by an identifier, the instance is considered valid if the manual inspection validates that the change indeed corresponds to pattern P_i. Otherwise, the instance is considered invalid.

4.2.1 Evaluated Pattern Instance Identifiers

4.2.1.1 Baseline Classifier The baseline tool we selected is called SEP[5]. SEP is a token-based classifier used to identify bug fix instances from revisions of Java files (a revision is a pair of file, say Foo.java version 1.1 and Foo.java version 1.2). According to the code symbols, this tool was used to gather the results presented in Pan et al.'s study [3].

4.2.1.2 AST-based Classifier We develop a tool that implements the AST classifier presented in Section 3.2. The tool is implemented in Java and uses ChangeDistiller [5] to obtain AST-level differences between consecutive revisions of a file. We use a publicly available implementation of ChangeDistiller[6].

We limit both tools to identify instances of 18 bug fix patterns from Pan et al. bug fix catalog. These patterns are those we are able to represent using our pattern formalization presented in section 4.1.3.

4.2.2 Analyzed Data

We randomly selected a sample of 86 revisions (pairs of Java files) from the CVS history of the Lucene open-source project (from 09/2001 to 02/2006). The sampling strategy is that those revisions contain a small number of source code changes, less than 5 AST changes (this excludes formatting and documentation changes). Lucene is one of the six open-source software applications used in Pan et al.'s work. The dataset is available on [8].

4.2.3 Experimental Results

Table 2 shows the result of the manual inspection for pattern instances from Lucene's revisions identified by our AST-based approach and SEP tool. For each algorithm, the table shows the number of valid pattern instances, i.e. the

---

[5]http://gforge.soe.ucsc.edu/gf/project/sep/scmsvn/
[6]http://www.ifi.uzh.ch/seal/research/tools/changeDistiller.html

	Valid	Not Valid	Missing
Pan et al's Token-based Approach	62	74	27
Our AST-based Approach	78	0	11

Table 2: The Results of the Manual Inspection of Bug Fix Pattern Instances. The row "Token-Based" corresponds to the instances identified by the token-based classifier. The row "AST-Based" corresponds to the instances identified by the AST classifier.

true positives (column "Valid") and the number of invalid instances, i.e., the false positive instances (column "Not Valid"). Moreover, it shows a number of missing instances (false negatives) i.e. valid instances that an approach could identify but the other could not (column "Missing"). This number is not an absolute number of false negatives, it is only relative with respect to the approach. In the remaining of this section we study the accuracy of both approaches.

4.2.3.1 Accuracy Definition We define the accuracy of a bug fix pattern identifier as follows:

$$accuracy = \frac{\text{number of bug fix instance correctly identified}}{\text{total number of instance identified} + \text{missing pattern instance}}$$

For instance, a pattern identifier that identifies 5 instances, all correctly, but misses 2 instances, has an accuracy of $5/(5 + 2) = 0.71$. Another example is an identifier that correctly identifies 4 instances, incorrectly 1 and misses 2. Its accuracy is $4/(4 + 1 + 2) = 0.57$. According to the accuracy values, the first identifier is *more accurate* than the second one.

4.2.3.2 Accuracy of Token-based Identification The token-based identifier finds 136 instances of bug fix patterns in the 86 commits. Table 2 shows that our manual inspection found 62 valid pattern instances (true positives), 74 invalid (false negatives) and 27 missing instances. The accuracy of the token-based instance identifier is $62/(62 + 74 + 27) = 0.38$.

Let us analyze some cases where the identifier finds invalid instances. For instance, the token-based identifier identifies from revision 1.4 of class "FilteredQuery", an invalid instance of pattern "Change of Method Declaration" (MD-CHG) and another invalid instance of pattern "Addition of precondition with jump" (IF-APCJ). The actual bug fix pattern in this commit is "Addition of Method Declaration" (MD-ADD). The first invalid instance is due to a wrong mapping between the lines of the revision. The added method is "mapped" to an existing method (with different signature), resulting in the change being interpreted as an update of the method declaration. For the second false positive, the invalid pattern instance is identified inside the code of the added method.

