Mining Change Histories for Unknown Systematic Edits

Molderez, Tim Christiaan; Stevens, Reinout; De Roover, Coen

Published in:
Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)

DOI:
10.1109/MSR.2017.12

Publication date:
2017

Document Version:
Accepted author manuscript

Mining Change Histories for Unknown Systematic Edits

Tim Molderez, Reinout Stevens and Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
Email: {tmoldere, resteven, cderoove}@vub.be

***Abstract*—Software developers often need to repeat similar modifications in multiple different locations of a system's source code. These repeated similar modifications, or systematic edits, can be both tedious and error-prone to perform manually. While there are tools that can be used to assist in automating systematic edits, it is not straightforward to find out where the occurrences of a systematic edit are located in an existing system. This knowledge is valuable to help decide whether refactoring is needed, or whether future occurrences of an existing systematic edit should be automated. In this paper, we tackle the problem of finding unknown systematic edits using a closed frequent itemset mining algorithm, operating on sets of distilled source code changes. This approach has been implemented for Java programs in a tool called SysEdMiner. To evaluate the tool's precision and scalability, we have applied it to an industrial use case.**

***Index Terms*—Systematic edits; change distilling; frequent itemset mining.**

I. Introduction

While developing software, developers can repeatedly perform similar, non-identical changes to a system's source code. Such a group of similar changes, also called *systematic edits* or systematic code changes [1], can be performed for several reasons: adapting code to a changed API, migrating to a different library/framework, refactoring, performing routine code maintenance tasks, fixing multiple occurrences of the same bug, implementing/modifying crosscutting concerns, code cloning, making changes in multiple branches/variants of a system, ...

An example of a systematic edit is given in Fig. 1. It illustrates a small refactoring of the `XmlParser` class, where the file to be parsed is now passed in via the `parse` method instead of the constructor. As the application may have several locations where an `XmlParser` is used, each of them must be changed in a similar manner. In other words, Fig. 1 shows two *instances* of the same systematic edit.

In this paper we present an automated approach that can find existing systematic edits in a given source code repository. This information is valuable in a number of different applications:

Detecting error-prone code - Due to their repetitive nature, performing systematic edits manually can be tedious and error-prone. By repeating a change that is similar, but not always identical, it is understandable that mistakes are easy to make. While these mistakes are often also easy to fix once found, finding them can be difficult. Knowing where the systematic edits are in a system gives developers an idea of which areas of the source code are prone to this type of errors.

```
  public User createUser(String userXml){
-   XmlParser uP = new XmlParser(userXml);
+   XmlParser uP = new XmlParser();
    uP.setSchema(userXsd);
-   uP.parse();
+   uP.parse(userXml);
    ...
  }

  public Event createEvent(String eventXml){
-   XmlParser eP = new XmlParser(eventXml);
+   XmlParser eP = new XmlParser();
    eP.setSchema(eventXsd);
    eP.setVerbose(false);
-   eP.parse();
+   eP.parse(eventXml);
    ...
  }
```

Fig. 1. Two instances of a systematic edit

Informed refactoring - A closely related application: knowing where systematic edits are located can also be helpful to decide whether or not refactoring the system is beneficial. If a similar change needs to be made in many locations, this can indicate the system's design should be improved. For example, in an application that generates reports, if changing the formatting of the entries in a report requires a change for every entry, this should be refactored.

Transformation generation - Finally, it also is possible that a systematic edit that was found can recur in the future. That is, the similar change performed in this systematic edit may need to be repeated again, e.g. because it is part of a routine maintenance task. In such a case, automating the systematic edit by means a program transformation would be beneficial. Knowledge of all existing instances of a systematic edit can be used to reduce the effort of specifying a program transformation. All instances could be (semi-) automatically generalized into a program transformation with tools such as LASE [2] or RASE [3].

While developers may have some notion of the systematic edits they performed in their own code, it would require a lot of effort to manually locate all existing systematic edits in a project. Consequently, an automated approach is required. Our approach to finding existing systematic edits involves using two techniques: first, we make use of a change distiller for Java that obtains all *fine-grained source code changes* that occur in a given git repository. To be more precise, source

code changes at the level of abstract syntax tree (AST) nodes: inserting nodes, removing them, moving them or updating their value. These fine-grained changes provide an additional level of precision compared to traditional, more coarse-grained, line-based diffs. Second, we use these fine-grained changes as the input to a closed frequent itemset mining algorithm, which looks for similar sets of fine-grained changes according to certain criteria to group such changes into sets, and a definition of what is considered similar. These similar sets of fine-grained changes then correspond to a systematic edit.

This approach has been implemented as an Eclipse plugin called SysEdMiner[1], which supports git repositories containing Java source code. To evaluate the tool, we have applied it to multiple repositories of TP Vision Belgium, an industrial partner for the research project this work is part of, to gauge the tool's correctness, usefulness and scalability (in terms of project and commit size).