We also found false positive instances caused by formatting changes between consecutive revisions. For example, the revision 1.3 of Lucene's class

GermanStemmer applies formatting changes in the source code and among the many modified lines, one local variable is initialized. The token-based identifier incorrectly identifies from this pair 21 instances of 9 different bug fix patterns. The formatting changes produce a complex mapping between the revision and its predecessor in many hunks. Consequently, the code inside these formatting hunks matching with a bug fix pattern definition is incorrectly identified as an instance.

4.2.3.3 Accuracy of AST-based Identification The AST-based identifier found 78 bug fix pattern instances. These instances were present in 53 different revisions. Moreover, the identifier could not identify 11 instances (missing). The accuracy of this identifier is $78/(78 + 11) = 0.88$.

Our manual inspection found that 100% (78/78) of the pattern instances were valid (the data is available). This implies all bug fix instances were *true positives*. These identified instances correspond to 9 different bug fix patterns. However, the algorithm also missed some instances, i.e., suffers from false negatives, which are discussed in Section 4.2.3.4.

4.2.3.4 False Negatives A false negative (or missing) instances is a valid bug fix pattern instance which is not identified by a pattern instance identifier. To detect those instances from the analyzed data, we cross the results obtained from both AST and token-based approaches. A missing instance of a pattern instance identifier A is not identified by A but is identified correctly by the other approach.

Table 2 presents the classification result. The token-based approach had 27 missing bug fix instances while the AST-based one had 11 false negatives.

Let us now analyze the false negatives of our approach. For 7 of 11 missing instances, the cause is due to the tree differencing tool (ChangeDistiller) we use to compute the differences between two consecutive revisions. ChangeDistiller does not compute changes inside anonymous and inner classes and there were 7 pattern instances in such classes in our data. For example, our algorithm does not identify an instance of pattern "Removal of if predicate (IF-RMV)" in revision 1.21 of class IndexSearcher, the instance is in the inner class HitCollector. Another case is that our approach does not see changes in the specification of thrown exceptions (keyword "throws" in Java), which are instances of pattern "Change of method declaration (MD-CHG)". For example, revision 1.4 of class TestTermVectorsWriter modifies the signature of the method by adding a clause "throws IOException". Our tree differencing algorithm does not consider those changes and this limitation impacts the accuracy of this particular bug fix pattern.

4.2.4 Conclusion

The manual analysis done in the presented experiment allows us to respond to our research question: our AST-based identifier is more accurate than the token-based used by Pan et al. in their experiments.

	#Commits	#Revisions	#Java Revisions
All	24,042	173,012	110,151
BFP	6,233	33,365	23,597

Table 3: Versioning data used in our experiment. Since we focus on bug fix patterns, we analyze the 23,597 Java revisions whose commit message contains "bug", "fix" or "patch".

The results of our experiment are summarized in Table 2. It shows that our AST-based identifier is able to identify: more valid bug fix instances (more true positives); less invalid instances (less false positives); less number of missing instances (less false negatives). Consequently, we can say that it is more accurate (0.88 vs. 0.38) than the token-based approach.

4.3   Learning the Abundance of Bug Fix Patterns

In this section, we use the bug fix pattern instance identifier presented in Section 3.2 to measure the abundance of bug fix patterns. The abundance allows us to measure the abundance of bug fix patterns. Then, one can define a probabilistic repair model formed of bug fix patterns and their frequencies [4].

This kind of probabilistic repair model could be used by bug fix pattern-based repair approaches such as PAR [2]. To create candidate fixes, PAR instantiates 10 bug fix templates, derived from bug fix patterns. Then, PAR navigates the search space in a uniform random way, that means, it takes randomly one bug fix template to be applied in a buggy location. The pattern abundance could be used in an extension of the strategy to navigate the search space. Instead of a random strategy, the extension could start navigating the space from the most abundant bug fix templates (i.e., the most frequent kind of fixes applied by developers) to the less abundant. This strategy could help to find a fix faster, avoiding applying infrequent changes in bug fixing.