After presenting an introduction to the ChangeNodes change distiller in Sec. II, this paper makes the following contributions:

- An automated approach to find systematic edits (Sec. III)
- Grouping and equivalence criteria (Sec. IV) to configure how fine-grained and strict a systematic edit should be
- An evaluation of the approach on an industrial use case (Sec. V)

II. Background on Change distilling

There are three main options to obtain changes to source code, so we can analyze them to find any systematic edits: traditional line-based diffs, change distilling and change logging. Line-based differencing (e.g. through the `git diff` command) can be used to indicate which lines have been added or removed between two versions of a file. While it is a relatively simple and fast way to obtain changes, the changes themselves are rather coarse-grained. For example, when only a variable name is changed in a lengthy statement, the entire line still is considered to be changed. Both change logging and change distilling are more fine-grained techniques, as changes are expressed at the level of AST nodes. Nodes can be inserted into an AST, removed, moved, as well as updated. A change distiller can compare two versions of a piece of source code; it will infer a possible sequence of AST-level changes, also called an edit script. When the changes in an edit script are applied in order, the original code is transformed into the modified code. Whereas a change distiller obtains its changes after-the-fact, a change logger will record all changes "live", as the developer is making changes to the code. Given these three options, we opted for change distilling, as it produces fine-grained changes, and can be applied to any existing source code repository without requiring additional software (i.e. a change logger) during development. Note that throughout this paper, we will refer to the output of a change distiller as fine-grained changes, or simply changes.

[1]The source code of SysEdMiner is available online at: https://gitlab.soft.vub.ac.be/tmoldere/sysedminer

The change distilling tool used in our approach is called ChangeNodes [4], [5] , which is an adaptation of ChangeDistiller [6] that operates on Eclipse JDT AST nodes. Both ChangeNodes and ChangeDistiller implement the tree differencing algorithm presented by Chawathe et al. [7].

A. JDT abstract syntax trees

We will first discuss the representation of Eclipse JDT ASTs, followed by more precise definitions of each type of changes to these ASTs, produced by ChangeNodes. Accessing the children of AST nodes is done through *properties*. For example, an `IfStatement` has three properties; an `expression` property, a `thenStatement` and an `elseStatement` property. Some properties may return a collection instead of a single item. These are called *list properties*. Some properties are mandatory, meaning that the AST node must always have a non-null value for them. The `name` property of a `MethodDeclaration` node is an example of a mandatory property. Mandatory properties ensure that every AST always represents syntactically legal Java code. Because ChangeNodes takes this into account when applying a sequence of fine-grained changes to an AST, all intermediate steps are valid Java ASTs as well.

We also require the notion of a *minimal representation* of an AST node. A minimal representation of a node is that node with no values for its non-mandatory properties, and a minimal representation of the values of mandatory properties. For example, the minimal representation of a `MethodDeclaration` is a method with a name, but without arguments, body, etc...

B. ChangeNodes edit scripts

Having discussed the representation of Eclipse JDT ASTs, we can now have a closer look at the output of ChangeNodes. The tool produces the following four types of changes:

- **`insert(node,parent,property,index)`**
  An AST node `node` is inserted as `property` in node `parent`. In case `property` is a list property, the node is inserted at `index`. Note that `parent` may not initially exist yet, and is only created by another insertion change. In this case, `parent` refers to that change.
- **`move(node,parent,property,index)`**
  A `node` is moved to location `property` of `parent`. In case `property` is a list property, the node is moved to `index`.
- **`update(node,property,value)`**
  The value of node `node` at location `property` is updated to `value`. This `property` must be a simple property, indicating that `value` is not an AST node but a literal (of type String, int, double, boolean, ...).
- **`delete(node,property,index)`**
  A `node` and its complete subtree are removed. If `property` is a list property, `index` indicates the index of `node'` in its list.

```
  public class Point {
    private int x;
    private int y;
    public double computeDistance(Point p) {
+     if (this.equals(p)) return 0;
      double dX = this.computeDeltaX(p);
      double dY = this.computeDeltaY(p);
      return Math.sqrt(Math.pow(dX, 2) + Math.pow(dY, 2));
}
    public double computeDirection(Point point) {
+     if (this.equals(point)) return 0;
      double dX = this.computeDeltaX(point);
      double dY = this.computeDeltaY(point);
      return Math.atan2(dY, dX) * 180 / Math.PI;
    }
  ... }
```

Fig. 2. Running example: add equality test

Both a move and insert produce minimal representations of an AST node; inserting a node will only result in a minimal representation of that node being added, and thus not the complete subtree. A move results in moving the minimal representation of that node. Its original location is replaced by a placeholder node that still contains the subtree located at `node'`. The subtrees of these nodes will be introduced by later change operations.

To make these fine-grained changes more concrete, consider the example in Fig. 2, where two new `IfStatements` are inserted. Note that we will reuse this example throughout the paper. A valid sequence of fine-grained changes that the change distiller may produce for this example is the following:

C1: **insert**(`IfStatement if(){}`, body of computeDistance, `statements`, 0)
C2: **insert**(`MethodInvocation equals()`, **C1**, `expression`, -)
C3: **insert**(`ThisExpression this`, **C2**, `expression`, -)
C4: **insert**(`SimpleName p`, **C2**, `arguments`, 0)
C5: **insert**(`ReturnStatement return;`, **C1**, `thenStatement`, -)
C6: **insert**(`NumberLiteral 0`, **C5**, `expression`, 0)
C7-C12: *(Analogous to **C1-C6**)*

Note that each of these changes insert a minimal representation of an AST node. For example, in change **C1**, a new empty `IfStatement` is created. As the `thenStatement` property of an `IfStatement` is mandatory, the minimal representation also includes a `Block` node to represent the empty then-branch.