4.3.1   Dataset

We have searched for instances of the 18 patterns mentioned in 4.1 in the history of six Java open source projects: ArgoUML, Lucene, MegaMek, Scarab, jEdit and Columba. In Table 3 we present the total number of commits (versioning transactions) and revisions (file pairs) present in the history of these projects. In the rest of this section, we analyze the 23,597 Java revisions whose commit message contains "bug", "fix" or "patch", in a case insensitive manner (row "BFP" in Table 3).

4.3.2   Empirical Results

Table 4 shows that our approach based on AST analysis scales to the 23,597 Java revisions from the history of 6 open source projects. This table enables

Pattern name	Abs
Change of If Condition Expression-IF-CC	4,444
Addition of a Method Declaration-MD-ADD	4,443
Addition of a Class Field-CF-ADD	2,427
Addition of an Else Branch-IF-ABR	2,053
Change of Method Declaration-MD-CHG	1,940
Removal of a Method Declaration-MD-RMV	1,762
Removal of a Class Field-CF-RMV	983
Addition of Precond. Check with Jump-IF-APCJ	667
Addition of a Catch Block-TY-ARCB	497
Addition of Precondition Check-IF-APC	431
Addition of Switch Branch-SW-ARSB	348
Removal of a Catch Block-TY-ARCB	343
Removal of an If Predicate-IF-RMV	283
Change of Loop Predicate-LP-CC	233
Removal of an Else Branch-IF-RBR	190
Removal of Switch Branch-SW-ARSB	146
Removal of Try Statement-TY-ARTC	26
Addition of Try Statement-TY-ARTC	18
Total	21,234

Table 4: The Abundance of Bug Fix Patterns: Absolute Number of Bug Fix Pattern Instances Found in 23,597 Java Revisions.

us to identify the abundance of each bug fix pattern. For instance, adding new methods (MD-ADD) and changing a condition expression (IF-CC) are the most frequent patterns while adding a try statement (TY-ARTC) is a low frequency action for fixing bugs. Overall, the distribution of the pattern instances is skewed, and it shows that some of Pan's patterns are really rare in practice. Interestingly, we have also computed the results on all revisions – with no filter on the commit message – and the distribution of patterns is rather similar. It seems that the bug-fix-patch heuristic does not yield a significantly different set of commits.

4.3.3 Summary

In this subsection, we presented the abundance of 18 bug fix patterns from the analysis of 6 open-source projects. We found that the most frequent changes to fix bugs are changes in *if condition* statements. Knowing this distribution is important in some contexts. For instance, from the viewpoint of automated software repair approaches: their fix generation algorithms can concentrate on likely bug fix patterns first in order to maximize the probability of success.

5 Discussion

5.1 Threats to Validity

Our results are completely computational and a severe bug in our implementation may invalidate our findings. During our experiments, we studied in details dozens of bug fix pattern instances (the actual code, the fix and the commit message) found by the tool and they were meaningful.

Another threat is the criterion to manually classify a bug fix pattern instance as valid or not. It could vary depending on who inspects it (an expert, a developer, a novice, etc.).

5.2 Limitations

In this section we sum up the limitations of our bug fix pattern formalization approach.

5.2.1 Context Dependence

We notice that some patterns only describe the nature of change itself, while others describe the change in a given context. By nature of change, we mean only the added and removed content; by context, we mean the code around the added and removed content. For instance, Pan et al. define a pattern representing the removal of a method call in a sequence of method calls. To us, this pattern is *context-dependent*. To observe an instance of removal of a method call in a sequence of method calls: 1) the change itself has to be a removal of a method call 2) the context of the removal has to be a sequence of method calls

on the same object. In total, there is a minority of 8/31 bug fix patterns of the refined catalog presented in section 4.1.1 that are context-dependent.

We do not consider those context-dependent bug fix patterns. This limitation could probably be overcome with a way to specify the "context" (the code surrounding the diff) at the AST level.

5.2.2 Limitations Inherited from ChangeDistiller

Another limitation of our pattern formalization approach is due to the change taxonomy used by tree differencing algorithm. ChangeDistiller misses some kinds of source code changes. For instance, an update operation in a class field declaration is not detected. This limitation prevents us to represent pattern "Change of Class Field Declaration" (CF-CHG) using AST changes as well as 4 other patterns. Those 5 patterns contain at least one change that is not covered by the change taxonomy of ChangeDistiller.