III. Overview of the Approach

Given a specific commit from a source code repository, we can now use ChangeNodes to obtain all fine-grained changes that occur within that commit. The next step of our approach is to use these changes as input to a mining algorithm. This section will first give an overview of our approach to finding systematic edits. Next, we go into more detail why we opted for closed frequent itemset mining, and how it maps to our problem domain.

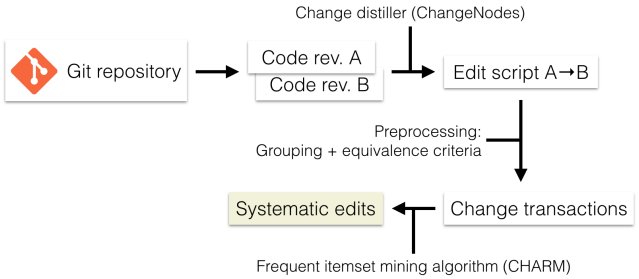

Fig. 3. Overview of the approach

A. Overview

The main steps of our approach are illustrated in Fig. 3. First, from a given git repository, each commit is analyzed separately to search for any systematic edits. For a given commit, the before and after version of each file in that commit is passed to the ChangeNodes change distiller. This results in an edit script, i.e. a sequence of fine-grained changes that, when applied in order, replays all modifications in this commit.

Before we can start mining these changes for systematic edits, a preprocessing step is needed to create suitable input for the mining algorithm. This preprocessing step consists of two components: first, it separates all changes into transactions. A transaction essentially is a set of changes. This separation into transactions happens according to a *grouping criteria*. In this paper, we chose to create a transaction for each method that was modified in a commit. That is, all fine-grained changes that occur within one method are grouped together in one transaction.

The second component of the preprocessing step consists of "generalizing" all transactions, according to equivalence criteria. Equivalence criteria determine when two changes are considered equivalent. Because a transaction represents a set of changes, and a set cannot contain duplicate/equivalent items, the equivalence criteria are needed to generalize this transaction. That is, to remove changes from the transaction until the transaction no longer contains any equivalent changes. Both the grouping and equivalence criteria are detailed in Sec.IV-A.

Finally, once the preprocessing step is complete, the transactions can be mined for systematic edits. To do this, we make use of the CHARM [8] closed frequent itemset mining algorithm. This algorithm will look for sets of changes that frequently occur together in different transactions, which effectively corresponds to a systematic edit.

B. Context

Before describing our approach in more detail, we should motivate our choice for using closed frequent itemset mining [9]. This choice is explained by two main characteristics of our problem domain: change ordering and discontinuity.

Ordering - A change distiller returns a *sequence* of changes, i.e. their order matters. For a change logger, this sequence corresponds exactly to the changes performed by the

developer. However, a change distiller typically produces only one of the many possible orderings of these changes, which may or may not correspond to what a developer would do. For example, when adding a new method, the insertions of each statement can happen in any order, as long as the target AST is produced at the end. To favour finding more systematic edits (at the trade-off of finding more false positives as well), we opted to disregard the ordering of changes in the edit script we obtained from the change distiller.

Discontinuity - It also is possible that, within a change sequence, a developer may perform changes that are unrelated to a systematic edit. For example, a systematic edit may involve inserting logging statements in multiple methods, but other unrelated statements may have been inserted in some of these methods as well. In other words, it is possible that edit scripts interleave changes related to a systematic edit with changes that are not related.

Given the two characteristics of ordering and discontinuity, we opted to use a closed frequent itemset mining algorithm [9] over related techniques to look for frequent patterns, such as frequent sequence [10], episode [11] or substring [12] mining. While frequent episode and sequence mining can find patterns in discontinuous data, frequent itemset mining, being set-based, is an order-insensitive technique.

The specific mining algorithm that we used is the CHARM [8] algorithm by Zaki and Hsiao. We chose CHARM as it does not involve candidate generation. Candidate generation mining algorithms would not scale for our problem domain, as modifying only a few lines of code can already result in a substantial amount of fine-grained changes.

C. Representing systematic edits as itemsets

To define the mapping between frequent itemset mining and finding systematic edits, we will first describe the terminology used in frequent itemset mining in more detail.

In frequent itemset mining, the aim is to find sets of items that frequently occur together in different transactions. In our case, we aim to find sets of changes that frequently occur together. To be more precise, let the *item base* $I = \{i_1, \ldots, i_m\}$ be a set of items. Any subset T of I is an *itemset*. A *transaction* $t = \langle tid, T \rangle$, where tid is a unique transaction identifier and T is an itemset. A transaction $\langle tid, T \rangle$ *contains* an itemset S if $S \subseteq T$. A *transaction database* D is a set of transactions. The *support* of an itemset in D is the number of different transactions in D that contain that itemset. A *frequent itemset* is an itemset where its support is above a certain threshold c. The problem of frequent itemset mining can now be defined as: given a transaction database and a threshold, identify all frequent itemsets.

In our context of finding systematic edits, an item corresponds to a fine-grained change and a systematic edit is a frequent itemset. That is, a systematic edit is represented as a set of changes that frequently occurs in different transactions. An instance of a systematic edit then corresponds to an occurrence of a frequent itemset in one transaction. The transaction database is obtained by dividing all fine-grained changes into groups. As mentioned before, the grouping criteria we used is to group all changes by method: all changes that occur within a method form the itemset T of a transaction $\langle tid, T \rangle$. The method itself can serve as the transaction identifier tid.

Finally, note that CHARM is a *closed* frequent itemset mining algorithm. The algorithm will only report closed frequent itemsets, which are frequent itemsets for which there are no superitemsets with the same support. In other words, there is no redundant information in the algorithm's output: it will only report systematic edits such that there are no larger/super systematic edits with the same number of instances.

IV. Grouping and equivalence criteria

As mentioned in Sec. III-A, a preprocessing step is needed to produce transactions that can be mined by the CHARM algorithm. The grouping and equivalence criteria that are involved in this step can greatly influence which systematic edits will be found during the mining process. Sec. IV-A first focuses on grouping criteria, followed by Sec. IV-B to discuss equivalence criteria.

A. Grouping criteria

To separate the changes produced by ChangeNodes into transactions, we chose to group all changes by method. That is, given a specific commit, there is one transaction for every method that is modified in that commit. Any change that inserts, removes, moves or updates a node within a particular `MethodDeclaration`, is part of the transaction for that `MethodDeclaration`. The method that a change is part of, also is referred to as the *container* of that change.

To illustrate, consider the example of Fig. 2. Recall that a transaction is denoted $\langle tid, T \rangle$, with tid a transaction identifier and T a set of changes. In this example, grouping the changes by the method they affect results in two transactions: $\langle$`Point.computeDistance`$, \{C1, C2, C3, C4, C5, C6\}\rangle$ and $\langle$`Point.computeDirection`$, \{C7, C8, C9, C10, C11, C12\}\rangle$.

In general, the grouping criteria can be configured such that changes are grouped by the subtree in which they occur, rooted at a specific type of AST node. The type of this root, i.e. the type of the container, has an influence on the systematic edits found by the mining algorithm.

First, changes can be discarded due to the grouping criteria. In our case, any changes that do not occur in a `MethodDeclaration` are lost. For example, changes to field declarations or import declarations.

Second, a frequent itemset cannot contain more items than the transactions in which it is found. Consequently, because changes are grouped per method, an instance of a systematic edit can only contain changes in one `MethodDeclaration`. If each instance of a systematic edit would involve changes in multiple methods, the approach would find multiple systematic edits instead. However, if transactions are grouped per `CompilationUnit`, this case would be handled correctly.

Third and finally, an itemset can only be considered frequent if it is found in multiple transactions. This means that, if the example of Fig. 2 would be grouped per `CompilationUnit`,

the similar modifications to both methods would end up in one transaction, and no systematic edit would be found.

Given these limitations, we opted for grouping by methods in this paper, as it seems reasonable that instances of systematic edits most commonly occur between different methods.

B. Equivalence criteria

Aside from grouping changes into transactions, equivalence criteria are needed to determine when two changes should be considered equivalent. Without equivalence criteria, each change is different from all other changes. Consequently, no transactions would share changes with other transactions, and no frequent itemsets could be found. As such, equivalence criteria make it possible to relate fine-grained changes to each other by searching for shared commonalities.

The equivalence criteria we have used in this paper are a conjunction of three conditions. In short, two changes are considered equivalent, if: their change type is equal, the node being modified is *structurally equal*, and the path between the container and the node being modified should be equal. We will now describe these conditions in more detail:

- **Change type equality** - The change type of both changes (insert, remove, move or update) must be equal.
- **Structural subject equality** - The subject of a change refers to the node being modified. In case of an insert, it is the new node being inserted. For a removal, it is the node being removed. For a move, it is the node being moved. For an update, it is the node for which an attribute is updated. To test whether two subjects are structurally equal, their ASTs (with the subject as root) are traversed. Both ASTs must be fully equal, except that the value of simple properties (identifiers, literal values, ..) are not compared. Everything else is compared: node types, property names and indices (for list properties).
- **Context location equality** - Whereas change type equality focuses on *how* a change was made, and change subject equality takes care of *what* was changed, the context condition is about *where* a change is applied. This context equality is useful to prevent noise/false positives. For example, when context is taken into account, several insertions of `null` literals will not automatically result in a systematic edit. The context of a change corresponds to the container that a change is part of. Two changes have an equivalent context if and only if they have the exact same location within the transaction. This location is specified as the path between the change's container node and the subject node, i.e. a list of property names (including indices for list properties).