Another limitation is the granularity of the tree differencing algorithm. ChangeDistiller works at the statement level. This prevents us to study certain fine-grain patterns. For example, the addition of a new parameter or the change of an expression passed as parameter of a method call cannot be detected. Also, as we discussed in Section 4.2.3.4, the tree differencing algorithm does not detect changes inside anonymous classes. Improvement or replacement of the tree differencing algorithm could potentially decrease the number of false negatives.

6 Related Work

6.1 Bug Fix Patterns and Defect Classification

Pan et al. [3] present a catalog of 27 bug fix pattern and a tool to extract instances of them from source code. In this paper, we present an AST-based approach to identify bug fix patterns that, according to our evaluation, it is more accurate than the approach presented in Pan et. al.'s work.

Nath et al. [7] manually measure the abundance of Pan et al. patterns in CheckStyle java open-source project. They inspect by hand a set of bug fix changes to calculate this measure and also propose three alternative patterns discovered from this manual inspection. A difference with their work is that we automatically identify bug fix instances.

DeMillo and Mathur [9] present a syntactic fault classification scheme. They use this schema to classify the errors reported by Knuth [10]. The fault classification algorithm automatically classifies errors using a top-down strategy. As in our work, they rely on tree comparison to obtain the differences between them.

Chillarege et al. [11] present an Orthogonal Defect Classification (ODC). It corresponds to a categorization of defects into classes called *Defect Types*. This categorization has 8 defect types. Then, they study the relation between these classes and the different stages on the software development process. The defect classification is at a higher level compared to bug fix patterns at the AST level.

Duraes et al. [12] present a analysis of a collection of real software faults. They refine the aforementioned ODC, considering the language construct and program context surrounding the fault location. Contrary to our automatic approach, they manually analyze "diff" of several open source programs to classify the faults.

Ostrand and Weyuker's [13] present a scheme for categorizing software errors. They characterized an error in distinct areas, including "major category", "type", presence, and use of data. For example, "major category" identifies what type of code was changed to fix the error. They develop this classification schema from change reports filled out by developers of an industry software product. Then, they present the number and percentage of errors of each area. The main difference with our work is they do not define their schema from source code artifacts but from written reports of developers.

6.2 Source Code Analysis

Dyer et al.[14] present domain-specific language features for source code mining. Their language features are inspired by object-oriented visitors and provide a default depth first traversal strategy. They provide an implementation of these features in the Boa infrastructure [15] for software repository mining. This work targets to mine, at a fine-grained level, source code elements of one particular revision of a program. The authors present an motivation example of discovering a particular change pattern i.e. added a null check between consecutive revisions. They applied an *ad-hoc* strategy to get the changes between those revision. As the author mention, this strategy includes inaccuracy cases i.e. adding and removing same number of null checkers. As differences, our work is based and relies on a differencing algorithm to obtain fine-grained changes between revisions. This gives us more expressibility to describe patterns that affect to more complex patterns.

Martin et al. [16] present a language called PQL (Program Query Language) that allows programmers to express such questions easily in an application-specific context. The authors have developed both static and dynamic techniques to find solutions to PQL queries. In particular, the static analyzer finds all potential matches conservatively using a context-sensitive, flow-insensitive, inclusion-based pointer alias analysis. For example, they present as example the specification of a pattern to check for potential leaks of le handles. In contrast to this work, our work formalize *change patterns* i.e. bug fix changes. Our patterns describe a set of changes between consecutives source code revisions, while their work consist on match a given pattern with the source code within a source code file.

Andersen and Lawall [17] as well as Meng et al. [18, 19] worked on approaches to find generic patches (also referred to as "systematic edits"). For instance, Meng et al.'s tool – Lase – is given a set of file pairs and learns both a generic patch and a pointcut matching the places where the patch can be applied. In other words, Lase works with instances already identified (the file pairs) but not already formalized. On the contrary, our approach works with no instances

already identified but with the pattern already formalized. Our tool can then automatically identify pattern instances.

Instead of mining commits, Negara et al. [20] look for patterns in fine-grain changes collected in the development environment. Compared to the patterns discussed in this paper, they are of different nature, at a smaller scale.