The equivalence criteria are used directly after the grouping criteria have created transactions. At this point, these transactions contain the changes produced by ChangeNodes as-is. After applying the equivalence criteria, the changes in each transaction will be generalized. That is, an abstraction will be applied to each change, such that it is now represented by a triple $\langle changeType, structuralSubject, location \rangle$, where the three elements respectively correspond to the equivalence criteria's conditions: change type, structural subject AST and change location. As an example, changes **C1** - **C6** of Fig. 2 are generalized as follows:[2]

C1: $\langle$`insert, if(){}, body : statements − 0`$\rangle$
C2: $\langle$`insert, equals(), body : statements − 0 : expression`$\rangle$
C3: $\langle$`insert, this,`
`body : statements − 0 : expression : expression`$\rangle$
C4: $\langle$`insert, ∗,`
`body : statements − 0 : expression : arguments − 0`$\rangle$
C5: $\langle$`insert, return;, body : statements − 0 : thenStatement`$\rangle$
C6: $\langle$`insert, ∗,`
`body : statements − 0 : thenStatement : expression`$\rangle$

Recall that, in this example, there are two transactions: $\langle$`Point.computeDistance`$, \{C1, C2, C3, C4, C5, C6\}\rangle$ and $\langle$`Point.computeDirection`$, \{C7, C8, C9, C10, C11, C12\}\rangle$. Once the second transaction is generalized, the generalized representation of **C7**-**C12** will look identical to **C1**-**C6**. For example, **C1** is `identical` to **C7**: both are inserts of `if(){}` at location `body:statements-0`. This means that, when using the generalized representations, it is clear that **C1** and **C6** are equivalent changes.

By using the equivalence criteria to generalize changes, we can establish an equivalence relation between different changes. We define an equivalence relation $\sim$ over the set of fine-grained changes, which considers two changes equivalent if they share certain commonalities. The equivalence relation has to fulfill certain requirements to yield valid results. In general, there are three basic properties that should hold. Given changes a,b and c:

- Reflexivity ($a \sim a$). Each change is equivalent with itself.
- Symmetry ($a \sim b \rightarrow b \sim a$). If a is equivalent to b, then b must also be equivalent to a.
- Transitivity ($a \sim b \wedge b \sim c \rightarrow a \sim c$). If a is equivalent to b, and b equivalent to c, then a must be equivalent to c.

Finally, it is important to note that, when generalization is applied, it is possible that two changes in the same transaction will have the same generalized representation. Because a transaction represents a *set* of items, generalization can reduce the number of changes in a transaction.

V. Evaluation

To evaluate our approach and its implementation in SysEdMiner, we have used it on an industrial use case at the company TP Vision Belgium. TP Vision is a wholly-owned subsidiary of TPV, an internationally-renowned PC monitor and TV manufacturer serving as original design manufacturer for well-known TV and PC brands in the industry. TP Vision oversees Philips TV business in most regions of the world.

We were contacted by TP Vision Belgium to analyse some of their projects and get a better idea of to what extent

[2]Note that the simple properties in **C4** and **C6** have been abstracted away with a $*$ wildcard character.

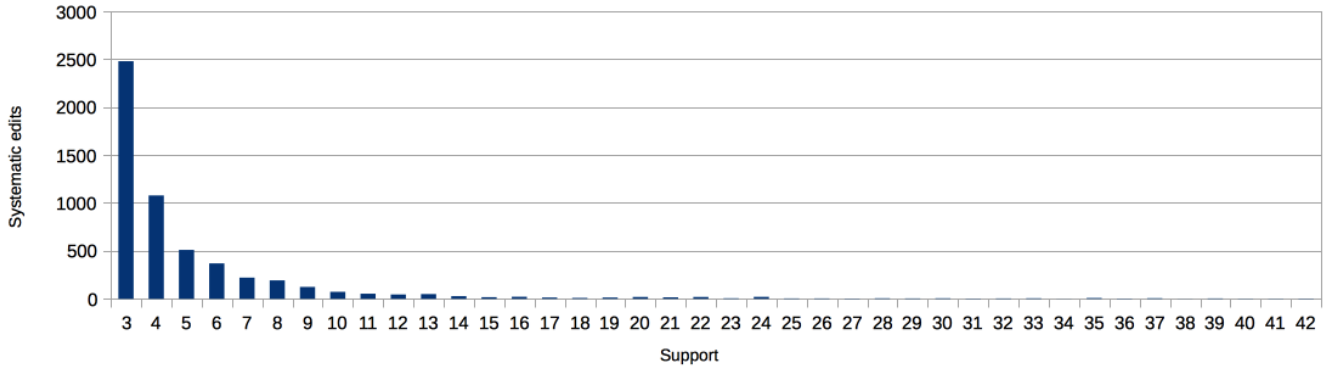

Fig. 4. Systematic edits per support level

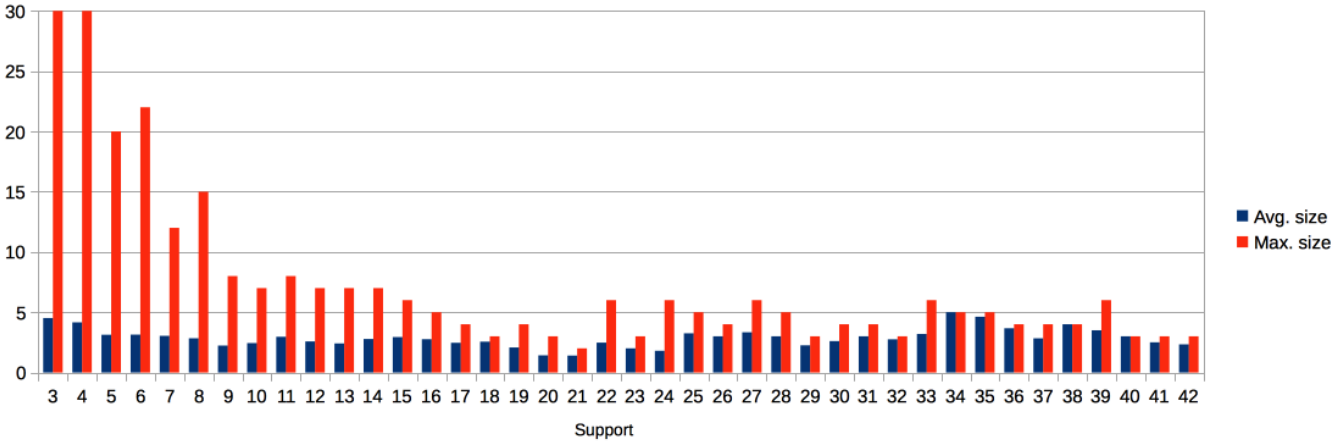

Fig. 5. Maximum and average instance sizes

developers perform repetitive modifications: Are there many systematic edits? How many instances does a systematic edit usually have? What is the average size of an instance?

We analysed 51 of TP Vision Belgium's git repositories with Java code, most of which contain Android applications. For each repository we had access to a year's worth of development activity. The total number of commits of all projects is 43756. In terms of performance, the actual mining of changes usually is much quicker than change distilling: while this greatly depends on how much code was changed in a commit, change distilling roughly took 30 seconds per commit on average, whereas mining only requires a second. If all commits were processed sequentially, the entire analysis would then take 15 days. However, as multiple commits can be independently processed in parallel, and we made use of an (Intel Core i7) 8-core processor, results can ideally be obtained 8 times quicker as well (given sufficient memory, which we did not measure).

The configuration we used to run our SysEdMiner tool has been mostly discussed in Sec. IV: changes are grouped into transactions per `MethodDeclaration`; changes are equivalent if the change type is equal, the subject is structurally

```
:body:statements-0:expression:arguments-0

org.droidtv.epg.EpgInfoSingleton.getOperatorAntenna
+ Log.d("getOperatorAntenna() 0-None 1- CAM"+
     operatorAntenna);
- Log.d(TAG,"getOperatorAntenna() 0-None 1- CAM"+
     operatorAntenna);

org.droidtv.epg.bcepg.epgui.EpgOptionsMenu.
     getControllablityOfNode
+ Log.d("getControllablityOfNode");
- Log.d(TAG, "getControllablityOfNode");

org.droidtv.epg.bcepg.epgui.EpgOptionsMenu.
     getSelectionIndex
+ Log.d("getSelectionIndex" + nodeIndexValue);
- Log.d(TAG, "getSelectionIndex" + nodeIndexValue);
```

Fig. 6. Three instances of deleting a parameter

equal and the context location is equal. The minimum support (minimum number of instances) is set to 3. Note that, while analysing one commit is single-threaded, the analysis can be sped up by analysing multiple commits, or multiple projects, simultaneously.

First, to gauge whether our tool can correctly find systematic

edits, we sampled 100 systematic edits from different projects with different support levels. These samples were chosen at random, making sure to include some samples for each project. We manually inspected these systematic edits by mapping their fine-grained changes back to the source code, presenting them in a diff-like format, and comparing the different instances. We identified that 78 out of 100 samples were indeed systematic edits. We did not find a particular correlation between support level and which results were correct. A simple example of a systematic edit that we found is given in Fig. 6, which shows 3 (out of 161) instances where the first parameter of a `Log.d` call is removed. At the top of the systematic edit, the context location is shown once, to determine which AST node exactly is modified. For each instance, the transaction identifier (method name) is shown, and the line that was affected by the change.

Those that were not identified would of course still abide by the grouping and equivalence criteria we chose, but it occurs that the equivalence criteria can overgeneralize and abstract away too much information. For example, if a string parameter is modified in many places, but there is no structure among the different places regarding the new value of the string parameter, we do not consider it a systematic edit. Finally, note that we only studied the tool's correctness, not its completeness. As is it is rare to find projects where all systematic edits are known in advance, and because the tool already can be quite useful even if it is not complete, we did not investigate this further.

After manually examining our sample of systematic edits, we moved onto analysing all data produced by SysEdMiner: Fig. 4 presents an overview of how many different systematic edits were found per support level. For instance, there are 2479 different systematic edits at support level 3. In other words, 2479 systematic edits in which a similar modification is performed exactly 3 times. When adding up the number of systematic edits for each support level, we end up with a total of 5474 systematic edits. While this is a reasonably substantial number, this figure does indicate that the vast majority of systematic edits has a low number of instances. Note that we do not show the entire X-axis; there are a small amount of additional systematic edits with support levels up to 278.

Fig. 5 tells us more about the size of each instance, per support level. The size of one instance in a systematic edit is measured in terms of fine-grained changes, which also corresponds to the size of the frequent itemset. This figure shows both the average and maximum instance size per support level. We can conclude that the average size remains quite consistently between 2-4 changes, regardless of the support level. However, the maximum size is different: it is much higher in lower support levels. (Support levels 3 and 4 both have a maximum size of 177.) This indicates that repeating a larger amount of code a few times can be tolerated sometimes. It also corresponds to the intuition that it is unlikely that large amounts of code will be repeated many times, as a developer would notice quickly.

VI. Limitations

In this section we discuss the limitations of our approach to detecting unknown systematic edits in distilled change sequences.