6.3 Bug Classification

Kim et al. [21] present an approach to detect potential bugs and suggest corresponding fixes. They define BugMem, a database of preprocessed bug fix hunks. BugMem detects whether a new source code change corresponds to a known bug and suggests a possible fix given a particular change based in the stored information. In contrast with our work, they do not use explicit bug fix patterns in their process as we do.

Nagwani et Verma [22] present a bug classification algorithm, by clustering textual similarities of bug attributes. As difference with our work, they use software reports (e.g. Bugzilla), while we work at source code level.

6.4 AST Change Patterns

Fluri et al. [23] use hierarchical clustering of source code changes to discover change patterns. As in our work, they use ChangeDistiller to obtain fine-grained source code changes. They concentrate on coarse grain change patterns (such as development change, maintenance change), while we focus on fine-grain, AST level bug fix patterns only.

7 Conclusion

In this paper we have proposed a new approach for specifying bug fix patterns and identifying pattern instances in commits. Our approach is based on the analysis of abstract syntax trees (AST). We have performed a comparative evaluation of the accuracy of bug fix pattern instance identification of our AST-based approach against Pan et al.'s token-based approach. Our approach minimizes both the number of false positives and the number of false negatives: it identifies more correct bug fix pattern instances (78 vs 62), reduces the number of false positives (0 vs 74) and reduces the number of false negatives (11 vs 27). Moreover, we present 6 new additional bug fix patterns that complement those from Pan et al. bug fix pattern catalog [3].

References

[1] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 364–374, IEEE Computer Society, 2009.

[2] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 802–811, IEEE Press, 2013.

[3] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, pp. 286–315, June 2009.

[4] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Software Engineering*, pp. 1–30, 2013.

[5] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, (Washington, DC, USA), pp. 35–45, IEEE Computer Society, 2006.

[6] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, pp. 725 –743, nov. 2007.

[7] S. K. Nath, R. Merkel, and M. F. Lau, "On the improvement of a fault classification scheme with implications for white-box testing," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, (New York, NY, USA), pp. 1123–1130, ACM, 2012.

[8] M. Martinez, L. Duchien, and M. Monperrus, "Appendix of "Accurate Extraction of Bug Fix Pattern Instances using Abstract Syntax Tree Analysis"," tech. rep., INRIA, 2013. Available at `http://goo.gl/y4f1r`.

[9] R. A. Demillo and A. P. Mathur, "A grammar based fault classification scheme and its application to the classification of the errors of tex," tech. rep., Software Engineering Research Center, Purdue University, 1995.

[10] D. E. Knuth, "The errors of tex," *Softw., Pract. Exper.*, vol. 19, no. 7, pp. 607–685, 1989.

[11] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification - a concept for in-process measurements," *IEEE Trans. Software Eng.*, vol. 18, no. 11, pp. 943–956, 1992.

[12] J. Duraes and H. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, pp. 849 –867, nov. 2006.

[13] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment," *Journal of Systems and Software*, vol. 4, no. 4, pp. 289–300, 1984.

[14] R. Dyer, H. Rajan, and T. N. Nguyen, "Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes," in *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, (New York, NY, USA), pp. 23–32, ACM, 2013.

[15] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 422–431, IEEE Press, 2013.

[16] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: A program query language," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), pp. 365–383, ACM, 2005.

[17] J. Andersen and J. L. Lawall, "Generic patch inference," *Automated Software Engineering*, vol. 17, no. 2, pp. 119–148, 2010.

[18] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 329–342, ACM, 2011.

[19] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and Applying Systematic Edits by Learning from Examples," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 502–511, IEEE Press, 2013.

[20] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 803–813, 2014.

[21] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, (New York, NY, USA), pp. 35–45, ACM, 2006.

[22] N. Nagwani and S. Verma, "Clubas: An algorithm and java based tool for software bug classification using bug attributes similarities," *Journal of Software Engineering and Applications*, vol. 5, no. 6, pp. 436–447, 2012.

[23] B. Fluri, E. Giger, and H. C. Gall, "Discovering patterns of change types," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, (Washington, DC, USA), pp. 463–466, IEEE Computer Society, 2008.