Section IV-A already discussed some considerations to applying frequent itemset mining to distilled change sequences. Currently, we group changes based on a certain AST node type, such as a method declaration or a class. Consequently, the approach currently cannot detect systematic edit instances that span multiple files. To be exact, the instances would still be detected, but they are split into multiple systematic edits. As such, we need to investigate whether we are missing other kinds of groupings that are not based on an encompassing AST node type. For example, we could incorporate data- and control flow information to group changes that affect the same data elements.

Our approach relies on frequent itemset mining. Frequent itemset mining requires that its itemsets are a *set* of items. Currently, our approach discards a change when an equivalent change is already present in its corresponding itemset. As a result, multiple instances of the same systematic edit, or systematic edits consisting of multiple, equivalent edits cannot be detected. The current evaluation results show that most systematic edits have quite small instance sizes, which can indicate that our current equivalence criteria may be too strict. As a remedy our tool can be easily configured to use different equivalence criteria.

We rely on a change distiller to procure change sequences. A change distiller relies on heuristics to determine what nodes are considered to be equal between two ASTs. Examples of such heuristics are the Levenshtein distance of a string representation of an AST node. As a result, two instances of a systematic edit may be represented by two different change sequences that do not share equivalent changes. Thus, our approach relies on the assumption that instances of the same systematic edits are represented by similar change sequences.

VII. Related Work

One area of related work is that of code clone detection tools, such as CBCD [13], CCFinder [14], dup [15] or Baxter (1998) [16]. While these tools typically are focused on analysing one version of the code, cloned code often indicates a systematic edit. However, clone detection tools cannot detect all systematic edits: a single instance of a systematic edit can be spread across different locations of the code. For example, consider a systematic edit where multiple fields and their getter/setter methods are added to classes. One instance would then correspond to the addition a field and its getter/setter, which may be in different locations of the class. To recognize this type of systematic edits, change information is necessary.

Another area of related work is that of automated API migration, which typically involve systematic edits to adapt all uses of one API into another API. The techniques used in this area can be related to ours, as it often involves creating a database of changes, and looking for patterns in this database. The main difference is that such change databases tend to

be purpose-built for API-level changes only, whereas our technique is more general-purpose, and can consider any fine-grained change. Our equivalence and grouping criteria can be configured to focus on API-level changes, but it is also suitable to detect a wider range of systematic edits. Diff-Catchup [17] detects migrations by looking for API differences at the level of UML models. The SemDiff [18] tool can recommend changes to API calls by looking for existing methods in which the call was removed. LibSync [19] extracts an API usage graph for two versions of a system, and uses frequent itemset mining to look for migration patterns. The work of Uddin et. al [20] uses clustering techniques to detect API usage patterns among all method/field-level changes that reference an API.

The most closely related work, where change data is mined, is that of Negera et. al [21] and [22] Kreutzer et. al. First, Negera et. al [21], use frequent closed itembag mining with overlapping transactions to identify frequent code change patterns (systematic edits) in a sequence of fine-grained code changes. The main difference with our work is that the fine-grained changes are obtained with a change logger rather than a change distiller. A change logger records all source code changes as they are performed by the developer, while a distiller tries to infer fine-grained changes after the fact, based on VCS information. The mining algorithm used is based on the CHARM [8] algorithm for closed frequent itemset mining. In order to allow code change pattern mining, the CHARM algorithm was modified to allow overlapping transactions, making it possible to process a continuous sequence of code changes ordered by timestamp, without any previous knowledge of where the boundaries between patterns of transformations are. Furthermore, the algorithm was updated to allow itembags instead of itemsets. This was necessary since a high-level program transformation may contain several instances of the same kind of code changes. The algorithm is able to identify repetitive code change patterns that may correspond to some previously unknown high-level program transformations. Evaluation of the algorithm showed effectiveness, usefulness and scalability by running the algorithm on previously collected data involving 23 participants with 1520 hours of development. Feeding the miner with this data resulted in a set of change patterns together with all occurrences of each pattern. As such, they have identified 10 kinds of previously unknown high-level program transformations.

Instead of relying on a change logger to provide change data, we collect change data by comparing version control snapshots. Consecutively, we use standard frequent itemset mining instead of frequent closed itembag mining with overlapping transactions. Also, our approach allows automatic transformation of source code according to a change pattern.

In the work of Kreutzer et. al [22], two similarity metrics are used to detect systematic edits: one based on agglomerative hierarchical clustering, and one based on the DBSCAN clustering algorithm [23]. The latter tends to find more systematic edits, but requires significantly more time. Both techniques operate on distilled changes produced by the ChangeDistiller tool. However, the notion of code changes used in this paper is more coarse-grained than our work, as the distilled fine-grained changes are bundled together into higher-level statement-level changes. Due to this design choice, some systematic edits may be missed, but this is traded in for better performance.

Next to Negera et. al and Kreutzer et. al, there also is less closely related research on applying standard data mining approaches to source code and source code changes. Both Ying et. al [24] and Zimmerman et. al [25] address the difficulty developers face in finding relevant source code fragments for a certain modification. In their work, change patterns are defined as files that have changed together frequently enough. A recommender tool tracks files changed by the developer: if a developer changes a set of files, the approach recommends a set of files that will likely need to be changed as well. To allow recommendation, association rule mining is used over pre-processed CVS data. Zimmerman et al. [25] use the same idea but allow working below file-level granularity. Mulder and Zaidman [26] proposed a method for identifying cross-cutting concerns in software systems with software repository mining. As cross-cutting concerns are scattered throughout the code base, modifying them requires changes in multiple files. This allows the use of frequent itemset mining for finding files that were frequently committed simultaneously. Next to this file-level mining a more expensive, but more fine-grained, method is proposed for mining methods that are frequently changed together. Li et. al [27] propose CP-miner, a tool for finding copy-paste and related bugs in operating system code. CP-Miner first identifies copy-pasted code and then performs bug finding. To achieve the former, the program is parsed, resulting in a stream of tokens. Tokens are then assigned numerical values, which allows formulation of the problem of identifying copy-pasted code as a sequential pattern mining problem on this stream of numerical values.

All of these approaches provide identification and recommendation but do not allow automatically modifying source code based on earlier code changes.

VIII. Conclusion and Future Work

In this paper we presented our approach to find unknown systematic edits, using the ChangeNodes change distiller to obtain fine-grained changes, which are then preprocessed by grouping and equivalence criteria, and finally mined with the CHARM algorithm. To evaluate the approach, we have applied it on 51 git projects in an industrial use case. Our tool SysEdMiner does find a substantial amount of systematic edits and, by manually sampling the results, found that 78 out of 100 samples are correct. We also found indications that larger instances are more likely to occur at low support levels rather than higher ones.

There are multiple directions of future work: a straightforward one is to further explore the design space of using different grouping and equivalence criteria. The tool was currently applied to a data set provided by one company; this can be further expanded with additional data from open-source projects, for example. Another one is to experiment

with finding changes across multiple, if not all, commits. This does not fundamentally change our approach, but the mining algorithm may not scale to such a large set of fine-grained changes. One option is to use abstraction to reduce the total number of changes. Another could be to use a sliding window over a large list of commits. This is based on our assumption that, if a systematic edit occurs in one commit, it is most likely that additional instances of that systematic edit would happen in the next few commits, rather than much later. Another direction of future work would focus on the applications of systematic edits that are found. They could be used as a metric to automatically decide whether or not refactoring the code is necessary, to avoid future instances of a systematic edit. Alternatively, the instances could be used to (semi-)automatically generate a program transformation that can perform future instances of a systematic edit. Similarly, it may be possible to create an approach that detects such future instances, and determine if any errors were made.

ACKNOWLEDGMENTS

We would like to thank TP Vision Belgium for making this study possible and for providing their repository data. We would also like to thank Arvid De Meyer for his contributions to the initial prototype of the SysEdMiner tool in the context of his master's thesis.

REFERENCES

[1] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070531

[2] 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*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 502–511. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486855

[3] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does Automated Refactoring Obviate Systematic Editing?" ACM Association for Computing Machinery, May 2015. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=244802

[4] L. Christophe, R. Stevens, and C. De Roover, "Prevalence and maintenance of automated functional tests for web applications," in *Proc. of the Int. Conf. on Software Maintenance and Evolution (ICSME14)*, 2014.

[5] R. Stevens, "A declarative foundation for comprehensive history querying," in *Proc. of the 37th Int. Conf. on Software Engineering, Doctoral Symposium Track (ICSE15)*, 2015.

[6] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *Transactions on Software Engineering*, vol. 33, no. 11, 2007.

[7] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proc. of the Int. Conf. on Management of Data (SIGMOD96)*, 1996.

[8] M. J. Zaki and C.-J. Hsiao, "Charm: An efficient algorithm for closed itemset mining," in *Proceedings of the 2002 SIAM international conference on data mining*. SIAM, 2002, pp. 457–473.

[9] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, *Discovering Frequent Closed Itemsets for Association Rules*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 398–416. [Online]. Available: http://dx.doi.org/10.1007/3-540-49257-7_25

[10] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the Eleventh International Conference on Data Engineering*, Mar 1995, pp. 3–14.

[11] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovering frequent episodes in sequences extended abstract," in *1st Conference on Knowledge Discovery and Data Mining*, 1995.

[12] P. Weiner, "Linear pattern matching algorithms," in *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, Oct 1973, pp. 1–11.

[13] J. Li and M. D. Ernst, "Cbcd: Cloned buggy code detector," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 310–320.

[14] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, Jul 2002.

[15] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of 2nd Working Conference on Reverse Engineering*, Jul 1995, pp. 86–95.

[16] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, Nov 1998, pp. 368–377.

[17] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 818–836, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70747

[18] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 19:1–19:35, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/2000799.2000805

[19] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 302–321. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869486

[20] G. Uddin, B. Dagenais, and M. P. Robillard, "Temporal analysis of api usage concepts," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 804–814. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337318

[21] 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 (ICSE14)*, 2014.

[22] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen, "Automatic clustering of code changes," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901749

[23] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise." AAAI Press, 1996, pp. 226–231.

[24] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, Sept 2004.

[25] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, June 2005.

[26] F. Mulder and A. Zaidman, "Identifying cross-cutting concerns using software repository mining," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*. ACM, 2010, pp. 23–32.

[27] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, March 2006.